Porting Ordinary Applications to Blue Gene/Q Supercomputers

Ketan Maheshwari*, Justin M. Wozniak*‡, Timothy G. Armstrong†,
Daniel S. Katz‡, T. Andrew Binkowski§, Xiaoliang Zhong¶,
Olle Heinonen¶, Dmitry Karpeyev‡, Michael Wilde*‡
* *Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, USA*
† *Dept. of Computer Science, University of Chicago, Chicago, USA*
‡ *Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, USA*
§ *Midwest Center for Structural Genomics, Argonne National Laboratory,*
Center for Structural Genomics of Infectious Disease and Computation Institute,
University of Chicago Chicago, USA
¶ *Materials Science Division, Argonne National Laboratory, Argonne, USA*

***Abstract*—Efficiently porting ordinary applications to Blue Gene/Q supercomputers is a significant challenge. Codes are often originally developed without considering advanced architectures and related tool chains. Science needs frequently lead users to want to run large numbers of relatively small jobs (often called many-task computing, an ensemble, or a workflow), which can conflict with supercomputer configurations. In this paper, we discuss techniques developed to execute ordinary applications over leadership class supercomputers. We use the high-performance Swift parallel scripting framework and build two workflow execution techniques–*sub-jobs* and *main-wrap*. The sub-jobs technique, built on top of the IBM Blue Gene/Q resource manager Cobalt's sub-block jobs, lets users submit multiple, independent, repeated smaller jobs within a single larger resource block. The main-wrap technique is a scheme that enables C/C++ programs to be defined as functions that are wrapped by a high-performance Swift wrapper and that are invoked as a Swift script. We discuss the needs, benefits, technicalities, and current limitations of these techniques. We further discuss the real-world science enabled by these techniques and the results obtained.**

1. Introduction

The Argonne Leadership Computing Facility (ALCF) hosts *Mira*, a 50K-node IBM Blue Gene/Q (BG/Q) with peak performance of 10 PetaFLOPS. While this system was designed for and is well-suited to large applications programmed using advanced parallelization libraries such as MPI and/or OpenMP, it lacks many features that are taken for granted on smaller-scale clusters and thus is not well-suited to classes of scientific applications such as ensemble or parameter sweep studies and workflow style computation, which are increasingly observed in scientific applications. For example, the minimum job size on Mira is 512 nodes (8192 cores, or half a rack), which is too large for some user requirements or for some application designs. On other systems, the queues may be configured to encourage very large jobs and discourage small jobs, which again may conflict with a science need for many small jobs. Many applications are originally developed with small- and medium-size execution on regular clusters in mind. Some of these applications are later used at large-scale in a many-task computing (MTC) style. (MTC [1] is an emerging computation style wherein the computation consists of a large number of medium-sized, semi-dependent tasks implemented as ordinary programs that are invoked from the command line.)

Porting such applications to leadership class supercomputers can be a significant challenge because current tools require users to manually partition a larger allocation into the appropriate size for their tasks. We propose two solutions for automating this process, which enable program in the Swift programming language to run small tasks on BG/Q.

The first technique automates use of the sub-block job feature of the BG/Q resource manager, Cobalt, which allows jobs to run on sub-blocks of an outer block allocation. Without our technique, a user must manually select a geometry (i.e., a subset of the nodes based on the network topology) and configure related low-level parameters.

The second technique we discuss is "main-wrap." This technique enables applications with command-line interfaces to be executed in an MTC mode over BG/Q systems. The BG/Q compute nodes do not support creation of subprocesses through `fork()/exec()`, so the usual approach of running the program as a separate process does not work. The main-wrap technique bypasses this restriction, enabling large-scale MTC computing on Mira without modifying code to use threads or message passing.

While the sub-jobs and main-wrap techniques both enable users to run their ordinary applications on BG/Q systems, they address two distinct problems. main-wrap addresses a BG/Q operating system level deficiency whereas sub-jobs addresses BG/Q resource management limitations. There are cases where one must be used over the other, for instance, when the source code of an application is not available one, the sub-jobs technique must be used, or where both can be used together, such as when a main-wrapped application

978-1-4673-9325-6/15 $31.00 © 2015 IEEE
DOI 10.1109/eScience.2015.8

must run repeatedly over a parameter space in smaller job sizes than minimum allowed by Mira (512 nodes). We compare and contrast the techniques in detail in Section 4.

1.1. Swift

We use the Swift parallel scripting language as a base and interface it with the BG/Q resource management system. Swift coordinates tasks which are then run directly by the two previously mentioned techniques. Swift [2] is well suited to many-task and workflow-style computation. It supports the use of a variety of computational environments including clusters, grids, supercomputers, and clouds. Two implementations of Swift exist: Swift/K and Swift/T. Swift/K is a high throughput framework which is used to enable the sub-jobs techniques, while the high performance Swift/T [3] is used to enable the main-wrap technique. Swift/K uses an XML dialect to internally represent tasks and their dependencies, while Swift/T uses Tcl.

