# CSC 435
# Project 1 – Your Very Own Baby BLAS
**FINAL FORM Due 6:00 p.m., April 26, 2010**

In this project you will write your own highly optimized BLAS (Basic Linear Algebra Subroutine) for a few fundamental operations in linear algebra. There is a theme in this project that will be repeated in your other projects – namely that it will be done in five separate phases. Each phase will have code and a small research style paper as a deliverable. The final paper will require that all of the "pieces" be combined into one final substantial research paper. The phases for this project are:

1. **Single Processor Optimization:** In this phase you will work to make each routine in your "Baby Blas" run as fast as possible on a single processor. In this phase your focus will be to develop code and use compiler optimization options to achieve the fastest CPU execution time. Code timings will need to be done on *zeus* and the machines in the cluster – and you will most likely need to use slightly different compiler options to get the fastest times on the different architectures. Where applicable, you should include references to the primary literature for the optimization techniques. You should also produce graphs of floating point operations per second vs. the dimension of your linear system. You should use the optimized ATLAS libraries on *zeus* and the cluster machines as your basis for comparison.

2. **Shared Memory Multiprocessing via OpenMP:** In this phase you will use the OpenMP library to reduce the wallclock runtime of your code by inserting OpenMP directives in your code. In addition to the items above, you should produce plots of floating point operations per second as a function of threads.

3. **Shared Memory Multiprocessing via pThreads:** In this phase you will use the *pthreads* library to reduce the wallclock runtime of your code by explicitly writing threaded code segments. As with the *OpenMP* phase, you should produce plots of floating point operations per second as a function of threads.

4. **Distributed Memory Parallel Processing:** In this phase you will move to parallel processing across computers. This is a BIG switch from the earlier phases and is much more demanding – but the payoffs can be huge. The focus in this phase is to use a single processor on each machine. Instead of plotting floating point operations per second vs. threads, you will plot floating point operations per second vs. systems.

5. **Mixed Memory Model Parallel Processing:** In this phase you will move to parallel processing across computers. The added feature is that you will also run threaded symmetric code on each system. Theoretically this should give you the maximum performance. But be warned – getting this performance comes with a price – your time in code development, debugging, tuning, and testing. In this case you will be looking for floating point operations per second per logical CPU core.

Now your code must be developed in C/C++ and built into a library that can be called from either C/C++ or Fortran. When you are done, you will have five separate libraries. Each library will contain the following routines:

- inner (dot) products of two vectors

- tensor product of two vectors (column vector times a row vector)
- matrix/vector multiplication (rotates the vector)
- matrix/matrix multiplication (square matrices only)
- matrix balance (square matrix only)
- direct linear solver
- iterative linear solver

These double precision routines will be called from Fortran – and it is up to you to determine how to call these most effectively to minimize execution time. I will tell you that the calls on the FORTRAN side will look like. In the case of the single processor codes, you will name the library *bblas.a* and it's routines will be called like:

`dot(N, vec1, vec2)` — dot is a double precision function

```
call tprod(N, vec1, vec2, rmat)
call vvm(N, vec1, vec2, rmat)
call mvv(N, vec, mat, vresults)
call mmm(N, mat1, mat2, rmat)
call mbal(N, mat)
call dls(N, mat, vec, rvec)
call ils(N, mat, vec, rvec)
```

In the case of the *OpenMP* optimized codes, you will name the library *mp_bblas.a* and it's routines will be called like:

`dot(num_threads, N, vec1, vec2)` — dot is a double precision function

```
call tprod(num_threads, N, vec1, vec2, rmat)
call vvm(num_threads, N, vec1, vec2, rmat)
call mvv(num_threads, N, vec, mat, vresults)
call mmm(num_threads, N, mat1, mat2, rmat)
call mbal(num_threads, N, mat)
call dls(num_threads, N, mat, vec, rvec)
call ils(num_threads, N, mat, vec, rvec)
```

In the case of the *pthreads* optimized codes, you will name the library *pthreads_bblas.a* and it's routines will be called like:

`dot(num_threads, N, vec1, vec2)` — dot is a double precision function

```
call tprod(num_threads, N, vec1, vec2, rmat)
call vvm(num_threads, N, vec1, vec2, rmat)
call mvv(num_threads, N, vec, mat, vresults)
```

```
call mmm(num_threads, N, mat1, mat2, rmat)
call mbal(num_threads, N, mat)
call dls(num_threads, N, mat, vec, rvec)
call ils(num_threads, N, mat, vec, rvec)
```

In the case of the *distributed parallel* using only one processor per machine optimized codes, you will name the library *mpi_bblas.a* and it's routines will be called like:

`dot(num_procs, N, vec1, vec2)` — dot is a double precision function

```
call tprod(num_procs, N, vec1, vec2, rmat)
call vvm(num_procs, N, vec1, vec2, rmat)
call mvv(num_procs, N, vec, mat, vresults)
call mmm(num_procs, N, mat1, mat2, rmat)
call mbal(num_procs, N, mat)
call dls(num_procs, N, mat, vec, rvec)
call ils(num_procs, N, mat, vec, rvec)
```

Finally, in the case of the *parallel model* using message passing and shared memory parallelism, you will name the library *pbblas.a* and it's routines will be called like:

