Gram–Schmidt Process in Different Parallel Platforms

(Control Flow versus Data Flow)

Genci Berati
Tirana University, Department of Mathematics
Tirane, Albania

***Abstract*—Important operations in numerical computing are vector orthogonalization. One of the well-known algorithms for vector orthogonalisation is Gram–Schmidt algorithm. This is a method for constructing a set of orthogonal vectors in an inner product space, most commonly the Euclidean space Rn. This process takes a finite, linearly independent set S = {b1, b2, ⋯, bk} vectors for k ⩽ n and generates an orthogonal set S1 = {o1, o2, ⋯, ok}. Like the most of the dense operations and big data processing problems, the Gram–Schmidt process steps can be performed by using parallel algorithms and can be implemented in parallel programming platforms. The parallelized algorithm is dependent to the platform used and needs to be adapted for the optimum performance for each parallel platform. The paper shows the algorithms and the implementation process of the Gram –Schmidt vector orthogonalosation in three different parallel platforms. The three platforms are: a) control flow shared memory hardware systems with OpenMP, b) control flow distributed memory hardware systems with MPI and c) dataflow architecture systems using Maxeler Data Flow Engines hardware. Using as single running example a parallel implementation of the computation of the Gram –Schmidt vector orthogonalosation, this paper describes how the fundamentals of parallel programming, are dealt in these platforms. The paper puts into evidence the Maxeler implementation of the Gram–Schmidt algorithms compare to the traditional platforms. Paper treats the speedup and the overall performance of the three platforms versus sequential execution for 50-dimensional Euclidian space.**

Keywords*—Gram-Schmidt Algorithm; Parallel programming model; OpenMP; MPI; Control Flow architecture

I. INTRODUCTION

Classifications of parallel programming paradigms are mostly related to the hardware architectures.

The paradigms of parallel programming can be divided generally into two categories: process communicates [1] and problem decomposition [2].

Process communication is correlated to the instruments by which parallel processes communicate and share sources to each other. The most common forms of process interaction are shared memory and message passing between processes. Shared memory is an efficient instrument for passing data between programs by accessing that same shared memory. Algorithms may run on a single processor in sequential or on multiple separate processors in sequential way or in parallel. In shared memory model, parallel tasks share a global address space which they read and write to asynchronously. In shared memory systems the code can create threads each of them can access the same variable in parallel.

Message passing is a concept from computer science related mostly with distributed memory architectures for the parallel programming platforms that is used extensively in the design and implementation of modern software applications; it is very important for some models of concurrency and object-oriented programming. In a message passing model, parallel tasks exchange data and communicate through passing messages to one another. Either shared or distributed can be based Control Flow [3] Von Newman traditional architecture.

The paper deals with three different programming platforms (OpenMP, MPI and Maxeler). These three platforms can be grouped in two different architectures, in Control Flow (OpenMP and MPI) and Data Flow (Maxeler) architectures. These two different computing architectures are compared and analyzed in this paper by choosing a typical dense operations and big data problem which is the Gram – Schmidt process.

Is chosen Gram Schmidt classic algorithm for a 50-dimensional inner product space. The algorithm has operations rising in a significant progression from step to step. If we have a set S1={o1, o2, …, on} of orthogonal vectors as basis for the inner product space L, then we can express any vector of space L as a linear combination of the vectors in S1:

Let as have an arbitrary basis {b1, b2, … , bn} for an n-dimensional inner product space L. The Gram-Schmidt algorithm constructs an orthogonal basis {o1, o2, … , on} for L. In our paper we take the arbitrary basis {b1, b2, … , b50} for an 50-dimensional inner product pace L and after performing the Gram-Schmidt algorithm into a sequential machine platform, OpenMP platform, MPI platform and Maxeler controlfolw machine we than constructs an orthogonal basis {o1, o2, … , on} for L each time. The paper intends to compare the performance of the parallel platforms and to measure the speedup for each platform. The characteristics of the algorithms regards to the number of nested loops and the numbers of operations for iteration will define the best platform to recommend.