1.2. Blue Gene/Q Architecture

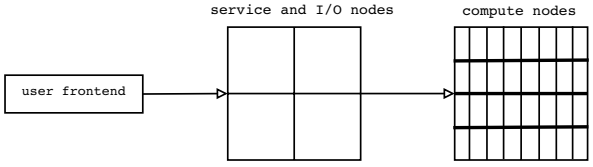

Figure 1. A simplified architecture of the Blue Gene/Q user interaction with compute nodes

The BG/Q machine is configured as three major functional components (see Figure 1.) First, the user front-ends are the nodes that act as login nodes for end-users. The front-ends are connected to the second component, the service and I/O nodes. The service nodes perform system management services such as I/O and execution monitoring, logical node partitioning, and running scheduler services. The I/O nodes provide file and process management services, network socket, and debugging services. The third component is the compute nodes, which execute user programs and interact with service and I/O nodes. The Mira BG/Q is configured as 48 racks of 1024 nodes each. Each rack in turn is configured as two midplanes (or half-racks) of 512 nodes each. Each node is comprised of 16 cores.
The compute nodes in the BG/Q system are interconnected in a five-dimensional torus geometry. This means that each node is connected with ten of its nearest neighbors. This configuration results in efficient internode communication when an application running on a sub-block of a larger outer block of nodes. However, when this happens, the user's choice of which confining topology to use for that sub-block leads to differing run efficiencies.

2. The Sub-job and Main-wrap Techniques

Here we discuss the design and implementation of the two techniques.

2.1. Sub-jobs

The BG/Q runs the Cobalt scheduler for resource management. Cobalt provides a feature through which multiple smaller, repeated, multi-node jobs can be submitted inside a larger multi-node outer job. This feature is called sub-block jobs (or sub-jobs for short.)
Sub-jobs are a convenient job management strategy for applications composed of many small- to medium-sized tasks. For example, docking applications are typically run to find the possible placement of many candidate molecules against one subject molecule, forming a large number of independent small-to-medium-sized tasks. Similarly, many molecular dynamics applications are run as relatively small computations over a large number of configurations, forming independent medium-sized tasks. In such computations, it is convenient to use sub-jobs, because they allow a user to repeatedly run tasks without interacting with the scheduler to request and obtain a new Cobalt block for each task.
However, in order to work efficiently with sub-jobs, users must understand the five-dimensional geometry of the BG/Q compute nodes. Constructing such geometries for different job sizes across applications is very tedious, and keeping track of completed jobs requires additional manual work.
ALCF provides scripts to help the user in this complex choice process, but even with the scripts, running many-task applications is inconvenient because the user must manually do size matching between tasks and blocks, in addition to activities such as booting the sub-blocks, limiting the number of jobs so as not to overload the outer block and, ensuring a time difference of 3-4 seconds between successive job submissions to avoid a possible system freeze (a hardware related requirement of BG/Q systems).
In our design, we abstract such low-level details by embedding and integrating these scripts into a higher level workflow execution framework.

2.1.1. Design. Applications that need to run multiple, simultaneous small jobs with interdependencies, such as those found in workflows, will benefit from using the sub-job technique. The design goal of this technique is to abstract, hide, and separate the complicated resource management process from the workflow. Users must be able to run their existing workflows expressed in Swift/K with little or no modifications under the new sub-jobs technique. This means that the mechanism to handle the nuances of resource management must be handled at the layers in which the Swift/K is exposed to the BG/Q Cobalt scheduler. This is convenient to adapt within Swift/K's execution mechanism, as it is implemented as layered calls (Swift/K's chain of invocation) to a series of components handling specific execution requirements (e.g., task generation, data management, monitoring, etc.) We solve this through a new shell script (*bg_sh*) that forms an interface layer between Swift's resource management system and the BG/Q Cobalt scheduler. The *bg_sh* script replaces application invocation under these circumstances, and the application invocation command line arguments become the arguments of this

script. The *bg_sh* script needs minimal information from users: the size of the outer block and that of a sub-block. This information is passed via regular Unix environment variables. The 3-4 seconds time difference between two invocations must be handled at a higher level because it affects the communication between the BG/Q service and compute nodes. This means that Swift/K's task dispatch mechanism must be adjusted to accommodate this time difference, which happens at a higher level than *bg_sh*.

2.1.2. Implementation. Figure 2 shows Swift/K's chain of invocation for a workflow using the sub-jobs technique. *Swift_invoke* starts the Swift/K script and the *Coaster_service* [4] on the current host and arranges for one or more *workers* (①) to be started on the BG/Q service nodes. The *Coaster_service* submits the *worker* to a service node as a Cobalt scheduler job which connects back (②) to the *Coaster_service* over TCP. As the Swift script runs, it submits tasks to the Coaster service over a TCP connection, which then sends tasks to available workers. The *worker* then invokes the *swift_wrapper* script (③), which sets up a sandbox area for application execution. It moves input data to the sandbox prior to application execution and output data from the sandbox to the working directory where the *swift_invoke* was invoked after execution. The *swift_wrapper* invokes the *bg_sh* (④) script with the application command line specified in Swift/K script source The *bg_sh* script sets up the sub-block environment, manages sub-blocks, and assigns application tasks to the available sub-blocks. *bg_sh* invokes (⑤) the Cobalt command *runjob* on the compute node and invokes (⑥) the application executable, *app_invoke*.
Since the *bg_sh* script is central to our sub-jobs technique, we will describe its invocation and behavior in detail. The script is invoked as:

```
bg.sh exe [preproc] [postproc] args
```

The steps involved in the *bg_sh* script are:

