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

Argonne National Laboratory
University of Chicago
Introduction

- Porting applications that are not specifically programmed for Blue Gene/Q (BG/Q) architecture/scale/size can be a significant challenge.
- This work addresses the challenge by implementing two techniques: sub-jobs and main-wrap that enables ordinary applications to run over BG/Q systems.
- The work has enabled real world applications to be run on BG/Q from a single workflow description which would have been tedious to run otherwise.
Overview

- Introduction
- Swift
- BG/Q Architecture
- The Sub-jobs Technique
  - Design
  - Implementation
- The Main-wrap Technique
  - Design
  - Implementation
- Compare and Contrast
- Applications
  - VASP
  - ematter
  - Rosetta
- Summary
Swift addresses most of these components
Goals of the Swift language

Swift was designed to handle many aspects of the computing campaign

- Ability to integrate many application components into a new workflow application
- Data structures for complex data organization
- Portability- separate site-specific configuration from application logic
- Logging, provenance, and plotting features

- Today, we will zoom in on a particular aspect of running scientific campaigns in HPC
Swift

Swift is a workflow language and runtime suited to both HTC and HPC platforms. Two versions of Swift exist: Swift/K and Swift/T. Both are ported to the Blue Gene/Q systems.

- Swift/K
  - Long running applications
  - Application with complex and/or legacy binaries
  - Workflows with time-varying resource demands

- Swift/T
  - Short running tasks
  - Workflows that fit well into a single allocation
  - Applications easily callable as libraries
When do you need workflow?
Typical application: protein-ligand docking for drug screening

\[ O(10) \] proteins implicated in a disease

\[ X \] O(100K) drug candidates

Tens of fruitful candidates for wetlab & APS

1M compute jobs

Work of M. Kubal, T.A.Binkowski, And B. Roux
Nested parallel prediction loops in Swift

1. Sweep()
2. {
3.     int nSim = 1000;
4.     int maxRounds = 3;
5.     Protein pSet[ ] = glob("*.protein");
6.     float startTemp[ ] = [ 100.0, 200.0 ];
7.     float delT[ ] = [ 1.0, 1.5, 2.0, 5.0, 10.0 ];
8.     foreach p, pn in pSet {
9.         foreach t in startTemp {
10.            foreach d in delT {
11.                ItFix(p, nSim, maxRounds, t, d);
12.            }
13.         }
14.     }
15. }
16. Sweep();

10 proteins x 1000 simulations x 3 rounds x 2 temps x 5 deltas = 300K tasks
Swift programming model: all progress driven by concurrent dataflow

```
(int r) myproc (int i, int j) {
    int f = F(i);
    int g = G(j);
    r = f + g;
}
```

- F() and G() implemented in native code or external programs
- F() and G() run in concurrently in different processes
- r is computed when they are both done
- This parallelism is *automatic*
- Works recursively throughout the program’s call graph
Swift programming model

- **Data types**
  ```swift
  int i = 4;
  int A[];
  string s = "hello world";
  ```

- **Mapped data types**
  ```swift
  file image<"snapshot.jpg">;
  ```

- **Structured data**
  ```swift
  image A[]<array_mapper...>;
  type protein {
    file pdb;
    file docking_pocket;
  }
  protein p<ext; exec=protein.map>;
  ```

- **Conventional expressions**
  ```swift
  if (x == 3) {
    y = x+2;
    s = @strcat("y: ", y);
  }
  ```

- **Parallel loops**
  ```swift
  foreach f, i in A {
    B[i] = convert(A[i]);
  }
  ```

- **Data flow**
  ```swift
  merge(analyze(B[0], B[1]),
       analyze(B[2], B[3]));
  ```

The Swift/K architecture

- Runs on the Karajan grid engine
  - Implements the Java CoG provider interfaces for compatibility with a wide range of schedulers: PBS, SGE, etc.
  - Includes a flexible pilot job service called Coasters
  - Automatically deploys worker agents to resources with respect to user task queues and available resources
  - Currently runs on clusters, grids, and HPC systems
  - Can move data along with task submission
  - Contains a “block” abstraction to manage allocations containing large numbers of CPUs, reducing impact on the system scheduler

Maheshwari et. al., swift-lang.org
BG/Q Architecture (simplified)

- Three main functional components: User frontend (login nodes), Service and I/O nodes, and Compute nodes.
- Compute nodes are connected to login nodes via service and I/O nodes.
- Promotes large blocks, running tightly parallel single applications, e.g., MPI-based.
Target infrastructure

- The IBM Blue Gene/Q *Mira* at Argonne National Laboratory has 49,152 nodes containing 786,432 CPU cores, totaling 10 petaflops.
- We desire to run scientific workflows containing many relatively small tasks (1 to 1024 nodes).
- The Swift/K implementation can generate ~1000 tasks/second, but filling the machine in this way would overload the scheduler (Cobalt).
- Solution: Expose and simplify a low-level, complex subjobs technique through Swift.
Cobalt, the BG/Q scheduler provides a mechanism by which a large block of compute nodes can be partitioned into smaller chunks of nodes for individual job submission.

The process of this partitioning is not trivial and incompatible with workflow style computation.

