**PERFORMANCE ANALYSIS OF THE PARALLEL CONJUGATE GRADIENT METHOD: POINT-TO-POINT COMMUNICATION VS COLLECTIVE COMMUNICATION**

**LLOYD MAWUNYEGA DZOKOTO**

**GLOBAL SOFTWARE DEVELOPMENT , HOCHSCHULE FULDA**

**TABLE OF CONTENTS**

1. **ABSTRACT**
2. **INTRODUCTION**
3. **MESSAGE-PASSING INTERFACE**
4. **CONJUGATE GRADIENT METHOD**
5. **PARALLEL CONJUGATE GRADIENT METHOD IMPLEMENTATION**
6. **EXPERIMENTAL RESULTS**
7. **PERFORMANCE ANALYSIS**
8. **SUMMARY**
9. **REFERENCES**
10. **SOURCE CODES**

**1. Abstract**

The objective of this project is to implement in C programming language a parallel Conjugate Gradient (CG) Method. Comparing both the parallel CG method with collective communication and with point-to-point communication would look at a performance analysis.

The parallel CG Method would be implemented using Message Passing Interface (MPI) in order to take advantage of its distributed memory architecture to determine the impact it has on the CG method.

**2. Introduction**

It is obvious that the desire for more computing power cannot be achieved by convectional, single-processor architectures, therefore many companies are investing into developing new architectures with multiple processors.

The Conjugate Gradient (CG) is one of the well known practically used methods in industries today. It was developed by Magnus Rudolph Hestenes with cooperation from Cornelius Lanczos and Edward Stiefel in Zurich . The CG method has become a very popular iterative method for solving systems of linear equations. There are many variants of the CG method which have been implemented but in this project the general CG method would be implemented.

Te CG method is very efficient in solving symmetric positive definite matrices with very high dimensions. The symmetric positive matrices could sparse (most matrix elements are zero), or non-sparse matrices.

The CG method has been found to be a very useful for the following reasons:

* Easier to program and requires less storage space.
* It provides a solution in *n* iteration steps.
* The matrix A is preserved
* A new estimate is determined at each step to serve as an improvement of the previous estimate.
* Much easier to start the iteration again, because the method keeps the last estimate.

**3. Message-Passing Interface (MPI)**

The Message-Passing Interface (MPI) is a library of definitions and functions mostly used in C and Fortran programming languages. The MPI standard is the most well-known specification which is used in supporting parallel programming.

MPI was carefully designed to permit for the parallelization of problems sets that can be spilt into many processes. It provides an efficient and less expensive platform for the implementation of algorithms in clusters and workstations. MPI makes it possible for developers of parallel software to write libraries of parallel program that are both portable and efficient.

**3.1 Architecture**

The underlying architecture of MPI can be pictured to be a collection of processor, each with its own local memory. Each processor has a direct access to data stored in its local memory without any communication required. Message passing between processes is made possible by an interconnection network.

Image goes here

**3.2 Communication**

The two main communication techniques thus Point-to-Point and Collective Communications are used to achieve message passing within processes.

Point-to-Point Communication can be described as a process where Process A sends a copy of its local data to Process B. In a Point-to-Point Communication, there are two MPI methods used to accomplish this namely: MPI\_Send and MPI\_Recv.

Once process A has executed MPI\_Send to send a message to process B, if there is no buffering defined on process B, process A must wait until process B calls MPI\_Recv. This form of communication is known as synchronous mode. Asynchronous mode is achieved when process A calls MPI\_Send to send the message to the buffer of process B. Process B can call MPI\_Recv when its read to receive the message without process A to wait. Asynchronous communication mode is however not recommended.

The second communication technique is Collective Communication. This communication technique involves of processes in the MPI\_COMM\_WORLD communicator. MPI\_COMM\_WORLD communicator consists of all the processes executing the program.

Collective Communications were previously considered points of synchronization thus once a process A calls MPI\_Bcast, all other processes must call MPI\_Bcast before process A can continue execution. However on current systems, process A can call MPI\_Bcast independently of the other processes. What is important to note is process A’s call to MPI\_Bcast must much with the call to MPI\_Bcast by rest of the processes.