1) Determine the system name and set the environment. ALCF hosts two smaller systems in addition to Mira: Cetus (4 racks) and Vesta (2 racks). This step enables support for the test hosts by setting the appropriate environment for them.
2) Determine "shape" of a sub-block based on the size obtained via externally passed environment variables. The "shape" is a string of the form of `AXBXCXDXE`, which represents the number of nodes in each of the five dimensions of the node-geometry. For instance, for a sub-block size of 64, the shape will be `2X2X4X2X2`, whereas that for a sub-block size of 512, it will be `4X4X4X4X2`. The number in each dimension can change as long as the product matches the size of the sub-block. A symmetrical shape is preferred for optimal inter-node communication. Block sizes larger than 512 do not need the shape and are treated as bootable blocks exclusively.
3) Invoke the BG/Q Cobalt-provided scripts to obtain the corner coordinates of the respective sub-blocks by supplying the sub-block shape. The size of the outer block is automatically determined at runtime by reading the Cobalt-specific environment. As a result, an array of corner coordinates will be obtained.
4) Invoke the user supplied preprocessing script (`preproc`), if any. This script is used to initialize application specific environment.
5) Use the obtained corner coordinates in a loop to invoke the `exe` with `args` obtained from the command line arguments to the *bg_sh* script.
6) If the sub-block size is larger than 512, invoke the block booting routines. As a result, several bootable blocks will be obtained. Boot these blocks and invoke the `exe` with `args`.
7) Invoke the user supplied postprocessing script (`postproc`), if any. This script is used to perform postprocessing activities after the application has run.
8) Finally, invoke the cleanup scripts to reclaim the sub-block by the outer block and optionally shutdown the booted blocks.

Note that the outer block remains active during the lifetime of the *worker* that requested the Cobalt scheduler block via its `qsub` command. The sub-blocks remain alive as long as there are dispatchable tasks available at the *Coaster_service* end. The available sub-blocks are reused for new tasks by assigning a unique index to each sub-block. This index is exported by the *worker* and is used by the *bg_sh* script. The multiplicity of workers and blocks are determined by the Swift configuration. However, most applications can be run in a configuration where one *worker* controls one outer block (obtained via `qsub`) and coordinates with multiple sub-blocks with the *bg_sh* script.

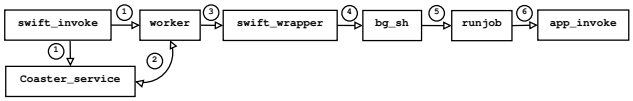

Figure 2. Swift/K sub-jobs chain of invocation

Figure 3 shows an example set of sub-job configurations for a 512-node outer block. The leftmost configuration shows the outer block divided into 8 sub-blocks, each of size 64 nodes. Dividing the outer block into smaller sized sub-blocks will yield a larger number of sub-blocks. In extreme cases, it is possible to have a single sub-block of the same size as the outer block (though this is not useful), or 1-node sized sub-blocks with the number of sub-blocks equal to the size of the outer block (one restriction is that the size of a sub-block must be a power of 2). Similarly, the size of the outer block (and thus the size of overall run) can be between 1 node and the number of nodes in the whole machine. However, sub-jobs work in two different

ways depending on the the size of outer block, 512 nodes or smaller versus larger than 512 nodes.
There are two types of sub-blocks: *prebooted* and *bootable* sub-blocks. As the names suggest, the pre-booted sub-blocks are booted along with the outer block while the bootable sub-blocks needs to be explicitly booted. For outer blocks larger than 512 nodes, bootable nodes must be used, while for outer blocks of 512 nodes or smaller, the user can choose between prebooted and bootable sub-blocks. In Swift/K's implementation of sub-jobs, this distinction is abstracted away and is transparent to end-user. In other words, users only need to specify the sizes of outer- and sub-blocks and Swift/K will determine how to invoke them.

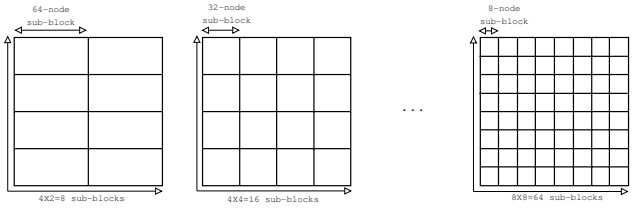

Figure 3. A range of sub-block configurations over a 512-node outer block

Figure 4 shows the sub-blocks system mapping between the Swift/K call-chain and the architectural components of BG/Q. As shown, *swift_invoke* and *Coaster_service* run over the user front-end. The *worker*, *swift_wrapper*, and *bg_sh* run on service nodes under the Cobalt scheduler. The Cobalt `runjob` command runs on compute nodes, which in turn invoke the application executable on sub-blocks of the chosen shape and size. In this example, 64 8-node sub-jobs are shown running over an outer block on 512 nodes.

2.2. Main-wrap

A traditional approach to run applications in many-task parallel mode is to use the POSIX `fork()/exec()` system call mechanism to fork a new process for application invocation. However, this approach is not suitable on the BG/Q's Compute Node Kernel (CNK) Operating System, which prohibits `fork()/exec()`. In this situation, ordinary applications can be run in many-task mode by *encapsulating the main function of the application in an HPC wrapper* and executing the wrapper on the BG/Q nodes. This approach enables programs (in almost any compiled language) to be called as a function rather than executed via a POSIX `fork()/exec()` invocation. Replacing the wrapper with an HPC workflow system such as Swift/T permits applications to be invoked from Swift/T on the BG/Q. The resulting workflow is more efficient, even on systems that do support the `fork()/exec()` invocation mechanism, since it avoids process management overheads.

