SCA Laboratory
Lab2: Parallelizing a Backward Substitution Code

C. Álvarez

Fall 2024

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Contents

Note: This laboratory is optional and its main purpose is to give you a deeper understanding of the algorithms explained in theory while abridging the work that you will need to develop for the numerical assignment.
If you decide to fill a report, deliver it in PDF at the “Racó”. There are some **questions** on the document that will guide you to elaborate it.

Part 1

OpenMP parallelization of a Backward Substitution Algorithm

The objective of this laboratory is to understand several different ways of accelerating the execution of a simple linear algebra algorithm, specifically a Backward Substitution one. As it has been explained in theory classes, the execution pattern is the same for Forward and Backward substitution algorithms as for several other triangular problems. You should refer to the "Direct methods" set of slides for more information.

In this laboratory you will use the same development environment as in the first laboratory. For connection and execution details please refer to Laboratory 1 documentation. Also, you can find in the first laboratory data some files that you may find useful to complete this laboratory (like `Paraver` configuration files).

All new files necessary to do this laboratory assignment are available in `/scratch/nas/1/sca0/sessions`. Copy `lab2.tar.gz` from that location to your home directory in `boada.ac.upc.edu` and uncompress it with this command line: `"tar -zxvf lab2.tar.gz"`. Also, remember to process the `environment.bash` file with this other command line `"source environment.bash"` in order to set all necessary environment variables[1].

1.1 Sequential Backward Substitution Algorithm

Inside the laboratory files you will find a sequential backward substitution algorithm already programmed in file `BackSubs.c`. Look at it and try to understand the code. As you can see the actual algorithm is a small part of the `main` and the longer parts of the code are in fact the initialization of the matrices and the final check.

Question 1: *Execute the code and time it for different sizes (e.g. 6000, 12000 and 24000 elements). How does the execution time grow with the number of elements?*

1.2 OpenMP parallelization

Now you are going to parallelize the sequential code with `OpenMP`. The parallelization should be really straightforward.

- Copy the sequential code into a new `BackSubs_omp.c` file so you can use the provided Makefile (although you can modify it, it is not necessary to do so).
- Try to parallelize the sequential code using `OpenMP` pragmas. You can use the "Direct methods" set of slides as a guide to do it.

[1]**Note:** since you have to do this every time you login in the account or open a new console window, we recommend that you add this command line in a `.profile` file in your home directory, which is executed when a new session is initiated.

- Also you should create a `submit-omp.sh` code to submit it to the queues. You can use the `submit-omp.sh` script provided or the one from the previous laboratory.

Question 2: *Copy the parallelized code. It is only necessary for you to copy the lines that do the actual work (not the full program).*

Now that you have parallelized the code you should time it and obtain a trace with `Paraver`. Be careful with the size of the traces that you generate.

Question 3: *Obtain a trace of the execution of the parallelized code with a size of 10 elements.*

As you can see in the trace (if you have not changed anything more in the code apart from parallelizing it), the time is not correctly measured. The time starts running before the initialization of the code and only stops after the final check is done. In order to properly observe the execution time of the code (specially with small size as is the case in this laboratory) you should time only the code that does the actual work. In this case this refers only to the 6 lines of the backward substitution that are the parallel part of the code.

Question 4: *Adjust the timing correctly. To be sure, copy here the timing code and the code TIMED (remember that you can ask to be sure that you are doing it correctly). Then repeat* ***Question 1****.*

Question 5: *Create a figure that shows the weak scaling of your code. When doing weak scaling, remember that you have to scale the computing load (not the size) with the number of threads. The number of operations in this algorithm scales with the square of the size, so when you multiply the threads by four, you only need to double the size. Also don't start with too many elements per one thread, otherwise the problem will be too large. Try to start with 6000 elements per one thread or less to avoid memory problems.*[2]

As you can see, you are probably obtaining less improvement than expected. Increasing the number of threads doesn't imply obtaining better performance. To explain this effect you can obtain a trace of the execution and zoom in the parallel part until you can observe how the system opens and closes parallelism.

