Parallelization of Algorithm for Addition of Two Big Integers

Abhilekh Shrivastava, Ria Kulshrestha,
Soumya Bisht, Vijayvargiya Sagar R.
Department of Information Technology,
National Institute of Technology, Karnataka
Surathkal, Mangaluru, 575025
sbisht2302@gmail.com,
abhilekhshrivastava703@gmail.com,sgvijay1402@
gmail.com,ria.kulshrestha16@gmail.com

Dr. Geetha.V
Department of Information Technology,
National Institute of Technology, Karnataka
Surathkal
Mangaluru, 575025
geethav@nitk.ac.in

Abstract— **Nowadays, computers are able to manipulate numbers directly not longer than 64 bits because the size of the CPU registers and the data-path are limited. Consequently, arithmetic operations such as addition, multiplication etc can only be performed on numbers of that length. In order to solve the problem of computation on big-integer numbers, different algorithms were developed. However, these algorithms are considerably slow because they operate on individual bits and are only designed to run over single-processor computers. Therefore in this paper, two types of algorithms for handling arithmetic addition on big integer numbers are presented. The first type of algorithm is sequential while the second is parallel. The sequential execution of the algorithm is done in C++ while the parallelization of the algorithm is done in OpenMP, MPI and C# under .NET framework .**

Keywords— C++, C# .NET, MPI, Parallelization, OpenMP.

I. Introduction

Modern computers are very good at handling and doing mathematical operations on numbers whose length does not exceed 32 bits or 64 bits. However, when numbers become larger than that, computer arithmetic, such as addition, subtraction, multiplication, and division, becomes almost impossible. This failure is due to the different constraints imposed by the underlying hardware architecture and programming language. Since most of today's computers have CPU with registers, which are 32-bit (4 bytes) and 64-bit (8 bytes) wide, they can only accommodate numbers of that length. Moreover, data types in programming languages are kind of at blame too. For instance, in the foremost programming languages, an int data type can hold up to 32 bits, and a long data type can hold up to 64 bits.

Conventional algorithms and techniques that were developed to solve the problem of arithmetic computation on big numbers convert big numbers from base-10 to base-2 and then they execute bitwise operations on the bit level. For example, arithmetic addition can be performed using the bitwise logical operators OR and XOR. Such algorithms are of complexity O(n), where n is the total number of bits composing each of the big operands.

In this paper, we propose few algorithms for handling arithmetic addition of big-integer numbers. The first one is a sequential algorithm designed to run on single-processor systems, and the second, is a parallel algorithm designed to run on multi-processor shared memory architecture systems. The aim behind these algorithms is to i) enhance the execution time, and to ii) present a parallel implementation for multiple processors systems that drastically decreases that amount of time needed to perform addition on big-integer numbers. Through the results obtained by implementing the suggested algorithm, certain type of applications such as cryptography, financial, astrological, mathematical, and scientific applications will be able to carry out computer arithmetic addition on numbers larger than 64 bits, at high speed.

The paper is structured as follows: Section II provides details on related work, section III explains proposed parallel algorithms of adding two bing integers and section IV provides details about experimental results and discussion followed by conclusion and reference.

II. Related Work

Many programming libraries were developed to solve the problem of performing arithmetic calculations on big-integer numbers. Some of them are proprietary third party dynamic link libraries (DLL), either available for free or sold at a given cost, or shipped as a part of the programming language Application Programming Interface (API). For instance, the MS .NET Framework 4.0 provides the BigInteger class in the namespace System.Numerics. The Java programming language provides another BigInteger class in the java.math package. They both carry out

arithmetic operations on big-integer numbers using bitwise operations. They first convert the base-10 big-integer input to a base-2 binary representation, then they employ the bitwise operators OR and XOR to perform binary addition on string of bits. The algorithm behind these libraries is of complexity O(n), where n is the total number of bits constituting each operand. In terms of time efficiency, the number of times the basic operations OR and XOR is executed, is equal to the number of bits in the big-integer operands. Moreover, most of these libraries are not designed to work in a parallel fashion, but are to operate over single- processor systems. By distributing the integers to be added among the processors, parallel prefix techniques are employed to rapidly add large numbers in a faster manner than if conventional machines were employed.