2.2.1. Design. Applications that are built with compiled languages start their execution from an entry point such as the `main()` function in C/C++. The design goal of the main-wrap technique is to enable invocation of the entry point of a codebase directly from other code without creating a separate process. This enables a MTC system like Swift/T to run many instances of the application over an HPC system such as BG/Q.
The design must take into account the way application code is built. Most applications are composed of one or more supporting libraries and built as either a standalone static executable or a dynamic executable that loads some libraries at runtime. This distinction has significant implications for how a Swift/T application is compiled and run. To build a standalone static executable, all libraries required by the application, their dependencies, system libraries, and the Swift/T language runtime must be linked into the executable. The design must also consider the command-line parameters of the application. When a C/C++ program is invoked, any command-line parameters are passed into the `main()` function with two arguments: argument count (`argc`) and vector (`argv[]`). main-wrap must enable command-line arguments to be passed into wrapped applications from Swift/T scripts through the same mechanism.

2.2.2. Implementation. Implementation of the main-wrap technique is done using a shell script that automates the various transformation steps as shown in Figure 5. In the first step (①), the definition of the `main(int argc, char** argv)` in the application's entry-point program is extracted and replaced (using Unix `sed`) with a non-main (e.g. `leaf_main(int argc, char** argv)`) function in a new copy of the program. This modified `main` function definition is also added to the corresponding header file(s). In the second step (②), the duplicated source file is compiled by linking it with the application libraries (assuming dynamic linking; we discuss a static build later). In the third step (③), a stub is generated in the Tcl language. Swift/T expects all extension functions (known as leaf functions) to be expressed though a Tcl interface. In the fourth step (④), an extension code is generated in C/C++ that contains appropriate Tcl interface so that it can be invoked from a Swift/T script. The extension also contains the code to capture the command line arguments passed to the user code, to pass them to the duplicated `main()` function. In the fifth step (⑤), the leaf function contained in the Tcl stub is defined and invoked as an application in a user level Swift/T script. This step is carried out manually since it depends on how the particular user/application invokes applications. The Swift/T script is then compiled (⑥) using an optimizing compiler [5] that generates Tcl code to be interpreted by the Swift/T runtime. The final compute node invocation (⑦) is made in a LRM (Local Resource Manager) wrapper (in this case a Cobalt scheduler script).
Finally, for the cases where an application is compiled and built as a standalone static executable, it is possible to create a self-contained Swift/T-wrapped executable. This is accomplished by specifying the libraries, sources, headers, and other scripts in a *manifest* file. This file is processed by a custom tool that takes the application entry point program as an argument and generates a C/C++ source code for

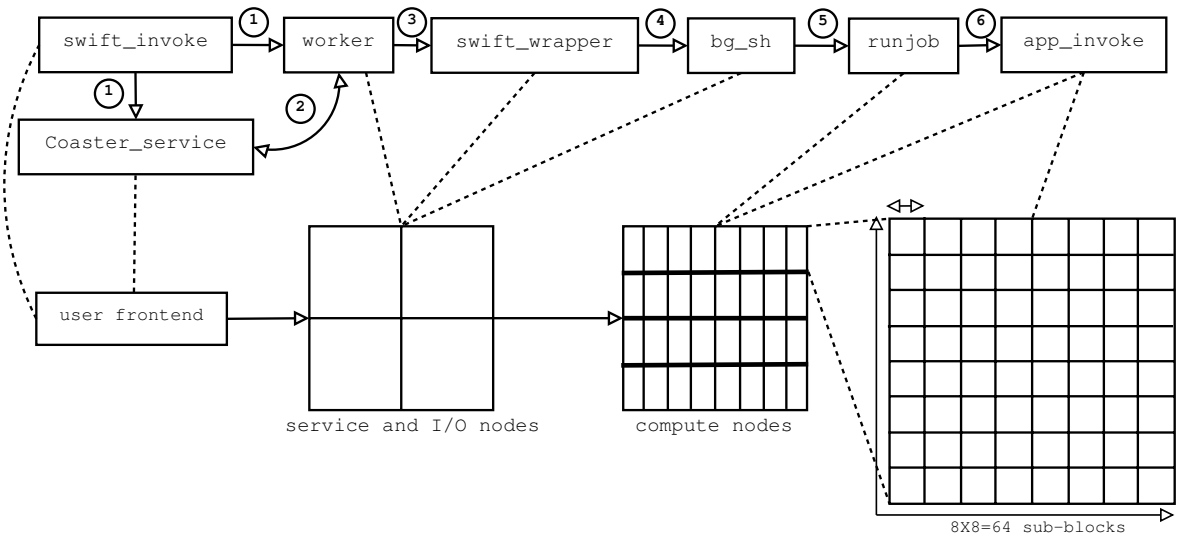

Figure 4. A mapping between the architectural components of BG/Q, the Swift/K call-chain and a sub-block configuration

standalone executable building. The generated program will initialize a Tcl interpreter with the required libraries. Any Tcl source code is embedded as character arrays. Once the C/C++ source code is generated, it can be compiled and linked with any required libraries into an executable using the platform's C/C++ compiler and linker.