[2]In a real world you probably want to go for problems as large as possible but in the laboratory we should stay inside the computing limits of Boada.

Part 2

MPI parallel code

In the file `BackSubs_mpi.c` you have the code of a naive `MPI` implementation of the Backward Substitution algorithm. Open it, understand it and compile it. As you can see the code starts broadcasting the initial matrix and vector. The same algorithm as in the OpenMP implementation is used. First the matrix and the vector are splitted between all the processes and each process initial and final rows assigned are computed. Then, each process computes its part of the B vector and broadcasts it to the other processes that update its right hand-side part of the equation (over the same B vector). As with this approach process 0 has to compute the final chunk of vector B (that contains the final solution computed in-place), there is no need for a final communication of the data to process 0.

As a difference from the previous version of the code you can see that the sharing of the matrix and the vector are not inside the timed region. In this case it has to be done this way because although the data is originally in one process and has to be shared, in several cases the initial sharing can be avoided if the initial data is already spread among the nodes or can be read in parallel from the source. This depends on the particular problem.

Now, compile and execute the `MPI` code. Be careful to use small matrix sizes (like 12000) as the system has problems to allocate big matrices in several threads in the same node and you don't want to collapse Boada.

Note that the data sharing is not the most effective one (as, there is no need for process 0 to distribute the whole matrix A and vector B among all processes, each process only needs its share).

Question 6: *First, try to execute the code using MPI on the different cores inside the SAME node. How does the code performance compare against the OpenMP version?*

Question 7: *Now, time the code when executing on 1 to 3 nodes with 1 MPI process per node. What can you explain about the performance?*

2.1 Improving the MPI code

Now, you can try to improve the `MPI` code that you already have. As you can guess it can be improved in several ways.

- The easiest way (from my point of view) is to try to do a cyclic partitioning of the rows computed by each node. This way the last node would not be idle nearly all the time.
- Thinking about doing an OpenMP partitioning inside a node it would be an even better approach to do cyclic partitioning of blocks of rows. That means, if we have a problem with 12000 rows and 3 nodes, that the first node could tackle rows 0-999, 3000-3999, 6000-6999 and 9000-9999.
- In environments with a lot of nodes it is often wise to not send the B vector data to all the remaining nodes from the generating node but to pass the data from one node to the next so the communication is shared between the nodes. Due to the fact that in the lab we are going to use only three nodes this code improvement will yield no results here.

- Also, it is usually unwise to send messages with a single data as networks are usually better at providing good bandwidth than small latencies (this is what the lab code is doing). Grouping some data in the same message can really improve the communication/computation balance, although the correct ratio depends heavily on the system that you are working on.

This task is not so simple as it seems and if you don't know MPI it is going to take you a lot of time, so the next question is only to be done by those of you that want to obtain an extra point in this lab report.

Question 8: ***Only do it for an Extra Point at your own risk.*** *Improve the MPI code. Copy your MPI improved code. Explain the code improvements and time the code when executing on 1 to 3 nodes with 1 MPI process per node. Compare with the results of Question 7.*

2.2 Joining MPI and OpenMP

Now that you have an `MPI` code you can try to parallelize it inside each node. You can try two approaches:

1. The simplest approach is to execute the `MPI` code using more processes than nodes. The system will assign more processes to each node cyclically. As each node has more than one core you can obtain some improvements. Try at least 2 and 4 processes per node (with different number of nodes).

   Question 9: *Time the performance of this approach.*

2. Add `OpenMP` pragmas to the `MPI` code and compile it with the correct flags. Depending on the version that you have already elaborated in the previous section, maybe the best `MPI` version is not the one that could be better parallelized with `OpenMP`. Also be careful with the compiler flags, you may need to modify the `Makefile`.

   Question 10: *Time the result when using both `MPI` and `OpenMP` in 3 nodes and try to obtain the combination that gives the maximum speedup. Copy in the report the final code and its performance.*

Part 3

BLAS

Basic Linear Algebra Subprograms (BLAS) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, matrix multiplication or even triangular systems solving. Although they initially were one specific development, the BLAS routines have become the de facto standard low-level routines for linear algebra libraries. Furthermore, although the BLAS specification is general, BLAS implementations are often optimized for speed on a particular machine, so using them can bring substantial performance benefits as BLAS implementations take advantage of special floating point hardware such as vector registers or SIMD instructions.