MPI has several advantages over other parallel programming models for the following reasons:

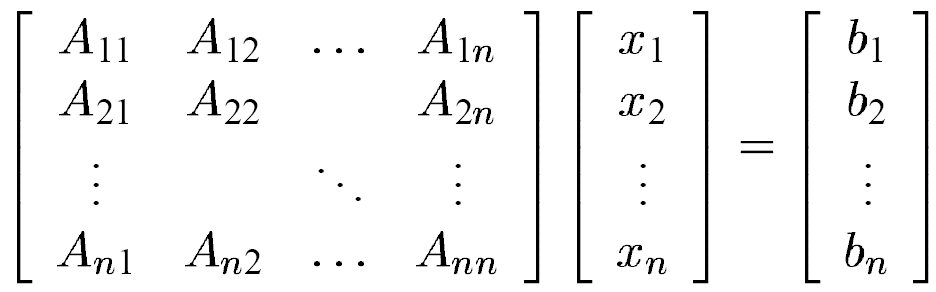
* Runs on a variety of Multiple Instruction Multiple Data (MIMD) architectures.
* Provides a multiprocessor programmer with tools to manage the memory hierarchy.
* Easier to debug compared to shared-memory programs
* Prevents the possibility of overwriting a variable controlled by another process.
* Executes deterministically.

**4. Conjugate Gradient (CG) Method**

This method is used to solve large linear symmetric positive-definite linear equations of the form:

AX = B (*1*)

Equation (1) can be represented in matrix terms as:



Where:

A is a known square , symmetric , positive-definite (or positive-indefinite) matrix.

X is an unknown column vector.

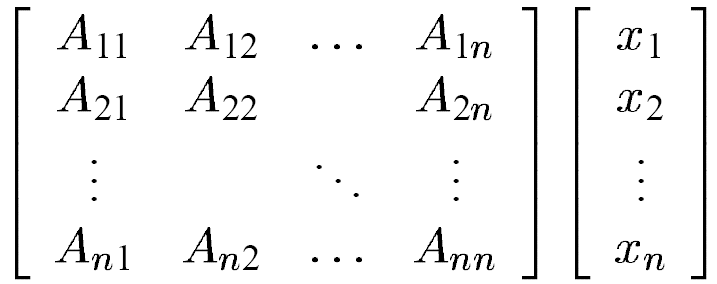
B is a known column vector.

A positive-definite matrix for nonzero vector X is defined as:

XT AX > 0 (*2*)

Expression (2) can be represented as:

[ *x1 x2 … xn* ]



*>0*

A matrix is defined as symmetric if it’s equal to its transpose.

A = AT (3)

Most of the iterative algorithms require stopping criteria to stop the algorithm from executing indefinitely. A large number of the iterative methods use machine epsilon number as the stopping criteria. The CG method however iterates at most the length of vector b.

**4.1 Algorithm**

Consider the function:

*Q(x) = 1/2xT . A . x – xT b + c (4)*

Has a minimizer at *x\**, which approximates the exact solution of equation (1)

The CG method generates a sequence of approximate solutions *x0, x1, x2, …, xn* in order to approximate the exact solution *x\** of equation (1). The approximate solutions are computed such that the next approximation approximates the exact solution with decreasing error. Hence the CG method approximates the exact solution with a high degree of accuracy making it one of the most practically used iterative method.

The procedure of the CG method is as follows:

The execution begins with vectors *x0* and *p0* are considered as zero vectors with:

*R0 = b - Ax0 (5)*

The next approximation is then computed as:

*Xk+1 = xk + αk pk (6)*

The value of the new approximation is dependent of the value of the previous approximation *xk, a scalar step αk and a direction vector pk.*

**4.1.1 Algorithm**

* x0 = 0, r0 = b – Ax0, p0 = r0
* **for** k = 0, 1, 2, 3, . . .n
* αk = (rTkrk) / (pTkApk) step length
* xk+1 = xk + αk pk approximate solution
* rk+1 = rk – αk Apk residual
* βk = (rTk+1 rk+1) / (rTkrk) improvement
* pk +1 = rk+1 + βk pk search direction
* **end**

From the algorithm above, it can be seen that the algorithm performs two operations of matrix-vector multiplications, four operations of inner product and five vector operations. The total number of operations for a symmetric positive-definite matrix can determine from:

*T1 = 2n3 + 13n2 (7)*

With complexity order O (n3).

