ATLAS on the BlueGene/L – Preliminary Results

Keith Seymour   Haihang You   Jack Dongarra
Department of Computer Science
University of Tennessee, Knoxville
Knoxville, TN 37996

May 9, 2006

1 Introduction

The goal of this work is to use ATLAS to produce a tuned linear algebra library for the BlueGene/L, while taking advantage of the parallelism of the machine to speed up the search process.

The properties of the machine present a few difficulties for a straightforward port of ATLAS. The compute nodes themselves are somewhat limited in functionality. We can not use them to compile test cases and some system calls (such as `fork`) are not available. The front end and compute nodes use different operating systems and CPUs, so cross-compilation is required on the front end. Also, rather than running the test cases locally, ATLAS will need to submit the jobs to a batch queue system.

2 Preliminary Compilation Issues

For simplicity, we will describe ATLAS as having two kinds of code:

1. code to manage the parameter search, deciding what test cases to generate and run.

2. dynamically generated code for each test case.

The code in the first category always runs on the front end since it requires system calls not available on the compute nodes. It also needs to be able to perform compilations, which can not be done on the compute nodes either. The code in the second category must run on the compute nodes since those are the processors we are tuning for.

The first step in porting/parallelizing ATLAS for the BlueGene/L was to modify the ATLAS makefiles to use the cross-compiler only when compiling the test cases and associated code (timing drivers, etc). All other parts of the code (from the first category above) will be compiled with the normal front end compiler. This required making several changes to the build process to ensure the proper compiler is being used.

3 Compilation of Multiple Test Cases

The compilation technique described above works fine for compiling one test case at a time, but we wanted to exploit the parallelism of the machine by running many test cases

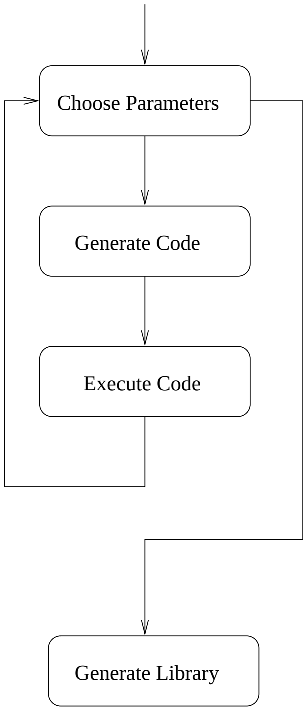

Figure 1: **Standard ATLAS Search**

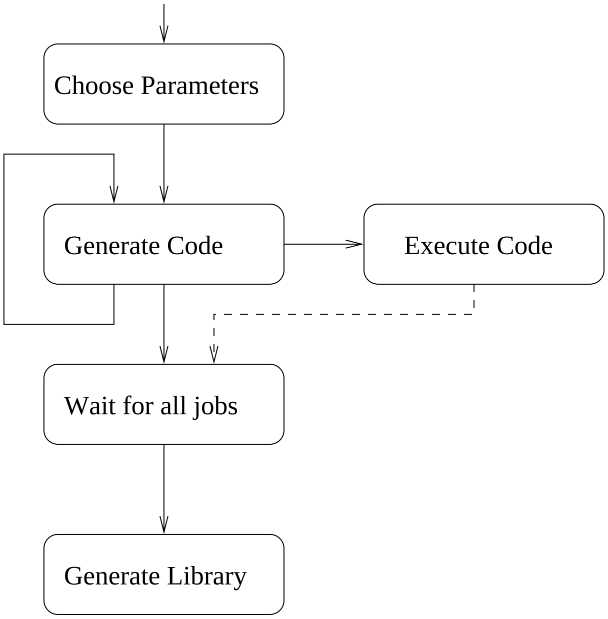

Figure 2: **Modified Search for BG/L**

simultaneously. This required a couple of changes to the way ATLAS generates the code. First, when multiple instances are generated, we need to be careful to avoid symbol name conflicts. This is handled by changing the names of external symbols to unique names. Second, to avoid having to submit each test case to the queue separately, we wrote a simple MPI driver that is linked together with all the test cases into one executable. Then we can submit to many processors at once and each processor determines which test cases to run based on its rank. After running the tests, each node sends its results back to the rank 0 node.

4 Running Jobs

In the single processor scenario, ATLAS uses `system()` to invoke `make` which builds and executes a single test case. Since we must replace the execution of the test case with a command to submit the job to the queue (`cqsub` or `llsubmit`), ATLAS needs to handle the non-blocking nature of the job submission. We have written code to keep track of the submitted jobs and determine when they have completed.

5 Search Techniques

ATLAS optimizes one parameter of the search space at a time. The overall structure of the ATLAS search is illustrated in Figure 1. However, optimizing only one parameter at a time would limit the amount of parallelism we could exploit, so we decided to optimize all parameters together. The first approach was to generate chunks of random parameter sets which are submitted as a parallel MPI job as described above, but having only one chunk running at a time. This approach was also not ideal since it did not make use of the time spent waiting for the chunk to finish executing. The current approach generates all the parameter sets in advance and overlaps the compilation of the test cases with the execution on the compute nodes. Since the compilation is generally faster than the test case execution, we can overlap the generation and submission of several chunks during the time it takes to evalute one chunk. This method is illustrated in Figure 2.

6 Results

Table 1 shows some preliminary results obtained on the 1024-node BlueGene/L at Argonne National Lab. The serial version generates a single test case and submits it to the batch queue, so only one processor is used at a time. However, this allows retaining the original search technique of ATLAS. When running in parallel, we submit to 64 nodes at a time with each node running 2 test cases, but the parameters for these test cases are generated randomly. "Parallel 1" refers to the first method, in which only one parallel job is running at a time. "Parallel 2" refers to the overlapping method, in which multiple parallel jobs are running simultaneously. Despite the fact that the parameters are generated randomly, we achieve nearly the same performance as the original ATLAS search.

Search Method	Number of Testcases	Number of Processors	Performance (Mflop/s)	Total Search Time (hours)
Serial (orthogonal)	607	1	880	13.5
Parallel 1 (random)	1111	64	860	3
Parallel 2 (random)	1024	64	825	1.5

Table 1: Preliminary Results – Argonne BlueGene/L

We also ran experiments on the 1024-node BlueGene/L at the San Diego Supercomputer Center using the "Parallel 2" method described above. The results are shown in Table 2. The total execution times are significantly longer in this case because of the long queue wait times.

Search Method	Number of Testcases	Number of Processors	Performance (Mflop/s)	Total Search Time (hours)
Parallel 2 (random)	1124	256	812	39
Parallel 2 (random)	4196	1024	812	67

Table 2: Preliminary Results – SDSC BlueGene/L