The reason why is selected the Gram – Schmidt algorithm is the time complexity. This algorithm complexity is $O(n^3)$. The operations in each iteration of the process rise progressively, so it is of large interest to study the behavior in different parallel programming platforms.

II. Gram–Schmidt Algorithm

To obtain an orthonormal basis for an inner product space L, we use the Gram-Schmidt algorithm to construct an orthogonal basis. For R^n with the Euklidean inner product (dot product), we of course already know of the orthonormal basis {(1, 0, 0, …, 0), (0, 1, 0, … , 0), … , (0, … , 0, 1)}. For more abstract spaces, however, the existence of an orthonormal basis is not obvious. The Gram-Schmidt algorithm is powerful in that it not only guarantees the existence of an orthonormal basis for any inner product space, but actually gives the way of construction of such a basis.

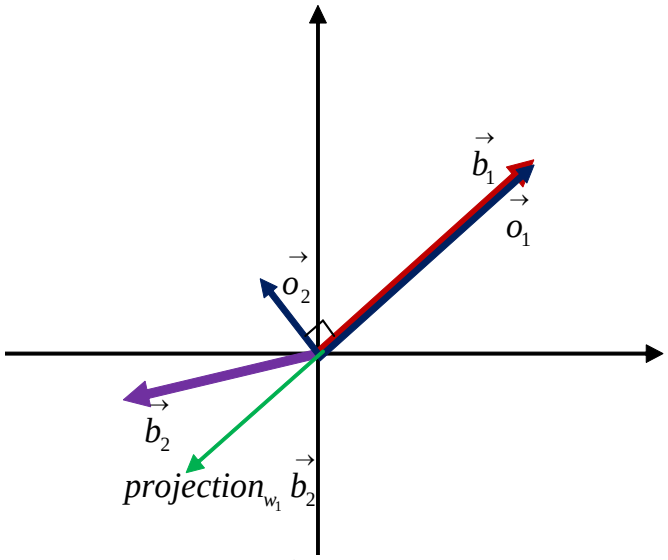

Fig. 1. Graphic representation of the Gram – Schmidt orthogonalisation

The Gram – Schmidt algorithm can be expressed in n steps to be performed. The algorithm steps are:

1 for i = 1 to n
2 vi = ai
3 for i = 1 to n
4 rii = ||vi||
5 qi = vi/rii
6 for j = i + 1 to n
7 rij = qi vj
8 vj = vj – rijqi

This algorithm is implemented in C++ code using Code::Blocks programming platform. This platform is chosen because it is portable to the parallel programming platforms.

III. Gram – Schmidt Vector Orthogonalisation Algorithm (Sequential Implementation)

We implemented the steps mentioned in the previous section in the Code::Blocks [1] with C++ compiler. In our implementation, we take k=n=50, where k is the number of the linear independent vectors and n is the dimension of the Euclidian space. The C++ program code of Gram – Schmidt algorithm for a 50 dimensional inner product space, in our example named space L, for k=50, looks like:

```
#include <cstdlib>
#include <iostream>
#include <math.h>
using namespace std;
double b[50][50];
double r[50][50], q[50][50];
int main(int argc, char *argv[]) {
int i, j;
for (int i=0; i<50; i++)
for (int j=0; j<50; j++)
{b [i][j]=rand() % 10;}
int k;
for (k=0; k<50; k++){
r[k][k]=0; // equivalent to sum = 0
for (i=0; i<50; i++)
r[k][k] = r[k][k] + b[i][k] * b[i][k]; //rkk = sqr(a0k) + sqr(a1k) + sqr(a2k)
r[k][k] = sqrt(r[k][k]);  //
cout << endl << "R"<<k<<k<<": " << r[k][k];
for (i=0; i<3; i++)
{q[i][k] = b[i][k]/r[k][k];
cout << "q" <<i<<k<<": "<<q[i][k] << ", ";}
for(j=k+1; j<50; j++)
{r[k][j]=0; for(i=0; i<50; i++) r[k][j] += q[i][k] * b[i][j];
cout << endl << "r"<< k <<j<<": " <<r[k][j] <<endl;
for (i=0; i<50; i++) b[i][j] = a[i][j] - r[k][j]*q[i][k];
for (i=0; i<50; i++) cout << "b"<<j<<": " << b[i][j]<< ", "; }}
system("PAUSE");
return EXIT_SUCCESS;}
```