Swift/K sub-jobs technique hides this process and simplifies execution workflows transparently on BG/Q systems.
swift_invoke invokes the workflow at login node

worker runs an agent on service nodes

coaster_service manages workflow task pool

swift_wrapper manages application I/O

bg_sh manages sub-blocks

runjob invokes the app

The app runs on a set of compute nodes
Sub-jobs Implementation

- Swift sub-jobs are implemented via the bg_sh script.
- It performs the following functions:
  - Determines the system name and sets the environment
  - Determines the shape and size of the sub-block passed by the user as an environment variable
  - Invokes the cobalt provided scripts to create and obtain the coordinates of the sub-blocks
  - Invokes the user-supplied pre-processing scripts (if any)
  - Invokes the application executable with arguments
  - Invokes the user-supplied post-processing scripts (if any)
  - Collects exit status, cleans up local data and shuts down the sub-blocks
Swift/T: Enabling high-performance workflows

- Write site-independent scripts
- Automatic parallelization and data movement
- Run native code, script fragments as applications
- Rapidly subdivide large partitions for MPI jobs
- Move work to data locations

Swift/T control process

Swift/T worker

C  C++  Fortran

MPI

64K cores of Blue Waters
2 billion Python tasks
14 million Pythons/s

14N tasks/s

10,000,000
1,000,000
100,000

10

30 100 200 1000 10000 100000

processes

tasks/second
Example execution

- Code

\[ A[2] = f(\text{getenv}("N")) ; \]
\[ A[3] = g(A[2]) ; \]

- Servers: evaluate dataflow operations, queue tasks

  • Perform `getenv()`
  • Submit `f`

  • Subscribe to `A[2]`
  • Submit `g`

- Workers: execute tasks

  • Process `f`
  • Store `A[2]`

  • Process `g`
  • Store `A[3]`

Dataflow script produces work for work queue

- Including MPI libraries, GPU tasks, ...
- What if SWIG isn’t an option?
The main-wrap Technique

- The Compute Node Kernel (CNK) of BG/Q does not support the POSIX fork()/exec() system calls which is the basis of task-level parallelism in traditional Swift.
- To address this, Swift/T lets applications link to its HPC many-task backbone that can then coordinate between tasks via an MPI implementation of a dataflow engine.
- This is done by “wrapping” the “main” function of an application via the main-wrap technique and invoking it as a single Swift/T workflow.
main-wrap Design

- Leaf_main.c[xx] wraps main.c[xx] which is compiled to leaf_main.o
- ext.c links leaf_main.o and a TCL_stub invokes it.
- The stub is then defined as an app function in user.swift
- user.swift compiles into user.TCL and is invoked from inside a cobalt_wrapper submitted to BG/Q.
The main-wrap technique is implemented using a shell script that automates the steps described in its design. The source file is duplicated and parsed where the main function is copied as leaf_main. The new file is compiled with application linkage instructions and an object file is produced. An extension code is produced which serves as a C-TCL interface since Tcl is the intermediate Swift/T language. The function is now callable as an app function from within a Swift/T workflow. Finally a compiled workflow is invoked from a cobalt job wrapper (wrapper sets up environment and does qsub).
sub-jobs vs. mainwrap: Compare and Contrast

<table>
<thead>
<tr>
<th>Swift/K sub-jobs</th>
<th>Swift/T main-wrap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime overhead due to block management</td>
<td>Minimum overhead</td>
</tr>
<tr>
<td>Minimum application debugging</td>
<td>Some debugging may be required as builds from source</td>
</tr>
<tr>
<td>Application source not needed, can work with just the executable</td>
<td>Application sources needed</td>
</tr>
<tr>
<td>Flexible resource mapping</td>
<td>Resource mapping depends on application scale and size</td>
</tr>
<tr>
<td>Favored for disk based data exchange</td>
<td>Favored for memory based data exchange</td>
</tr>
</tbody>
</table>
Application enabled via subjobs: VASP

- VASP (Vienna Ab-initio Simulation Program) enabled in workflow mode using sub-jobs
- The workflow includes preprocess and postprocessing steps
- Currently running a modest 16 512-node tasks over 8000 cores for 12 hours on the Mira machine
Application enabled via main-wrap: Rosetta

- Main-wrap technique is used to enable the “FlexPepDock” refinement protocol from the Rosetta Protein folding suite.
- The application runs a two-level nested foreach loop doing 30 refinements (invocation to FlexPepDock) for 160 protein molecules resulting in a total of 4800 concurrent application invocations.
Summary

- One of the first efforts porting ordinary applications and workflows inside Blue Gene/Q supercomputers
- Enables a wide variety of applications, both HTC and HPC
- The techniques can work in combination if needed
- The techniques can work on top of existing parallel libraries such as MPI and OpenMP
- Supports large scale computation: mainwrap shown to run up to 1 million concurrent tasks for toy applications
Acknowledgements

- Swift is supported in part by NSF grants OCI-1148443 and PHY-636265, NIH DC08638, DOE Office of Science ASCR Division, and the UChicago SCI Program
- Some work by DSK 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 authors and do not necessarily reflect the views of the National Science Foundation.