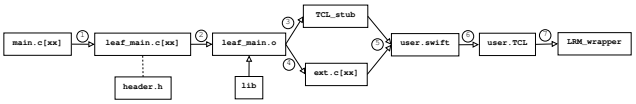

Figure 5. Swift main-wrap automation steps

3. Applications

In this work, we have examined four applications, the first two of which have been enabled by the sub-jobs technique, and the second two of which have been enabled by the main-wrap technique.

3.1. ematter (Material Science)

This application consists of a workflow that involves the invocation of `LAMMPS` and `Smeagol` tools. In our example, it is applied to TiN-Ta-HfO_2-TiN heterostructures to mimic HfO_2-based memristive switching devices [6]. It performs molecular dynamics (MD) calculations using `LAMMPS` to take into account the effect of temperature, and then performs density functional theory (DFT) - Non-Equilibrium Green's function (NEGF) calculations to calculate the electronic transport properties using `Smeagol`. `Smeagol` needs to read in the quenched atomic positions generated by `LAMMPS`, which we automate through a workflow. Figure 6 shows an overall workflow of the ematter application.

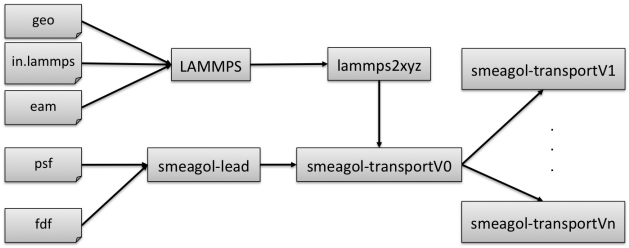

Figure 6. The overall ematter workflow.

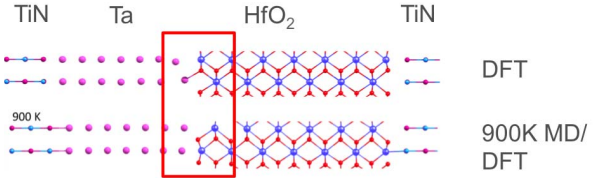

Figure 7. The HfO_2 heterostructures optimized by just DFT and MD/DFT.

The TiN-Ta-HfO_2-TiN heterostructure is modeled by 113 atoms, with a total length of about 7 nanometers. TiN is metallic and serves as an electrode on both sides. Tantalum (Ta), which is expected to work as the oxygen reservoir, is inserted between the left TiN electrode and HfO_2 (see Figure 7). The total energy of the optimized MD/DFT-derived structure is 1.2 eV lower that the DFT-only optimized structure, indicating the MD/DFT structure is more likely to exist in reality.

Interface structure is very important for transport calculations. Compared with the DFT-derived structure, the MD/DFT-derived structure exhibits a larger Ta-HfO_2 dis-

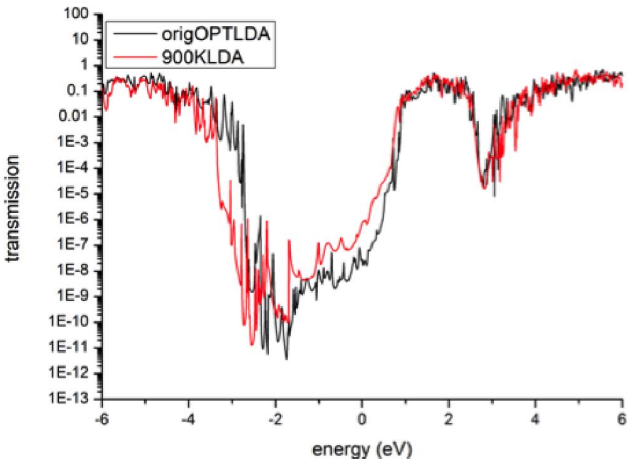

Figure 8. The transmission function of DFT (origOPTLDA) and MD-DFT (900KLDA) structures

tance and a smaller HfO_2-TiN distance. The structure difference results in different transmission functions (see Figure 8). While the transmission functions above the Fermi level (energy=0 in the figure) are very similar for both structures, the transmission function of the MD/DFT-derived structure is shifted by about 0.5 eV to the left.

3.2. VASP (Material Science)

VASP [7] is a popular materials modeling tool that can be used to perform atomic level calculations over first principal methods such as density functional theory (DFT). VASP-based workflows can be used to to study the behavior of materials by performing a series of calculations where one or more of a given set of basic parameters are perturbed. One such workflow is shown in Figure 10. The workflow starts from a basic input dataset over which a preprocessing step is run in order to produce a unique parameter set. VASP is invoked over this parameter set and selected results are collected and moved to a database for analysis and archival purposes. The Swift/K code for the workflow is shown in Figure 9. Lines 3-12 show the application function definition. Note the invocation to *bg_sh* script in lines 10-11. Lines 14-15 define the preprocess and postprocess scripts. Lines 17-20 define the basic VASP input set. The preprocessing script takes the lattice_factor (`fact`) and plugs it in one of the input files (`INCAR`). The `foreach` loop shown in line 22 generates 64 such independent lattice_factor values and invokes VASP in parallel over a parameter range of 1.0 to 6.4 with a step size of 0.1.

