Performance of the Parallel Algorithms 1 and 2 on the Matrix Powers Kernel

George Hines, Javier Rosa, and Erin Summers

Motivation

The information flow in most real systems is well represented by sparse matrices; circuits, physical models, and routed communication networks to name very few. Solution of problems that arise in these domains generally make use of the Krylov subspace:

\[ \mathcal{K}_k(A;x) = \{x, Ax, A^2x, \ldots, A^kx\}. \]

Through intelligent parallelization, it is possible to compute the order-\(k\) Krylov subspace of a sparse matrix \(A\) in less time than is necessary to do \(k\) sparse matrix-vector multiplications by \(A\). The algorithms that realize this speedup are found in [2], and have been implemented by M. Mohiyuddin. As in [2], we will refer to the two main algorithms as "PA1" and "PA2," for "Parallel Approach 1 (or 2)."

Krylov subspace methods are well studied, and we needn't go into detail here. See [2] for details.

Intro to KSMs

The following brief descriptions of PA1 and PA2 are outlined in more detail in [2].

PA1 maximizes the overlap in computation and communication, but produces some redundant work by data near the processors boundaries. The elements of the Krylov subspace that can be computed without communication are calculated on each processor. At the same time, each processor sends the components of \(x\) that are needed by the neighboring processors to compute the remaining entries. The processors wait until the remote components of \(x\) arrive and finish their portion of the computation.

PA2 aims to minimize the redundant work done in PA1 by making each processor computing the least redundant set of values needed by the neighboring processors. At the same time, the processor broadcasts these values and computes the rest of the local computation for its particular section. After computing all of the local values, the processors wait for the remote entries to arrive and then finish the work. Though the work is minimized, the overlap between the computation and communication is slightly reduced. Additionally, the complexity of the PA2 algorithm is greater that that of the PA1.

Figures 1-3 pictorially exemplify the dependencies for PA1 and PA2 for computing the Krylov subspace for a tridiagonal matrix.
Figure 1: First phase, locally computable components

Figure 2: PA1
For stencil matrices, as the number of desired Krylov basis vectors increases, the gains from PA2 fall off and performance resembles PA1. This is potentially not true for other sparse matrices, whose dependency patterns are not as simple as that of stencil matrices (Figure 3).

**BUPC and UPC**

Unified Parallel C (UPC) is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The advantage of the UPC model is that it can be used with both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time. UPC is freely available. See [1] for details.

Existing UPC implementations include direct, massive compilation systems (GCC-UPC, Cray UPC) and source-to-source translation complemented with extensive runtime libraries (Berkeley UPC, HP UPC, and MuUPC).

Berkeley UPC (BUPC) includes asynchronous extensions that allow for overlap of communication and computation. Hence, one must use BUPC in order to achieve the optimality described in PA1 and PA2.

**Analysis Software**

When analyzing the performance of a program, there are two major techniques:
profiling and tracing. Profiling keeps track of statistics about the programs runtime performance and reveals at a high level where time is being spent in the program. Tracing keeps a log of all of the activities performed by the program in the trace file. Though these trace files can be long, the exact behavior of the program can be reconstructed through these files.

Throughout this project, we made use of several performance analysis tools [3], [4], [6], [7], [9]. Some ended up not being appropriate for our analysis, but we did spend a hefty amount of time utilizing the tools. We provide a tutorial at the end of this paper, which outlines the necessary steps to get the tools working on Franklin. We hope that future classes can benefit from our experience using these tools on Franklin.

In particular we used the profiling functionality in the Parallel Performance Workshop (PPW) [7] to compare the speedups of PA1 and PA2 outlined below. We also used PAPI [6] to measure the actual flop rates which was a required parameter for the theoretical performance model. Basic timers were used to gather the wall-clock time necessary to compute the speedups of the PA1 and PA2 algorithms, which we compare to the theoretical bounds on performance.

**PA1 vs. PA2**

During our study of the PA1 and PA2 algorithms, we were intrigued. Comparison of PA1 vs. PA2 was done by using the profiling functionality of PPW. The algorithms were compared by the inclusive time spent in the akx_pa2_gather routine under the "Total" column in PPW. Table 1 shows the results for a 9-point stencil operation on a 64x64 mesh.