**5. Parallel Conjugate Gradient (CG) method Implementation**

Parallelization involves a computer or computers with multiple processors to perform a common task.

The parallel CG method was experimented by comparing the performance using Point-to-Point Communication and using Collective Communication.

The test data considered in the experiment are dense matrices and an assumption that the matrix sizes are evenly divisible by the number of processes used in the experiments.

**5.1 Reading data from file**

In the Message-Passing Interface (MPI) world, there is no rule as to how data should be read from standard input or written to standard output by the processes. The approach to achieve reading and writing totally depends on the preference of the developer. Any process or all processes could be assigned the task of reading input data and writing output data.

For the purposes of this project, process 0 is the only process designated to read from standard input and write to standard output.

Since both implementations are using the same data set, the timing overhead involved in reading the data was not considered.

**5.2 Row-wise Block-Striped Decomposition**

Parallelism is achieved for a data-parallel program by dividing the data among processors and each processor processing its own set of data.

In order to perform matrix-vector multiplication within the Parallel CG method, domain decomposition provides each process with a row(s) from the matrix A and the vector b. The vector x is replicated across all processes.In order to perform an inner product, a process requires a row from matrix A and a full column vector of b. However, based on the domain decomposition each process has some parts of vector b. In the Point-to-Point Parallel CG implementation, every process that requires the other elements of vector b to perform an inner product must communicate with the other processes via MPI\_Send and MPI\_Recv. However, in the second implementation, calls to MPI\_Send and MPI\_Recv by processes would be replaced by MPI\_Allgather. Likewise a point-to-point communication to perform a summation in the Point-to-Point implementation would be replaced by MPI\_Allreduce.

Performance related analysis would be examined to determine how communication between the processes influences the general performance of the computation.

**P0**

|  |
| --- |
| B0 |
| B1 |
| B2 |
| B3 |

**P1**

**P2**

**P3**

|  |  |  |  |
| --- | --- | --- | --- |
| A00 | A01 | A02 | A03 |
| A10 | A11 | A12 | A13 |
| A20 | A21 | A22 | A23 |
| A30 | A31 | A32 | A33 |

|  |
| --- |
| X0  X1  X2  X3 |

**A X B**

**Figure 2: Row-wise block striped decomposition of Matrix A and vector B amongst four process.**

**5.3 Computations**

Point-to-Point Communication: Whenever a process needs to perform a matrix-vector multiplication, all process would have to communicate because each process has a portion the vector B. Since the objective of this implementation was to use only point-to-point communication, a function allGather(float\*) was developed. It is very obvious at this point that the minimum number of calls to this function is equal to the number of processes(procsnum). Within each process call to the allGather() function, an amount of procsnum -1 communications are required. Apart fro the allGather function there also allSum(), BcastVector, … which all involve a substantial amount of communication.

The communication within is expected to be relatively higher than the implementation using Collective Communication.

Collective Communication: Unlike the Point-to-Point Communication, Message Passing Interface (MPI) functions like MPI\_AllGather, MPI\_Bcast, MPI\_AllReduce, and MPI\_Scatter where used in the segments of the program whenever all processes had to communicate. These communication functions make at most log2P calls in order to communicate with all processes P.

This greatly reduces the idle time involved in the Point-to-Point Communication. The Collective Communication is therefore expected to give a better performance compared to the Point-to- Point Communication.

**5.4 Taking Timings**

MPI provide an easy and very efficient function to time program execution. In order to measure the program execution time of, the processes must be synchronized at the beginning of the code. The synchronization of the processes is achieved by calling MPI\_Barrier(MPI\_Comm comm). This function would make every process in comm to wait until every process in comm has called it. The actual timing is done by MPI’s timer MPI\_Wtime(void). It returns clock time as a double precision value that represents the number of seconds elapsed.

An average time of the Parallel Conjugate implementation using only Point-to-Point Communication was measured and compared with the Parallel Conjugate implementation using Collective Communication.

**6. Performance Analysis**

The objective of writing parallel programs is to solve bigger problems in less time. However, it is important to keep in mind the cost factor associated in achieving this objective.

For this project, Input and Output (I/O) operations are considered as constants in measuring the performance of the parallel programs. The reasons include the fact that in most instances of computing the Conjugate Gradient (CG) Method in a practical project, the CG method is a subset of a bigger problem to be solved. Hence its I/O operations may be interacting with other programs. Secondly, I/O operations require the computer’s Read-Only Memory (ROM) and Central Processing Unit (CPU).