3.3. Rosetta (Protein Folding)

The `FlexPepDock` refinement protocol is part of the Rosetta protein modeling software suite. `FlexPepDock` is designed to create high-resolution models of complexes between flexible peptides and globular proteins. [8]

```
1  type file;
2
3  app (file _o, file _e, file _outcar, file _contcar)
4      runvasp(
5       file _vaspreproc, file _vaspostproc,
6       file _vasp_incar, file _vasp_poscar,
7       file _vasp_potcar, file _vasp_kpoints,
8       int _latt_factor) {
9
10   bgsh "vasp.bgq.ibm" @_vaspreproc @_vaspostproc
11   _latt_fact stdout=@_o stderr=@_e;
12 }
13
14 file vaspreproc <"vaspreproc.sh">;
15 file vaspostproc <"vaspostproc.sh">;
16
17 file incar <"INCAR">;
18 file poscar <"POSCAR">;
19 file potcar <"POTCAR">;
20 file kpoints <"KPOINTS">;
21
22 foreach fact in [1.0:6.4:0.1]{
23  file output <single_file_mapper;
24  file=strcat("logs/vasp-", fact, ".out.txt")>;
25  file error  <single_file_mapper;
26  file=strcat("logs/vasp-", fact, ".err.txt")>;
27  file outcar <single_file_mapper;
28  file=strcat("output/vasp-outcar-", fact)>;
29  file contcar <single_file_mapper;
30  file=strcat("output/vasp-contcar-", fact)>;
31  (output, error, outcar, contcar) =
32  runvasp(vaspreproc, vaspostproc,
33          incar, poscar,
34          potcar, kpoints, fact);
35 }
```

Figure 9. Swift/K representation of the user-level workflow for the `VASP` application.

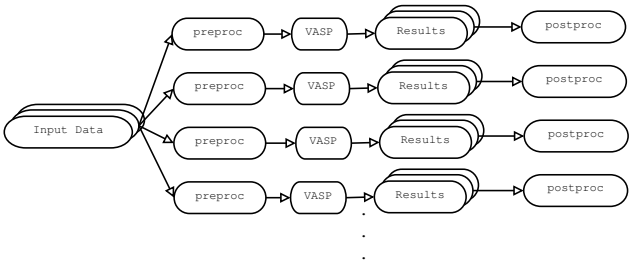

Figure 10. A workflow based on VASP calculations

`FlexPepDock` starts with coarse model of a peptide-protein complex. An initial step, called pre-packing, removes internal clashes between amino acid sidechains and the protein and peptide. The protocol then optimizes the position of the peptide using a Monte-Carlo minimization approach (see Figure 11), and calculates an energy score for the complex. The protocol is repeated n times, where n is defined by the user at runtime. The final results are presented as a ranked list of optimized models, where the lowest energy represents the most stable complex.

We used `FlexPepDock` to study a set of 2048 peptide-HLA (human leukocyte antigens) complexes. HLA are proteins located on the cell surface that present antigenic peptides to generate immune defense reaction. The peptides comprised of proteolyzed protein fragments that are degraded by cytosolic proteinases and are typically of

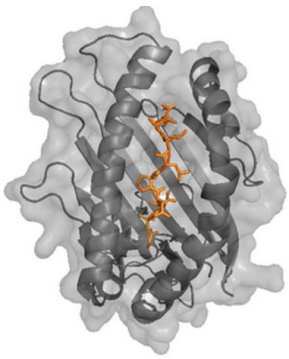

Figure 11. A nonameric peptide is modeled into the binding groove of an HLA molecule using the `FlexPepDock` refinement protocol.

length between 8-12 amino acids. Once bound to the HLA molecule, the peptide-HLA complex is presented for recognition to the T-cell receptors of CD8+ cytotoxic T-cells. The t-cells can identify peptides from foreign bodies and will initiate an immune response.

Our peptide data set contain representatives from the Immune Epitope Database (IEDB), a repository that collects and organizes data on major histocompatibility complex (MHC) binding experiments [9]. This set was previously studied by Binkowski et. al. to analyze the effects of amino acid substitutions in the HLA binding groove using a different methodology [10]. We applied the `FlexPepDock` refinement protocol to assess the viability of incorporating the more complex methodology into the reported analysis workflow.

3.4. DOCK (Protein Docking)

We study a mock version of the DOCK [11] protein folding application called *mockdock*. The mockdock application accepts two files as input (representing a protein file and a peptide file), and returns a number. The input files are read by mockdock, but their contents are not processed. The program prints a single integer on standard output, based on the length of the input files.

4. Experiments

We performed a number of experiments by applying the sub-jobs and main-wrap techniques to the four science applications.

The sub-job enablement of the ematter application had two qualitative impacts: first, rapid prototyping of workflow design and proof-of-concept execution, and second, production level execution of the workflow for a medium-size dataset. We ported and executed the workflow with 8 parallel 16-node sub-blocks over an outer block of 128 BG/Q nodes.