Fig. 2. Sequential Gram – Schmidt vector orthogonalisation. Program code in C++ (Code::Blocks)

The average execution time of this sequential algorithm is around 110 seconds. Now let's see in the next session the parallel implementation of this algorithm in OpenMP [2] parallel platform for C++.

IV. The Gram–Schmidt Vector Orthogonalisation Algorithm for Openmp Platform

A parallel program is composed of parallel executing processes. A task-parallel model [4] focuses on processes , or threads of execution . These processes sometimes share the same sources, which emphasizes the need for communication between those processes. Task parallelism is a natural way to express message-passing communication between processing units. It is usually classified as MIMD/MPMD or MISD [5].

A parallel model consists of performing operations on a data set which usually regularly structures in an array. A set of tasks will operate on this data, but independently on separate partitions. In a shared memory system, the data will be accessible to all tasks, but in a distributed-memory system it will divide between memories.

Parallelism is usually classified as SIMD/SPMD (Single Instruction-Multiple Data)/(Single Program-Multiple Data) [6].

[1] Code::Blocks, "A free C, C++ and Fortran IDE". http://www.codeblocks.org/

[2] OpenMP Specifications. http://www.openmp.org/blog/specifications

The systems are categorized into two categories. [7] The systems of the first category were characterized by the isolation of the abstract design space seen by the programmer from the parallel, distributed implementation. In this, all processes are presented with equal access to some kind of shared memory space. In its loosest form, any process may attempt to access any item at any time. The second category considers machines in which the two levels are closer together and in particular, those in which the programmer's world includes explicit parallelism [8]. This category discards shared memory based cooperation in favor of some form of explicit message passing.

A classical shared memory parallel platform is OpenMP. OpenMP (Open Multiprocessing) is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran programming language. OpenMP [3] is an application program interface providing a multi-threaded programming model for shared memory parallelism; it uses directives to extend sequential languages It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms like a standard desktop computer or supercomputer. After the configuration of the Code:: Blocks for OpenMP, we implemented on this platform our algorithm. The code is parallelized for our nested loops as below:

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#include <omp.h>
using namespace std;
double b[50][50];
double r[50][50], q[50][50];
int main(int argc, char *argv[]) {
int i, j;
#pragma omp parallel for
for (int i=0; i<50; i++)
for (int j=0; j<50; j++)
{b [i][j]=rand() % 10;}
int k;
#pragma omp parallel for
for (k=0; k<50; k++){
r[k][k]=0; // equivalent to sum = 0
for (i=0; i<50; i++)
r[k][k] = r[k][k] + b[i][k] * b[i][k]; //rkk = sqr(a0k) + sqr(a1k) + sqr(a2k)
r[k][k] = sqrt(r[k][k]);  //
cout << endl << "R"<<k<<k<<": " << r[k][k];
#pragma omp parallel for
for (i=0; i<3; i++)
{q[i][k] = b[i][k]/r[k][k];
cout << "q" <<i<<k<<": "<<q[i][k] << ", ";}
for(j=k+1; j<50; j++)
{r[k][j]=0; for(i=0; i<50; i++) r[k][j] += q[i][k] * b[i][j];
cout << endl << "r"<< k <<j<<": " <<r[k][j] <<endl;
for (i=0; i<50; i++) b[i][j] = b[i][j] - r[k][j]*q[i][k];
for (i=0; i<50; i++) cout << "b"<<j<<": " << b[i][j]<< ", "; }}
system("PAUSE");
return EXIT_SUCCESS;}
```

Fig. 3. OpenMP Gram – Schmidt vector orthogonalisation. Program code in C++ (Code::Blocks)

For directive in the code above splits the for-loop, so each thread in the current team handles a different portion of the loop. The main directive used is "#pragma omp parallel for". This statement is used to open the switch of OpenMP in this algorithm code. Only small changes in C++ sources code are required in order to use OpenMP. So each thread gets a different section of the loop, and they execute their own sections in parallel. We executed this code in the quad core computer where we before executed the sequential algorithm. The average execution time is 30 seconds. Significant speedup is reached for the Gram Schmidt algorithm when we use OpenMP parallel features. By trying this C++ code for OpenMP in PC with different number of cores and the execution time is shown in the table 1, is made a speedup analysis.

V. THE GRAM–SCHMIDT VECTOR ORTHOGONALISATION IN MPI

Message Passing Interface (MPI) is a standardized and portable message-passing system designed. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs [9] in different programming languages such as Fortran, C, C++ and Java. Message passing this model uses communication libraries to allow efficient parallel programs to be written for distributed memory systems. These libraries provide routines to initiate and configure the messaging environment as well as sending and receiving data packets. Currently, the most popular high-level message-passing system for scientific and engineering applications is MPI (Message Passing Interface)[4].

We executed the code in a cluster with four computers with the same parameters like the quad core in which we executed the sequential algorithm and OpenMP code. The average time of execution is 22 seconds. The C++ code adopted for using the MPI library for parallelization of our Gram Schmidt vector orthogonalisation algorithm. Both platforms OpenMP and MPI are control flow based architectures. The figure 2 below show the control flow design architecture.

[3] OpenMP Specifications. http://www.openmp.org/blog/specifications/

[4] Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/index.html

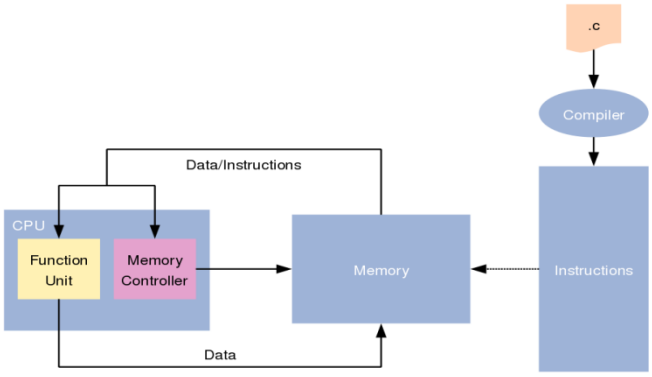

Fig. 4. Control Flow architecture design (source Maxeler)