<table>
<thead>
<tr>
<th>k \ nprocs</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0.82</td>
<td>0.73</td>
<td>0.63</td>
<td>0.7</td>
<td>0.56</td>
</tr>
<tr>
<td>8</td>
<td>1.08</td>
<td>1.02</td>
<td>0.77</td>
<td>0.68</td>
<td>0.58</td>
</tr>
<tr>
<td>16</td>
<td>1.39</td>
<td>1.67</td>
<td>1.45</td>
<td>0.87</td>
<td>0.67</td>
</tr>
</tbody>
</table>

**Table 1: PA2 speedup vs. PA1**

Clearly, PA2 is the better algorithm when the power is high relative to the number of processors. As number of threads increases, the algorithm may become communication bound making overlapping communication and computation a more important optimization. PA1 is clearly the faster of the two.

**Modeling**

In [2] a detailed model is constructed for the speedup obtained relative to \( k \) sparse matrix-vector multiplies by executing algorithm PA2 with overlapped communication and computation. This model is simulated using hypothetical parameters that are close to those published for NERSC's Franklin (Cray XT-4) machine with MPI benchmarks. The simulated matrix sizes are exceedingly large, and the results
reflect an optimization over the number of compute cores up to the maximum
number available.

We attempted to verify this model using the UPC implementation of the sparse
matrix powers kernel. Because of NERSC job quotas, we were not able to cover the
whole space simulated in [2]. The smallest matrices considered in their simulations
had $n = 1024$, and the sizes increased in powers of 2 until $n = 2^{22}$. The maximum
number of processors available in the simulation was $p_{\text{max}} = 8100$. We had up to
64 processors to work with. Denoting by $T(n,k)$ the time necessary to run the
sparse matrix powers kernel for $k$ powers of a matrix with dimension $n$, the speedup
is given by

$$S(n,k) = \frac{k \cdot T(n,1)}{T(n,k)}.$$ 

This expression assumes the number of processors as a parameter of $T(n,k)$. An
important consideration is that the model presented in [2] does not consider the
effects of the memory hierarchy. This is equivalent to the assumption that all local
data fits in cache. With effective flop rate measurements in PAPI we found that
computations with matrix dimension smaller than $n = 128$ fit in cache on one core.
When scaling up to multiple cores we need to use enough processors so that the
local computation fits in cache, i.e., is smaller than $n = 128$. For instance, for a
problem with $n = 256$ dense matrix approximations suggest we would need at least 4
processors (this of course depends on the sparsity pattern; for stencil matrices
doubling the problem size would actually only require multiplying the minimum
required number of processors $p$ by a constant greater than $\sqrt{2}$ but less than $p$.

The parameters of the model are

- $t_f$, the time (in seconds) per flop
- $\alpha$, the network latency (seconds)
- $\beta$, the network bandwidth (seconds).

The $t_f$ was measured by instrumenting the AKX PA2 functions with $PAPI\_flops$
calls. $\alpha$ and $\beta$ were measured by timing a large number of small message sends and
a large number of large message sends, respectively. The model parameters for our
jobs were measured to be:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_f$</td>
<td>$1.2366 \times 10^{-9}$</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$2.3627 \times 10^{-5}$</td>
</tr>
<tr>
<td>$\beta$</td>
<td>$6.15 \times 10^{-10}$</td>
</tr>
</tbody>
</table>

Table 2: Model Parameters

Theoretical vs. Actual Performance

Computational performance was measured using the $\text{gettimeofday}$ function in
$\text{<time.h>}$. Each node had a single active thread for the PAPI and timer based tests in
order to avoid encountering the different latency and bandwidth parameters for
communication amongst threads on a multi-core die. The following plots show speedups for 2, 4, 8, 16, 32, and 64 processors. These speedups do not agree with the speedups predicted by the model. It is likely that the measured model parameters are incorrect. While the $\beta$ parameter reasonably agreed with the theoretical bandwidth given by NERSC, $t_f$ and $\alpha$ did not. Given the measured parameters in Table 2 the theoretical and actual speedups
completely disagree. It is likely that measuring latency and floprates requires a more subtle approach or even larger sample sizes.

Agreement with the theoretical model does occur, using the published parameters, for a large number of processors and a mesh size of 128:

![Graphs showing speedup and flops for different mesh sizes and stencil sizes.](image)

Here are the same plots using measured parameters:
More comparison plots are available in the directory named "imagery" inside the archive included with this report.