The sub-job technique is particularly well-suited to VASP-based workflows. As a prototype, we configured and executed the VASP workflow in sub-job mode over 512 nodes (8192 cores) of BG/Q with 64 parallel 8-node sub-blocks. VASP's current MPI-based implementation is known to be sensitive to memory usage and scalability with respect to the MPI-ranks and input parameters. Consequently, careful calibration is required in choosing the shape of a VASP job. A lower number of nodes for a particular run might offer insufficient memory whereas a larger number of nodes (and a proportional number of MPI ranks) might result in poor scalability. Providing a larger number of nodes with lower MPI ranks will waste compute resources. The easy tuning of the size of sub-block with the sub-job technique resulted in rapid calibration of the optimal job size for a given parameter set.

```
1  import io;
2  import files;
3  import string;
4
5  (int v) leaf_main(string A[]) "leaf_main" "0.0"
6                                "leaf_main_wrap";
7
8  main{
9   /* Collection of pdb files, */
10  file PDB[]=glob("/proj/ExM/hlac/*complex*.pdb");
11  int nstruct=3; // nstruct per job
12  int m=30; // number of jobs per pdb
13
14  foreach pdb in PDB{
15   foreach i in [1:m]{
16    leaf_main([
17     "-database",
18     "/home/vsachde/minirosetta_database",
19     "-pep_refine",
20     "-s", filename(pdb),
21     "-ex1",
22     "-ex2aro",
23     "-use_input_sc",
24     "-nstruct", fromint(nstruct),
25     "-overwrite",
26     "-suffix", "run"+fromint(i) ,
27     "-scorefile",
28     split(filename(pdb), "/")[4]+i+".sc"
29     ]);
30 }}}
```

Figure 12. Swift/T representation of the user-level workflow for the `FlexPepDock` application.

The main-wrap technique was applied to `FlexPepDock`, as shown in Figure 12. Lines 5-6 define the wrapped main function, which is invoked inside the `main` method of the Swift/T script. In this example, the `FlexPepDock` application was executed in a two-level nested foreach loop (lines 14 and 15) doing 30 refinements over each of 160 PDB molecules resulting in a parallel invocation of 4800 application instances. The code is general enough for running the application using any combination of command line arguments (lines 17-31) without rebuilding. The code also illustrates usage of Swift/T builtin functions such as `glob` (line 10) to populate array from filesystem, `filename` (line 21) to obtain the file name from a mapped variable and `fromint` (line 26) to convert integer to string, and `split` (line 31) to split a string over a delimiter. The main-wrap technique allowed the application to be ported to the BG/Q in MTC form with minimum performance overhead. It was run in production as described in Section 3.3. Additionally, this implementation acts as a proof of concept for the family of related application in the Rosetta Commons software.

We applied the main-wrap technique to mockdock, and then evaluated and benchmarked the performance results with large-scale runs. The benchmark study was done on Mira for up to 1,000,000 application invocations, with performance results as shown in Table 1. Each application task invocation ("Tasks" column) ran for approximately 30 seconds of real time. The number of parallel application tasks running at a time were set to be equal to the number of available cores, with one task per core. When the number of application tasks exceed the number of available cores, the additional tasks are started and assigned to cores as the previous tasks finish and the cores become available again. In this experiment, since the tasks are of same duration, we observe sets of tasks that nearly simultaneously start, run, and then finish. We refer to one of these sets as a "wave."

The number of load-balancing and task-distribution servers in each run ("Servers" column) was incremented as the number of tasks increased. We use a ratio of load-balancing servers to core of 1:∼500 for smaller runs and 1:∼300 for larger runs. We increase the ratio for larger runs in order to mitigate the communication bottlenecks between Turbine [12] (Swift/T's task dispatcher) and the load-balancing servers. However, we still see a slight lag in scaling, which we attribute to I/O overhead on the BG/Q. We note that a limited number of I/O nodes interact with a vast number of compute nodes (see Figure 1). For example, the 1 million 30-second tasks run in 18 waves over 62,656 cores in 556 seconds, whereas ideally they should complete in 540 (18×30) seconds.

Tasks	Nodes	Cores	Servers	run time (s)
256,000	1002	16,032	32	492
512,000	2004	32,064	64	501
768,000	3008	48,128	128	495
1,000,000	3923	62,756	256	556

TABLE 1. PERFORMANCE OF MOCKDOCK APPLICATION USING MAIN-WRAP ON THE ALCF BG/Q SUPERCOMPUTER MIRA.

5. Technique Comparison

The sub-jobs and main-wrap techniques have both similarities and differences, which lead to their differing suitability for different application types and situations, as shown in Table 2.

sub-jobs	main-wrap
Higher runtime overhead related to block management	Minimum runtime overhead
Minimum application debugging as runs from executable	Significant application debugging as rebuild required
Application source not needed	Application source needed
Works with any executable	Currently supported C/C++
Flexible in mapping resources to application	More rigid in resource mapping

TABLE 2. COMPARISON OF THE SUB-JOBS AND MAIN-WRAP TECHNIQUES

Sub-jobs are generally well-suited for rapid porting of applications intended to run at medium scales (hundreds to thousands of tasks) whereas main-wrap is suited to applications running at massive scales (millions of tasks). While either technique can be used for most applications, there are situations where one technique is preferable over other. One often occurring situation is the porting of legacy applications for which source is not available. Since the main-wrap technique compiles the application from source, sub-jobs must be used in this case. Sub-jobs can be good to use for irregular sized jobs because of the packing flexibility, but one can also use the sub-jobs technique to run an application that was main-wrapped.

6. Related Work

As of this writing no similar work that addresses interfacing a workflow system with BG/Q sub-jobs is known. However, some related technologies are described in the original IBM Redbook for BG/Q [13]. BG/Q offers sub-jobs at the core level limited to one core per node. We do not implement this feature in the current implementation. It also provides a paradigm related to MTC called Multiple Program Multiple Data (MPMD), which is implemented through special directives specified in a configuration file that is referred to in the Cobalt job definition. MPMD is limited: user must manually specify MPI ranks of the application with a restriction that all ranks in the same node must belong to a single application.