III Algorithm for addition of big integers

A. Sequential Algorithm for Adding Big Integers using C++

Just like in elementary school how we have learned adding multiple large numbers, in the same manner the sequential algorithm for adding two big integers works. Following is the explanation of sequential algorithm:

1. Two big-integer operands a and b, both of type string and possibly not of the same length, are fed to the a function called AddBigInteger(a,b).

2. Both string operands a and b are then parsed and divided, from right to left, into smaller chunks or tokens ti(p), where i is the token index and p is the operand to which ti belongs. Consequently, operand a = tn-1(a)… t0(a) and operand b = tm-1(b)… t0(b), where n and m, are the total number of tokens constituting each of the operands. The length of each single produced token ti is less than or equal to 18. (In the C# programming language, the largest integer data type is long (signed by default) which can store up to 19 digits. Since in mathematical addition there is always a potential arithmetic overflow, it is crucial to reserve 1 digit for a possible carry, resulting in 19-1=18 digits represented by 60 bits). The resulting tokens will be stored as strings in two arrays, each for a particular operand.

3. The tokens contained in the two arrays are to be converted from string to long data type. In other words, each single token, now representing an array element with a maximum length of 18 digits, is to be converted to an integer value of type long. The conversion is required because arithmetic addition cannot be performed on string types.

4. Both arrays, now containing long type tokens, are aligned on top of each other. Starting from the rightmost token, each two corresponding tokens are added as in performing addition using a pencil and a paper: ti(c) = ti(a) + ti(b) where ti(c) should be less than or equal to 18 digits; otherwise, the leftmost digit (the 19th digit) is truncated and added as a carry to the result of ti+1(a) + ti+1(b); ti+1 is the next token on the left of the two tokens being currently added. It is worth noting that sometimes the length of ti(c) can be less than 18 digits. This is the case when ti(a) and ti(b) are the last leftmost tokens.

5. Finally, all the produced ti(c) are to be concatenated together to attain result = tr-1(c)… t0(c). It is important to note that this algorithm can handle operands of different sizes, in a sense that excessive tokens, which should logically belong to the largest operand, are just appended to the final result.

The figure (Figure 1) below summaries the various steps performed by the sequential algorithm in order to add two operands a and b

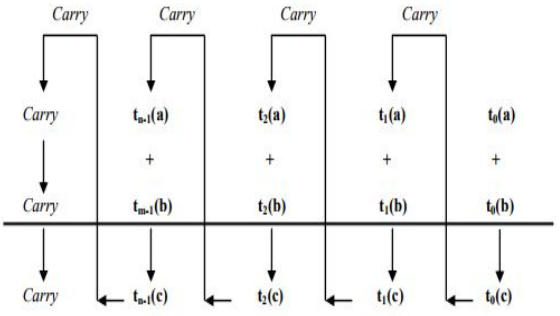


Figure 1: Adding two big numbers using the proposed sequential algorithm

B. Parallel Algorithm for Adding Big Integers

The parallel algorithm proposed in this paper is a multithreaded parallel algorithm designed to be executed over multi-processor shared memory architecture. Ordinarily, the algorithm starts by breaking down big-integer numbers into blocks or tokens of 60 bits each. Then addition starts in a sequence of multiple iterations. On the first iteration, each two corresponding tokens are assigned to a particular thread, which will then add them using a particular microprocessor, while the generated carries from each thread are stored in a shared array. On the second iteration, previous carries stored in the shared array are added properly to the previous result. Iterations continue until no more carries are generated from a previous iteration.

Below are the steps the parallel algorithm execute to add two big-integer numbers:
1. Two very large numbers operand a and operand b, both of string type and possibly not of
the same length, are fed to the algorithm. AddBigInteger_Parallel(a , b)