This discrepancy likely has several confounding causes; it is expected that agreement would occur for a larger number of processors given correct parameters as the theoretical model is asymptotic; the SPMV code for PA1/2 is not cache optimized; finally the latency ($\alpha$) measurements are likely incorrect.

Cache effects are not modeled and may account for oddities in the speedup results. As the underlying SPMV-like computation of PA1/2 is not sequentially optimized (the original goal of this project) the working set for each processor may become small enough to fit in cache. This would cause an unmodeled speedup and change the actual flop rates. For the test runs with mesh dimension of 128 and larger processor counts, the working set size may have been reduced to the point of increasing the flop rate to meet the NERSC reported rates for Franklin. This would account for why the model seems to better fit the observed data. Unfortunately, this doesn't explain away the discrepancies for larger mesh sizes. Overall, the latency discrepancy was the most surprising. The latency was measured by timing several thousand one machine word UPC write operations. We still have yet to determine why these latency measurements do not match those reported by NERSC.

**Conclusions**

In our study, we found that PA1 and PA2 behaved as expected. For larger amounts of threads (i.e. situations where there is more communication), PA1 was faster, since it was designed to maximize communication and computation overlap. PA2 was slower in these situations because it is designed to minimize the redundant work done in PA1, and if the power of the matrix is low, then there is not a lot of redundant work to be minimized and the additional complexity of PA2 becomes an issue. When the number of processors is lower and the power is higher, there is a larger redundancy that PA2 avoids. When it comes to achieving actual speedups commensurate with modeled speedups, obtaining accurate model parameters is crucial and not as simple as initially it would seem. One must also be sure to avoid cache effects when testing against a model with latencies, bandwidth, and flop rates for a single level memory hierarchy.
Future Work

There has been significant interest expressed in transitioning the matrix powers kernel from a research library to a usable library. At the moment the data structures and partitioning algorithms in the code are particular to 3-, 9-, or 27-point stencil matrices. Therefore in addition to more thorough documentation, the code needs to be outfitted with a general partitioner, possibly an interface to the ParMETIS library. On top of the partitioner, it will become advantageous to consider more general data structures. For instance, it is clear how to partition a stencil matrix for sparse matrix operations, but as was recently suggested, it is of interest to consider how one might perform sparse matrix operations on an adaptive mesh. The existing PAPI instrumentation and the PPWizard configurations should be more deeply integrated into the codebase to further assist developers in their attempts at optimization. Additionally, once accurate model parameters are found, comparing actual speedups to predicted speedups will offer insights into what optimizations are necessary.
Analysis Software Tutorial

Throughout this project, we implemented several performance analysis tools on Franklin. Implementation of these tools often took much more time than expected. In this section we provide quick tutorials on how to use the performance tools on Franklin. We hope that future students on this course will benefit from our experience in using these tools.

First of all, we highly suggest that anyone interested in using parallel performance analysis tools read [5], which provides an overview of the functionality of many of the tools available. Secondly, it noted that using any of the tools will increase the runtime, so proceed wisely.

UPCTrace [9] is included with the BUPC module on Franklin and provides a UPC/GASNet trace summary.

1. Load BUPC module
   >> module load bupc
2. Compiling: In Makefile or when compiling, add "-g -trace" after the upcc command.
   upcc -g -trace ./hello.upc,
3. Running: Add "-trace" after the upcrun call in your jobfile. After you run your job, there will be a upc_trace-* data file for each thread that ran.
   upcrun -trace -n 2 -N 2 ./hello.upc
4. Analyzing Data: To combine the trace files, run the following command:
   >> upc_trace -t upc_trace-*
   Note: This may take some time for large programs
5. Making sense of the output: The analysis output will contain PUT, GET, and BARRIER reports. Puts and gets are each reported based on the source line that performed the access with a count and message size statistics. The "GLOBAL" and "LOCAL" flags indicated if access was performed locally using shared memory or using network communication. The BARRIER report lists each barrier executed by the program run, grouped by source line number with a count and timing statistics. Each barrier operation has a NOIFYWAIT and WAIT entry. NOTIFYWAIT indicates the time interval between the upc_notify and corresponding upc_wait operation for the barrier. WAIT indicates the time interval spent blocking at the upc_wait operation awaiting barrier completion. High WAIT times generally indicate load imbalance, which could possibly be resolved by separating the upc_notify and upc_wait operations to increase the NOTIFYWAIT time and thereby overlap some of the barrier time with useful computation.