BLAS originated as a Fortran library in 1979 and its interface was standardized by the BLAS Technical (BLAST) Forum, whose latest BLAS report can be found on the netlib website[1]. This Fortran library is known as the reference implementation and is not optimized for speed but is in the public domain. Also, the routines have bindings for both C and Fortran so they can be used from a significant amount of high level languages.

The boada cluster has two different implementations of BLAS already installed: ATLAS and MKL.

- Automatically Tuned Linear Algebra Software (ATLAS) attempts to make a BLAS implementation with higher performance. To accomplish this goal, ATLAS defines many BLAS operations in terms of some core routines and then tries to automatically tailor the core routines to have good performance. A search is performed to choose good block sizes. The block sizes may depend on the computer's cache size and architecture. Tests are also made to see if copying arrays and vectors improves performance. For example, it may be advantageous to copy arguments so that they are cache-line aligned so user-supplied routines can use SIMD instructions. The tests make the process of installing ATLAS slower than a "standard" installation but fortunately in boada it is already installed.

  While ATLAS performance often trails that of specialized libraries written for one specific hardware platform, it is often the first or even only optimized BLAS implementation available on new systems and is a large improvement over the generic BLAS available at Netlib. For this reason, ATLAS is sometimes used as a performance baseline for comparison with other products.

- Intel Math Kernel Library (Intel MKL) is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS and other functions as LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math. The routines in MKL are hand-optimized specifically for Intel processors so they are supposed to deliver the best possible performance in these processors. The installation usually also performs several tests to further adapt the code to the specific processor characteristics.

3.1 Using BLAS

One simple way to optimize numerical codes is to adapt them to use one of such already programmed libraries. Depending on the specificity of the algorithm, finding the adequate set of functions to implement

[1] "BLAS Technical Forum". netlib.org. Retrieved 2019-10-01

it can be either really simple or cumbersome. However, once the program is adapted a decent performance (in respect of what the hardware can deliver) could be expected.

Fortunately, for this laboratory a BLAS level 3 function called trsm could be used. In order to obtain information about the BLAS functions, both netlib and MKL webpages can be used as reliable sources although when in doubt you should consider that netlib is the original one. Also, take into account that you are using C and not Fortran.

Question 11: *Which are the levels of BLAS and what do they mean?*

Question 12: *What does the function trsm do?*

Using a BLAS function is as simple as invoking it with the correct parameters. In addition you should do, depending on the library:

- ATLAS:
  - include the header files clapack.h and cblas.h
  - use the -lblas -llapack -lm library flags in the compilation options
- MKL:
  - include the header files mkl_lapacke.h and mkl.h
  - use the -lmkl_core -lmkl_sequential -lmkl_rt -lm library flags in the compilation options
  - add the path to the MKL libraries (already in the Makefile)

Remember that C always stores matrices by rows and that a column vector is a matrix with leading dimension 1.

Question 13: *Implement the program using the trsm function of ATLAS or MKL (cblas_dtrsm). Which is the code that you have typed?*

Now you can compare the OpenMP code that you have made in the first part of this laboratory against the BLAS code you have now. Neither ATLAS nor MKL allow you to easily adjust the number of threads that you are using, they automatically select what they think is the best parallelization. Consequently, you should limit your comparison to the best OpenMP version you have obtained. This also illustrates one of the problems of using libraries, that is, you can not adjust the resource usage of your system across the different parts of your program that are executing in parallel at a given time.

Question 14: *Time the library approach and compare the results against the ones in Question 4. Comment your deductions.*

3.2 Joining everything

Now we could try to join the BLAS version of our code with the MPI version. However we are not going to do it because in fact, that means that you should create a new MPI version of your code that uses the BLAS functions (and not some arbitrary code that you have tailored). This is one of the main reasons why from some years ago, whenever it is possible scientific codes are programmed with block partitioned algorithms (something that we will study in the next laboratory session).