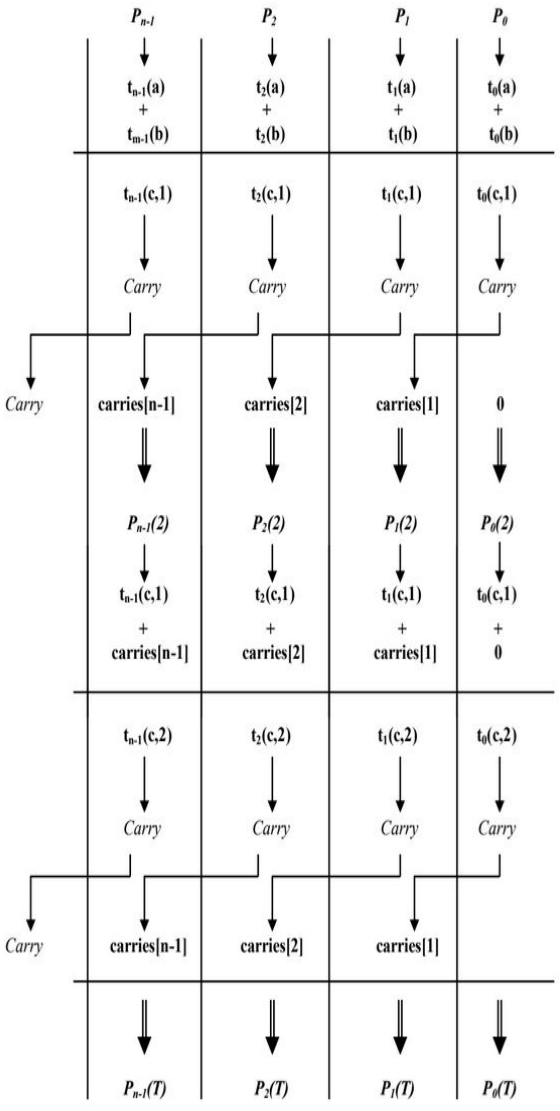
2. Both string operands a and b are then parsed and divided from right to left into smaller
chunks or tokens ti(p), where i is the token index and p is the operand to which ti belongs. Consequently, operand a = tn-1(a)... t0(a) and operand b = tm-1(b)... t0(b), where n and m are the total number of tokens constituting each of the operands. The length of each single produced token ti is less than or equal to 18 (In the C# programming language, the largest integer data type is long (signed by default), and which can store up to 19 digits or 2 raised to 63=9223372036853775808. Since in mathematical addition there is always a potential arithmetic overflow, it is crucial to reserve 1 digit for a possible carry, resulting in 19-1=18 digits represented by 60 bits). The resulting tokens will be stored as string in two arrays, each for a particular operand.

3. The tokens contained in the two arrays are to be converted from string to long data type. In other words each single token, now representing an array element with a maximum length of 18 digits, is to be converted to an integer value of type long. This conversion is required because arithmetic addition cannot be performed on string types

4. Each processor pi in a multiprocessor system is assigned two tokens, one from each operand. Therefore, the processor pi is assigned tokens ti(a) and ti(b) with the purpose of calculating ti(c) = ti(a) + ti(b). For instance, p0 will calculate t0(c), p1 will calculate t1(c), p2 will calculate t2(c) and so on and so forth. We are to assume that the number of processor is equal to the number of tokens; otherwise, tokens are distributed equally among processors. For instance, if the number of processors is half the number of tokens, each processor will be assigned 4 tokens (2 from each operand) to be calculated as in sequential approach. ti(c) = ti(a) + ti(b) and then ti+1(c) = ti+1(a) + ti+1(b)

5. The Carry generated from each ti(c) is handled using multiple processing iterations, and a
shared array called carries[0...n-1] is used to store all the produced carries. For that reason, we have added a new variable called T as in ti(c,T) to represent the iteration into which ti(c) is being calculated. T=1 is the first iteration and T=n is the nth iteration. In this approach, if a carry surfaced after calculating ti(c,1), carries[i+1] is set to 1. It is i+1
so that on the next iteration T=2, carries[i+1] will be correctly added to the previously calculated ti+1(c,1). Likewise, if another carry surfaced from ti(c,2), carries[i+1] is set to 1 overwriting any previous value. Consequently, on the next iteration (T=3) carries[i+1] will be correctly added to ti+1(c,2). This will keep on looping until no more carries are generated (array carries[0...n-1] contains no 1's). As an example, if on the first iteration (T=1), a carry is generated from t4(c,1), then carries[5] is set to 1, p5 (processor 5) starts a second iteration (T=2) in an attempt to calculate t5(c,2) = t5(c,1) + carries[5]. In the
meantime, all other pi, where carries[i]=0, will refrain from executing. If after T=2 no carries was generated, the loop process stops.