The two main conditions that affect the performance of parallel programs: input size and number of process. The performance of a parallel program is considered acceptable it its performance compared to its serial variant shows a higher performance. The two performance behaviours that are compared are speedup and efficiency.

Speedup is defined as:

S(n,p) = Tm(n) / Tn(n,p)

Where:

Tm(n) : runtime of the serial program on one (1) process.

Tn(n,p) : runtime of the parallel program on p process.

Efficiency is defined as:

E(n,p) = S(n,p) / P

Where:

S(n,p) : is speedup

P : number of process.

**7. Experimental Results**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Matrix** | **Serial** | **Data dist.** | **CG** | **clock** | **speedup** | **Data dist.** | **CG** | **clock** | **speedup** | **Data dist.** | **CG** | **clock** | **speedup** |
| **512** | 0,005 | 0,096 | 0,01 | **0,136** | **0.037** | 0,112 | 0,008 | 0,15 | 0,033 | 0,103 | 0,007 | 0,153 | 0,033 |
| **1024** | 0,016 | 0,27 | 0,014 | 0,416 | **0.038** | 0,317 | 0,008 | 0,353 | 0,045 | 0,315 | 0,009 | 0,441 | 0,036 |
| **2048** | 0,039 | 1,03 | 0,042 | 1,066 | **0.037** | 1,079 | 0,027 | 1,118 | 0,035 | 1,083 | 0,019 | 1,123 | 0,035 |
| **4096** | 0,186 | 3,754 | 0,182 | **3.965** | **0.047** | 3,925 | 0,121 | 4,083 | 0,046 | 3,962 | 0,065 | 4,066 | 0,046 |
| **8192** | 0,562 | 15,697 | 0,707 | **16.443** | **0.034** | 15,718 | 0,36 | 16,12 | 0,035 | 16,062 | 0,244 | 16,185 | 0,035 |

****

**7.1 Explanation:**

The results from the table shows that the time taken in the point-to-point communication between processes, is relatively equal to the program execution time. This results points out the fact that in using a point-to-point communication, idle time increases with increasing matrix size.

The speedup is ratio of the time taken to execute the serial program to the time taken to execute the same program on p processes. An increase in number the number of p increases communication and gives a low speedup.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| **Matrix** | **Serial** | **Data dist.** | **CG** | **clock** | **speedup** | **Data dist.** | **CG** | **clock** | **speedup** | **Data dist.** | **CG** | **clock** | **speedup** |
| **512** | 0,005 | 0,006 | 0,006 | 0,125 | 0,04 | 0,01 | 0,007 | 0,155 | 0,032 | 0,01 | 0,008 | 0,158 | 0,032 |
| **1024** | 0,016 | 0,02 | 0,011 | 0,319 | 0,050 | 0,033 | 0,007 | 0,363 | 0,044 | 0,052 | 0,026 | 0,384 | 0,042 |
| **2048** | 0,039 | 0,077 | 0,038 | 1,044 | 0,037 | 0,113 | 0,018 | 1,076 | 0,036 | 0,137 | 0,014 | 1,11 | 0,035 |
| **4096** | 0,186 | 0,3 | 0,176 | 3,951 | 0,047 | 0,442 | 0,117 | 4,07 | 0,046 | 0,518 | 0,062 | 4,101 | 0,045 |
| **8192** | 0,562 | 1,195 | 0,685 | 15,603 | 0,036 | 1,751 | 0,457 | 15,971 | 0,035 | 2,036 | 0,234 | 16,013 | 0,035 |



**7.2 Explanation:**

Unlike the Point-to-Point Communication results from the previous table, the Collective Communication shows a much more efficient data distribution. A Collective Communication involves all processes in the communication which takes away idle time. However, the Speed Up behaviour is similar to the Point-to-Point Communication in a decline as number of processes increase.

**8.Summary**

It is now clear the effectiveness of a Collective Communication over a Point-to-Point Communication. Collective Communications is the recommended communicative approach when all processes are involved. On the other hand, a Point-to-Point Communication should be used whenever a process needs to communicate with some process and not all processes executing the program.

**9. References**

**10. Code Samples**