```
#include <cstdlib>
#include <iostream>
#include <math.h>
#include <mpi.h>
using namespace std;
double b[50][50];
double r[50][50], q[50][50];
int main(int argc, char *argv[]) {
int i, j, rank, nrprocs, count, start, stop;
MPI_Init(&argc, &argv);
// get the number of processes, and the id of this process
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
// we want to perform 50 iterations in total. Work out the
// number of iterations to perform per process...
for (int i=0; i<50; i++)
for (int j=0; j<50; j++)
{b [i][j]=rand() % 10;}
int k;
for (k=0; k<50; k++){
r[k][k]=0; // equivalent to sum = 0
for (i=0; i<50; i++)
r[k][k] = r[k][k] + b[i][k] * b[i][k]; //rkk = sqr(a0k) + sqr(a1k) + sqr(a2k)
r[k][k] = sqrt(r[k][k]);  //
cout << endl << "R"<<k<<k<<": " << r[k][k];
for (i=0; i<3; i++)
{q[i][k] = b[i][k]/r[k][k];
cout << "q" <<i<<k<<": "<<q[i][k] << ", ";}
for(j=k+1; j<50; j++)
{r[k][j]=0; for(i=0; i<50; i++) r[k][j] += q[i][k] * b[i][j];
cout << endl << "r"<< k <<j<<": " <<r[k][j] <<endl;
for (i=0; i<50; i++) b[i][j] = a[i][j] - r[k][j]*q[i][k];
for (i=0; i<50; i++) cout << "b"<<j<<": " << b[i][j]<< ", "; MPI_Finalize();}}
system("PAUSE");
return EXIT_SUCCESS;}
```

Fig. 5. MPI Gram – Schmidt vector orthogonalisation. Program code in C++ (Code::Blocks)

VI. THE GRAM–SCHMIDT VECTOR ORTHOGONALISATION IN MAXELER

Dataflow architecture [10] is a computer architecture that differs in significant contrasts to the traditional Control Flow - Von Neumann architecture. Dataflow architectures do not have a program counter, or (at least conceptually) the executability and execution of instructions is based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable: I. e. behavior is undetermined [5].

Dataflow machines have been around for more than two decades. Implementation challenges left the technology hidden for many years, but last five years the data flow parallel programming is becoming more and more a technological reality. One of the dataflow machines is the Manchester Dataflow Machine (MDFM) using single-assignment language SISAL. Another successful dataflow machine is Maxeler machine. The Gram – Schmidt algorithm is implemented to the MPC-X Series [6] machine.

The implementation process of our algorithm includes the adaption of the C++ host code for export to the MPC module. The Data Flow Engine (DFE) part of an accelerated solution itself contains two components: one or more Kernels, responsible for the data computations; and a single Manager, which orchestrates global data movement for the CPUs, DFEs and Kernels+Memory inside. Hence, accelerating an application requires the user to write three program parts: Kernel(s), A Manager, and a CPU application. The Kernel and the Manager is created by writing programs in MaxJ: an extended form of Java adding operator overloading. Using MaxCompiler requires only minimal familiarity with Java. A developer executes a MaxCompiler-based program to produce a ".max file" containing the DFE configuration, meta-data and SLiC functions. The CPU application is compiled and linked with the .max file, SLiC and MaxelerOS, to create the application executable.

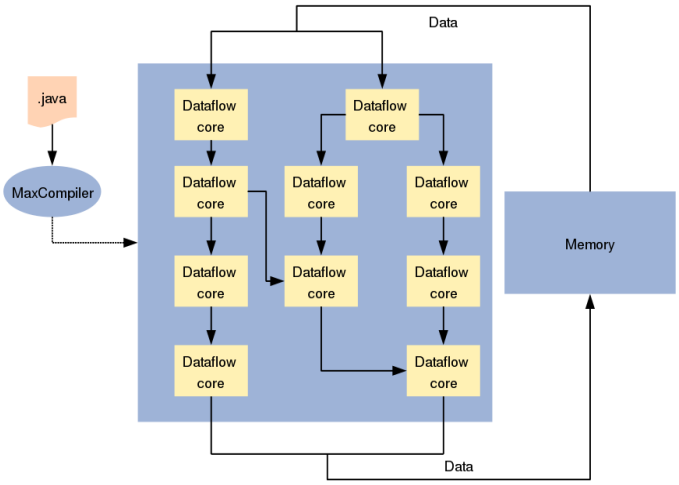

Fig. 6. Data Flow architecture design (source Maxeler)

[5] http://en.wikipedia.org/wiki/Dataflow_architecture, Retrieved on 21 April 2015

[6] https://www.maxeler.com/products/mpc-xseries/, Retrieved on 20 January 2015.

VII. RESULTS

This paper was dealing the behavior of parallel Gram – Schmidt vector orthogonalization algorithms with respect to OpenMP, MPI and Maxeler platform. The results we found are satisfactory. The number of input data size increased Maxeler gives very good performance. Nevertheless, the performance factor presented here is the execution times and speedup of the implementations for same input data size realized in the parallel programming models.