6. Finally, all the ti(c) produced after many iterations are to be concatenated together: result= tn-1(c)... t0(c). Figure 2 summaries the different steps performed by the parallel algorithm in order to add two operands a and b.



C. *Parallel Algorithm for Adding Big Integers using C# under .NET framework*

```
public void AddBigInteger_Parallel(string a, string b)
{
    tokens_A = ParseOperand(a);
    tokens_B = ParseOperand(b);
    result = new long[tokens_A.Length];
    carries = new int[tokens_A.Length];
    threads = new Thread[numberOfProcessors];
    CreateThreads();
}

private void Process()
{
    int index = --sharedIndex;
    if (T == 1) // First iteration
    {
        result[index] = tokens_A[index] + tokens_B[index];
    }
    else result[index] = carries[index] + result[index];
    if (index != 0) // not the leftmost token
    {
        if (result[index].ToString().Length > 5)
        {
            carries[index - 1] = 1;
            result[index] = result[index] % 1000000;
        }
        else carries[index - 1] = 0;
    }
    terminatedThreads++;
    IsProcessingDone();
}

private void CreateThreads()
{
    sharedIndex = numberOfProcessors;
    for (int i = 0; i < numberOfProcessors; i++)
    {
        threads[i] = new Thread(new ThreadStart(Process));
        threads[i].Start();
    }
}

private void IsProcessingDone()
{
    if (terminatedThreads == numberOfProcessors)
    {
        if (AreMoreCarries())
        {
            T++;
            CreateThreads();
        }
        else DisplayResults();
    }

}
```

D. *Parallel Algorithm for Adding Big Integers using OpenMP*

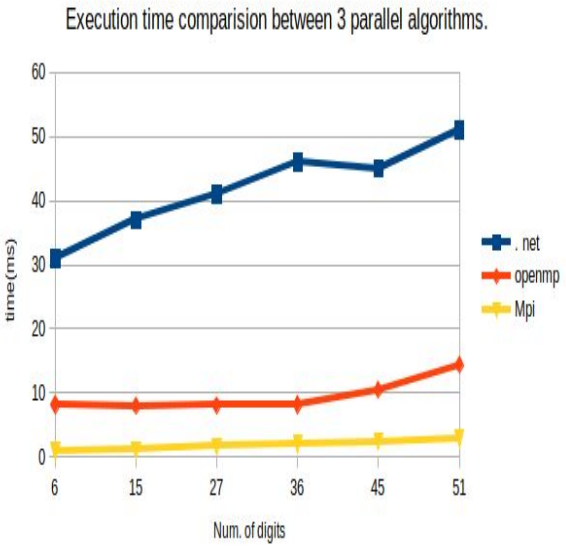
```
#pragma omp parallel shared(a,b,v) firstprivate(carry)
  #pragma omp for
    for(int i=a.size()-1; i>=0 ;i-=blockSize){
      int num1 =0 ,num2=0,sum;
      for(int j=max(i-8,0);j<=i;j++){
        num1=num1*10 + a[j]-'0';
        num2=num2*10 + b[j]-'0';
      }
      sum = num1+num2+carry;
      string sumString = to_string(sum);
      if(sumString.size()>9 && i>8){
        carry=sumString[0]-'0';
        sumString.erase(sumString.begin());
      }
      //cout<<sumString <<" sumString "<<i<<endl;
      if(v[i]=="")
        v[i]=sumString;
      if(carry==1 && i-blockSize>=0){
        tokenadd(a,b,v,1,i-blockSize);
      }
    }
```

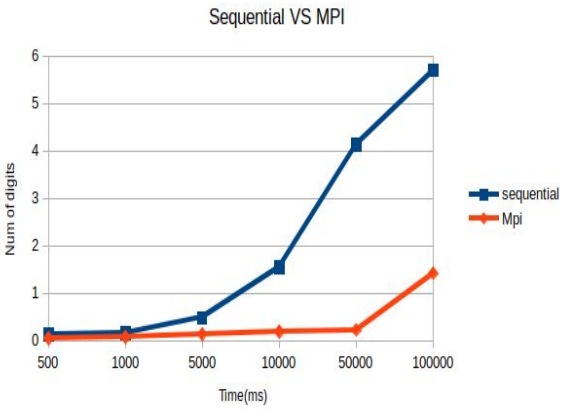
E. *Parallel Algorithm for Adding Big Integers using MPI*