`dot(num_procs, threads_per_processor, N, vec1, vec2)` — dot is a double precision function

```
call tprod(num_procs, threads_per_processor, N, vec1, vec2, rmat)
call vvm(num_procs, threads_per_processor, N, vec1, vec2, rmat)
call mvv(num_procs, threads_per_processor, N, vec, mat, vresults)
call mmm(num_procs, threads_per_processor, N, mat1, mat2, rmat)
call mbal(num_procs, threads_per_processor, N, mat)
call dls(num_procs, threads_per_processor, N, mat, vec, rvec)
call ils(num_procs, threads_per_processor, N, mat, vec, rvec)
```

In addition, the code should be able to be compiled and run on *zeus* and the cluster machines. I am also expecting each of you to utilize a makefile for both the compilation of the code as well as creating the library. You will need to build and maintain a directory structure so that a single makefile can be used to build all of the libraries. I will provide more information on this later.

When done, each of you should tar the directory(ies) containing your source code, and makefile. The source code should be well commented so I will know what improvements you made and you should also annotate your makefile so I will know what compilation flags you think work best. I will "make" your library, link it in with my own code, and run speed trials.

**FINAL PHASE**

In the final phase of the project you are going to build ONE unified library containing all of your routines. While C++ will allow overloading methods, C and Fortran will not. To make your library as generally applicable as possible, you will need to use a prefix for each routine type. Remember, this only applies to your unified library. Single processor routine names will be unchanged. Those

utilizing OpenMP will start with a prefix `omp` (e.g – `ompdot`). Those that use pthreads will have the prefix `pth`. Single core parallel distributed parallel code will use the prefix of `mpi` and the code using multicore distributed code will simply use the prefix `par`, (e.g. – `pardot`). When you are done you will have the ability to write a single program that can can effectively test ALL of your routines.

You will need to create a makefile entry that copies, renames, compiles, and builds your unified library. You unified library will be called ubblas.a. Obviously we will get a bit creative with the script files within the makefile!

We will have a day toward the end of the semester in which we will each demonstrate our unified library. In preparation for that day the following deliverables are due on the following dates.

## DELIVERABLES AND DUE DATES

**February 15, 2010:** Single processor library. Save the entire directory, including the makefile, in a tar file. You will, at a minimum, need to prepare and turn in plots of flops vs. matrix dimension for the operations of dot product and matrix multiplication. You should also make flops vs. matrix dimension plots for the direct and iterative linear solvers. You should also write a brief paper describing what methods you used to optimize the code and what optimization flags need to be included in the makefile for maximum performance. You should also attempt to explain your performance results and provide any references from the literature that support your claims. To make this easier, you may want to make comparison plots between your initial code, your final code, and the optimized ATLAS library.

**March 15, 2010:** SMP parallel library. Save the entire directory, including all of the items from the previously built single processor library, and the makefile, in a tar file. You will, at a minimum, need to prepare and turn in plots of flops vs. matrix dimension for the operations of dot product and matrix multiplication – but this time you will need to also include the number of threads utilized. You should prepare plots for both your *OpenMP* optimized code as well as your *pThreads* optimized code. Once you start demonstrating performance across multiple threads, you should fit your performance data to Amdahl's law.[1] You should also make performance plots for the direct and iterative linear solvers. Write a brief paper describing what methods you used to optimize the code and what optimization flags need to be included in the makefile for maximum performance. You should also attempt to explain your performance results and provide any references from the literature that support your claims. If you see any roll-off behavior in your performance curve, attempt to explain this. To make this easier, you may want to make comparison plots between your initial code, your final code, and the optimized threaded ATLAS libraries. Look for examples of these types of plots in the literature and include examples in your paper if applicable to your particular problem. Since in this phase you looked at both

**April 5, 2010:** Message Passing Code. In this part of the project you will repeat everything you did in the SMP study, including sending me a tar file of all the code you have built to this point. Since you will be now writing code for distributed processors, you will need to prepare Amdahl's law plots for speedup and efficiency as a function of the systems used. Again, you will need to write a brief paper to explain your code modifications to improve code performance, results, and any hindrances to parallel speedup.

[1] Look on page 44 of your *Fundamentals of Parallel Processing* text

**April 19, 2010:** Full Parallel Code. Repeat everything you did in the message passing step, but now your parallel speedup will be based on the number of cores utilized. In this stage of parallelization there is some disagreement on the manner to get the maximum speedup and scalability. Some think that the the code should run only as an MPI code with each core on an SMP machine handling a different MPI process. Others think that each machine should have a single MPI process with multiple threads. Try to answer which method is best for your particular library.

**April 26, 2010:** FINAL PHASE. You will now turn in your unified library and include ALL of the code from your previous exercises in a tar file. You should provide a makefile that builds everything. You will also need to write a documented API for your unified library. A unified paper will need to be turned in, which incorporates all of your prior small papers, but now it will include a section on the unified library and also an introduction and conclusion. During the final exam period we will run the codes to decide who has the fastest library.

This is a project where your individual creativity can really shine. Do not feel like you only have to create the methods I have described above – there are many functions that you can include that could significantly enhance your codes performance. This is especially the case with the linear and iterative solvers! Have fun with this – but write your own code. Plagiarizing from the BLAS/Atlas libraries will not be tolerated and will wind you up in front of the honor council.