The speedup achieved by a parallel application varies for different programming models. The models chosen in this paper are only considered from the speedup perspective.

The results of the execution time of the algorithms in three machines are shown in figure 7. In this figure is shown the time in seconds in sequential (column 2) in OpenMP (column 3), MPI (column 4) which are Control Flow based architecture. In the figure 7 is shown also the speedup reached in OpenMP (column 5), speedup reached in MPI (column 6) and the speedup reached in the Maxeler machine (column 7). In Maxeler machine, the speedup is high, but limited and independent to the memory performance.

Memory	(Sequential Program) Execution Time	(OpenMP Program) Execution Time	(MPI Program) Execution Time	Speed up Seq/OpenMP	Speed up Seq/MPI	Speed up Seq/Maxeler
1MB	120	60	55	2.00	2.18	20
2MB	90	28	30	3.21	3.00	20
4MB	70	20	25	3.50	2.80	20
8MB	60	15	23	4.00	2.61	20

Fig. 7. Execution time analysis

VIII. CONCLUSIONS

For Gram – Schmidt vector orthogonalisation the parallel approach demands rethinking algorithms, adaption of the programming approach and environment and underlying hardware. There are a lot of possibilities to effectively create parallel version of the algorithms. To be efficient and to have the optimal performance in algorithm execution is very important to select the proper platform related to the contextual problem. In our example is pretty obvious that the Maxeler technology is the most efficient platform. The Maxeler machine spends some initial time for transfer from host to DFE, but control time is extremely slow compare to processing time. Data Flow architecture offers significant capabilities to accelerate scientifically numerical computations, such as the Gram – Schmidt vector orthogonalisation. Improvements in Maxeler and bus technology indicate that Data Flow will increase their lead over general purpose processors over the next few years. In this paper is shown that Maxeler machine with its software system are not wedded to von Neumann architectures nor to the von Neumann execution model. Maxeler platform works very well for calculating Gram – Schmidt vector orthogonalisation reaching a significant speedup.

This paper is addressed to the programmers, by providing taxonomy of parallel language designs. They can decide which language to use for contest of their project.

REFERENCES

[1] Gregory r. Andrews, ACM Computing Surveys, “Paradigms for Process Interaction in Distributed Programs”, Vol 23, No 1, March 1991, page 50-52

[2] Stephen Boyd, Lin Xiao, and Almir Mutapcic, Notes for EE392o, Stanford University, Autumn, 2003: , “Notes on Decomposition Methods”, page 1

[3] Micah Beck, Keshav Pingali, Department of Computer Science Cornell University, Ithaca, NY 14853, “From Control Flow To Dataflow”, page 2

[4] Dounia Khaldi, Pierre Jouvelot, Corinne Ancourt and Franc¸ois Irigoin, “ Task Parallelism and Data Distribution: An Overview of Explicit Parallel Programming Languages”, page 10-16

[5] W, F, McColl, “A General Model of Parallel Computing”, programming research group, Oxford University materials, page 6

[6] Mark A. Nichols, Howard Jay Siegel, Henry G. Dietz, “Data Management and Control-Flow Aspects of an SIMD/SPMD Parallel Language/Compiler”, page 224-225

[7] Christoph Kessler, Jörg Keller, “Models for Parallel Computing: Review and Perspectives”, PARS, Mitteilungen, December 2007, ISBN 0177-0454, page 3

[8] Manar Qamhieh, Serge Midonnet, “Experimental Analysis of the Tardiness of Parallel Tasks in Soft Real-time Systems”, 18th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), May 2014, page 5-6

[9] Board of Trustees of the University of Illinois, “Introduction to MPI”, 2001 page 16

[10] ARTHUR H. VEEN, Center for Mathematics and Computer Science “Dataflow Machine Architecture”, ACM Computing Surveys, December 1986, Vol. 18, No. 4, December 1986, page 2