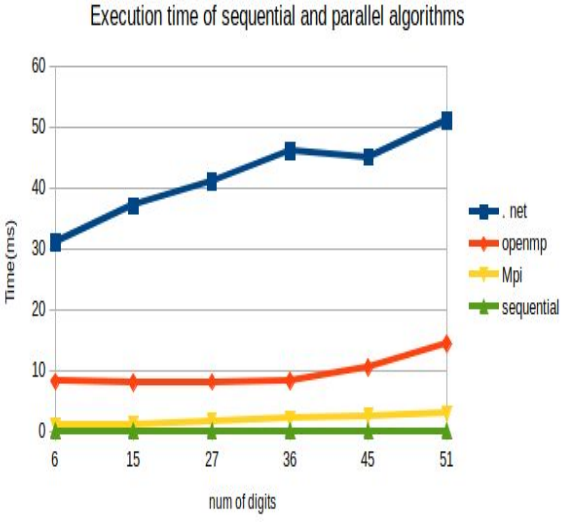
```
void make_sum_worker(int rank, int
num_size, int comm_size)

{

      int tasks = num_size /
comm_size;

      int start_pos = 0; // I have
zero offset in my pieces of nums.

      char *num1 = (char
*)calloc(tasks, sizeof(char));

      char *num2 = (char
*)calloc(tasks, sizeof(char));

      char *sum_with_one = (char
*)calloc(tasks, sizeof(char));

      char *sum_without_one = (char
*)calloc(tasks, sizeof(char));

      assert(num1 && num2 &&
sum_with_one && sum_without_one);

      MPI_Recv(num1, tasks, MPI_CHAR,
0, MPI_ANY_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
```

```
      MPI_Recv(num2, tasks, MPI_CHAR,
0, MPI_ANY_TAG, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

      int sum_with_one_stat = -1;

      if (rank != comm_size - 1)

            sum_with_one_stat =
add_nums(sum_with_one, 1, num1, num2,
start_pos, tasks);

      int sum_without_one_stat =
add_nums(sum_without_one, 0, num1,
num2, start_pos, tasks);

      if (rank == comm_size - 1)


MPI_Send(&sum_without_one_stat, 1,
MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);

      int with_one_stat = 0;

      if (rank != comm_size - 1) {

            MPI_Recv(&with_one_stat,
1, MPI_INT, rank + 1, MPI_ANY_TAG,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

            if (with_one_stat)
      MPI_Send(&sum_with_one_stat, 1,
MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);

            else
MPI_Send(&sum_without_one_stat, 1,
MPI_INT, rank - 1, 0,
MPI_COMM_WORLD);

      }

      if (with_one_stat)

            MPI_Send(sum_with_one,
tasks, MPI_CHAR, 0, 0,
MPI_COMM_WORLD);

      else

            MPI_Send(sum_without_one,
tasks, MPI_CHAR, 0, 0,
MPI_COMM_WORLD);

      free(sum_with_one);

      free(sum_without_one);

}
```

IV Results and Discussion







V Conclusion

Sequential outperforms every other algorithm for small inputs because of the parallel processing overhead and the overhead of creating a parallel environment.

C# took more time than others because of it using an interpreter while the other uses a compiler.

OpenMP takes more time than MPI because it calculates the sum of all tokens assuming there is no carry, and have to recalculate if there is any carry. While in case of MPI both sum with carry and without carry is calculated and is accordingly used depending on the carry.

At the beginning, when operands were upto 50 in length, the difference was not that evident. However, when numbers became larger, the gap increased and the execution time for parallel algorithms was significantly reduced.

VI Reference

1.https://en.wikipedia.org/wiki/Parallel_programming_model

2.https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/

3.https://www.codeproject.com/Articles/701175/PARALLEL-PROGRAMMING-IN-Csharp-AND-OTHER-ALTERNATI

4.https://arxiv.org/ftp/arxiv/papers/1204/1204.0232.pdf

5.http://ijcsit.com/docs/aceit-conference-2016/aceit201653.pdf