GASP (A Standardized Performance Analysis Tool Interface for Global Address Space Programming Models)

The Global Address Space Performance (GASP) tool interface [3] supports instrumentation of any GAS programming model implementation, while simultaneously allowing existing performance analysis tools to leverage their tool’s infrastructure and quickly add support for programming languages and libraries using the GAS model. If the programmer is not interested in developing their own GASP-enabled tool, or using the low-level GASP functions, then we suggest using the turnkey Parallel Performance Wizard (PPW), which is detailed below. With the PPW
profiling communication in parallel GAS languages becomes almost as simple as using serial profilers.

Getting started with GASP:
1. Download code from "gasp-dump" section http://gasp.hcs.ufl.edu/implementors.html
2. In Makefile set:
   UPCC=upcc -inst-toolname=gasp-test
3. To run gasp-test, create a jobfile for Franklin and run the program like so:
   upcrun -n 4 -N 2 gasp_test

**Parallel Performance Workshop (PPW)** [7] is a performance analysis tool designed for partitioned global-address-space (PGAS) programs and features an easy-to-use interface and tight integration with PGAS programming models via the GASP interface. PPW is included in the bupc environment module on the NERSC machines. In our particular project we used the profiling run, communication statistics, and trace tools.

An excellent tutorial can be found here: http://ppw.hcs.ufl.edu/docs/html/tutorial.html. One can either install PPW or use the web interface http://ppw.hcs.ufl.edu/jws/ ppw.jnlp.

Basic:
1. When compiling instead of "upcc", use "ppwupcc".
2. In the job file, add "ppwrun --output=FILENAME.par" before the upcrun command. After submitting and running successfully, a .par file will be generated.
   Ex: ppwrun --output=FILENAME.par upcrun -n 4 -N 2 ./hello.upc
3. Copy the .par file onto your local machine and evaluate using PPW.

Note: The following options (profiling, communication, trace) require a lot of overhead on their own and should not be combined. It is advised to do one kind of analysis at a time.

Profiling: If you are curious to know more about the performance of your particular application, you might want to perform an initial profiling run on a few nodes. PPW will provide a breakdown of how much time is spent executing different functions. To profile your application, type "--inst-functions" after "ppwupcc" in your makefile.
   Ex: UPCC=ppwupcc --inst-functions

Communication: To view more details about communication amongst threads, you can use either "--comm-stat" or the more detailed line by line "--line-comm-stat" after the ppwrun command in your jobfile. This will allow you to view the "Data Transfers" tab in PPW.
   Ex: ppwrun --comm-stat --output=FILENAME.par upcrun -n 4 -N 2 ./hello.upc

**Trace using Jumpshot**
Jumpshot [4] is a graphical trace viewer, can be used very easily with PPW. To run Jumpshot, you can either download the software or run from the web interface (http://www.jxxi.com/webstart/app/jumpshot-trace-viewer.html).

1. Insert "--trace" after the ppwrun command in your jobfile
   Ex: ppwrun --trace --output=FILENAME.par upcrun -n 4 -N 2 ./hello.upc
2. Open the .par file in PPW
3. Export -> SLOG-2 (Jumpshot) Export the PPW data to the appropriate Jumpshot file type
4. Open .slog2 file in Jumpshot

**PAPI** (Performance Application Programming Interface)

PAPI [6] provides an interface and common abstractions of the performance counter hardware. Although, one should be wary of how PAPI is implemented on your particular system as the hardware counters available varies between systems: PAPI may have to infer metrics on one system whereas on another a direct counter is available. Regardless, PAPI enables software engineers to see, in near real time, the relation between software performance and processor events.

1. Load papi module: module load xt-papi
2. Compile: Add these flags to your compile command
   "PAPIARGS = -I/opt/xt-tools/papi/3.7.2/v23/include -L/opt/xt-tools/papi/3.7.2/v23/lib -lpapi &nbsp; -lpfm"
   &nbsp;Ex: At the end of your "$(UPCC)" line in the Makefile, add "$(PAPIARGS)"

3. PAPI doesn't run automatically like PPW or UPCTrace. You must instrument your code with the appropriate PAPI functions. Perhaps the simplest function is "PAPI_flops", which updates a flop counter, a wall-clock timer, a processor timer, and an instruction count each time it is called.

**References**

   &lt;http://upc.lbl.gov/&gt;.


   &lt;http://gasp.hcs.ufl.edu/&gt;.