Work similar to the main-wrap technique was carried out by the Charm++ group [14]. In their AMPI design and implementation, multiple simulated MPI processes are run in the same memory space. The work is related but is not the same: AMPI interleaves execution of multiple programs running in the same memory space, while in main-wrap we try to run them one-after-another.

7. Summary

We discuss two techniques to run ordinary applications in MTC mode on the BG/Q supercomputers at the ALCF leadership class computing facility.

First, the sub-block jobs technique lets users submit multiple, independent, repeated jobs within a single larger resource block. Sub-block jobs is a mode of running jobs on BG/Q systems wherein one can allocate a larger "outer" block of compute nodes and repeatedly submit jobs of smaller sized sub-blocks to this outer block. Swift/K workflows coordinate Cobalt sub-block job submission: multiple MPI, OpenMP or serial jobs within one or more large Cobalt jobs. The current implementation provides tools, scripts and example use-cases to run Swift/K applications in sub-block mode over the ALCF resources. The benefit of this approach is that the user does not have to invoke the sub-block specific routines involving the details of the underlying node interconnect hardware. Additionally, with the same Swift/K script and configuration, the user gets the flexibility to run jobs in sub-block or non-sub-block mode depending on the scale and size of a run. The approach

transparently allows user to run jobs directly via Swift/K. Users can run multiple 'waves' of jobs asynchronously and in parallel without restarting the outer block. We showed sub-jobs used to enable two material science applications utilizing up to 5,192 cores.

Second, the main-wrap technique allow users to wrap and invoke regular C/C++ application codes into the high-performance Swift/T execution engine and lets them run these codes as tasks within a many-task application on BG/Q systems. It lets users run their ordinary applications in HPC mode. Our automated approach makes it easier and transparent for users to use the main-wrap technique. We showed main-wrap used to enable two molecular dynamics applications utilizing up to 63,012 cores, and our experiments shows good performance scalability for up to 1 million application invocations.

Acknowledgments

This work was supported in part by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. We thank Kevin N. Harms and Raymond M. Loy of ALCF for help with sub-block jobs and application installation. Work by Katz was supported by the National Science Foundation while working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] I. Raicu, I. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Proc. of Many-Task Comp. on Grids and Supercomputers, 2008*, 2008.

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

[3] J. Wozniak, T. Armstrong, M. Wilde, D. Katz, E. Lusk, and I. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 95–102.

[4] M. Hategan, J. Wozniak, and K. Maheshwari, "Coasters: Uniform resource provisioning and access for clouds and grids," in *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, ser. UCC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 114–121. [Online]. Available: http://dx.doi.org/10.1109/UCC.2011.25

[5] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster, "Compiler techniques for massively scalable implicit task parallelism," in *Proc. SC*, 2014.

[6] P. Zapol, D. Karpeyev, K. Maheshwari, X. Zhong, B. Narayanan, S. Sankaranarayanan, M. Wilde, O. Heinonen, and I. Rungger, "Coupled molecular-dynamics and first-principle transport calculations of metal/oxide/metal heterostructures," *Bulletin of the American Physical Society*, vol. 60, 2015.

[7] G. Kresse and J. Furthmüller, "Software VASP, Vienna (1999)," *Phys. Rev. B*, vol. 54, no. 11, p. 169, 1996.

[8] B. Raveh, N. London, and O. Schueler-Furman, "Sub-angstrom modeling of complexes between flexible peptides and globular proteins," *Proteins: Structure, Function, and Bioinformatics*, vol. 78, no. 9, pp. 2029–2040, 2010.

[9] B. Peters, H.-H. Bui, S. Frankild, M. Nielson, C. Lundegaard, E. Kostem, D. Basch, K. Lamberth, M. Harndahl, W. Fleri, S. Wilson, J. Sidney, O. Lund, S. Buus, and A. Sette, "A community resource benchmarking predictions of peptide binding to MHC-I molecules," *PLoS Computational Biology*, vol. 2, no. 6, p. e65, 2006.

[10] T. A. Binkowski, S. R. Marino, and A. Joachimiak, "Predicting HLA class I non-permissive amino acid residues substitutions," *PLoS ONE*, vol. 7, no. 8, p. e41710, 08 2012.

[11] P. T. Lang, S. R. Brozell, S. Mukherjee, E. F. Pettersen, E. C. Meng, V. Thomas, R. C. Rizzo, D. A. Case, T. L. James, and I. D. Kuntz, "DOCK 6: Combining techniques to model RNA–small molecule complexes," *Rna*, vol. 15, no. 6, pp. 1219–1230, 2009.

[12] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for high performance many-task applications," *Fundamenta Informaticae*, vol. 128, no. 3, pp. 337–366, 2013. [Online]. Available: http://dx.doi.org/10.3233/FI-2013-949

[13] G. Megan, *IBM System Blue Gene Solution Blue Gene/Q Application Development (IBM Redbooks)*. IBM Press, 2013.

[14] G. Zheng, S. Negara, C. Mendes, L. Kale, and E. Rodrigues, "Automatic handling of global variables for multi-threaded MPI programs," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, Dec 2011, pp. 220–227.

(The following paragraph will be removed from the final version)