

EASTERN MEDITERRANEAN UNIVERSITY



CMPE 523

Parallel and Distributed Programming

Instr: Alexander G. Chefranov

Term Project

Odd/Even Parallel Prefix Algorithm

by Enver Bashirov

24 December 2017

Abstract

This report examines the implementation and testing of the odd/even parallel prefix algorithm. The purpose of this report is to examine the algorithm compared to the single processor working time. Parallel computing does not have a distant past and is still being improved by various new algorithms. Converting an algorithm to a multi threaded/processor one is not so simple but nevertheless, when done so, can be fruitful. Information about parallel computation and history, discussions on the efficiency of the algorithm used and tools and environments worked with, finally the results of the testing carried out can be found in the rest of the report.

Table of Contents

Abstract	1
Table of Contents	2
Parallel Computing Background	3
Parallel Computation Model	3
Distributed Computation Model	4
Client and Server Model	4
Brief History of Parallel Computing	5
Future of Parallel Computing	7
von Neumann Architecture	7
Flynn's Classification	8
Amdahl's Law	9
Project Description	9
Algorithm Description	10
Odd/Even Prefix Algorithm (Original)	10
Improved Algorithm	11
User Documentation	12
System Requirements	12
Network Configurations	13
Firewall Configuration	15
Running the Software	16
Technical Documentation	17
Sequential, Recursive and Parallel Prefix Implementations	17
Server/Client-Side Socket Programming	20
Load Distribution and Distributed Prefix	21
Libraries and Header File	24
Tests and Results	26
Conclusion	29
Appendix	30
References	35

A. Parallel Computing Background

1. Parallel Computation Model

Traditionally, software has been written for serial computation. Creation of serial algorithm is done by breaking problem into a discrete series of instructions in which instructions are executed sequentially, one by one. Executions are done by single processor and at most one execution can be carried out at any moment which is also the description of a serial computation.

Simply, parallel computing is the simultaneous work of multiple computation nodes or processors to come up with a single solution to a given problem. If there is a given problem, the problem is broken down into discrete parts that can be solved concurrently. Each part of the problem is further divided into a series of instructions. Each instruction execution is simultaneous and is on a different processor. Also, the overall coordination mechanism is employed in order to have correct executions at the right time. The final result from each concurrent work is collected into a single one.

By having this definition for parallel computation, one might not think that there could be even more improvements in terms of overall instruction completion since this approach gives the single best time that could be derived from a single machine.

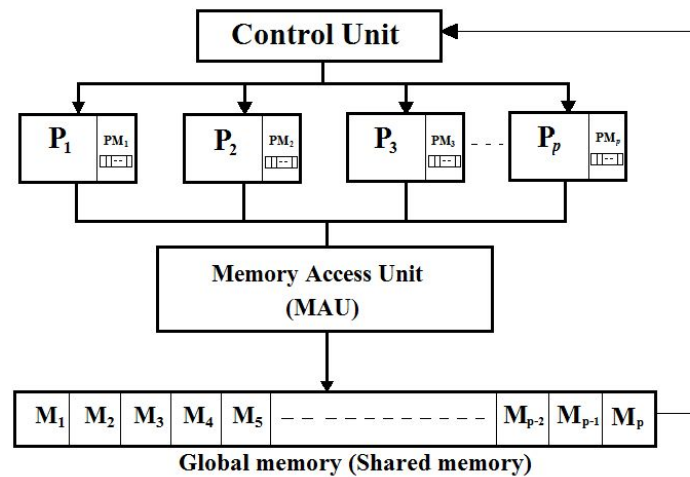


Figure 1. Parallel Computation Model

2. Distributed Computation Model

Think about a system where multiple workers are working simultaneously on a job given by a single officer in order to cut the overall time spent on a job. How convenient, sounds like a parallel computation model. Now, think about group of workers each having an officer doing the work division for them, but also there is this head of department where the main work division is done. If this head of department is a server, the officers are clients and the workers are threads or processors, the distributed computation model arises.

More formally, a distributed computer system consists of multiple software components that are on multiple computers, but run as a single system. The computers that are in a distributed system can be physically close together and connected by a local network, or they can be geographically distant and connected by a wide area network. A distributed system can consist of any number of possible configurations, such as mainframes, personal computers, workstations and so on. The goal of distributed computing is to make such a network work as a single computer ^[2]. Two main benefits of such systems are offered; scalability and redundancy.

On a single machine hardware is a constraint since increasing the core or processor capacity is not possible. When distributed the number of machines become a trait since the capacity can be expanded as much as wanted thus scalability is the first and foremost trait of distributed systems. Redundancy, on the other hand, is also a huge help. There might be machines that at a particular moment might not provide service, so if one machine is unavailable, work is not stopped rather transferred to an another machine.

3. Client and Server Model

A common way of organizing software to run on distributed systems is to separate functions into two parts: clients and servers. A client is a program that uses services that other programs provide. The programs that provide the services are called servers. The client makes a request for a service, and a server performs that service. Server functions often require some resource management, in which a server synchronizes and manages access to the resource, and

responds to client requests with either data or status information. Client programs typically handle user interactions and often request data or initiate some data modification on behalf of a user^[2]. Below figure shows a simple client-server model in which one of each are found.

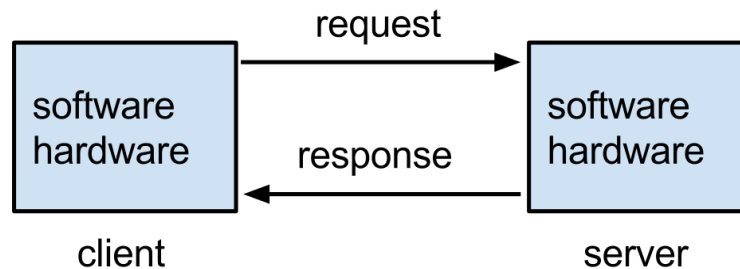


Figure 2. Client Server model

For example, a client can provide a form onto which a user (a person working at a data entry terminal, for example) can enter orders for a product. The client sends this order information to the server, which checks the product database and performs tasks that are needed for billing and shipping^[2]. In our case, server works as a distributor node. As soon as the client joins to server, server gives job to client and they start working on their work individually. After job completions, results are gathered and combined on the server side.

4. Brief History of Parallel Computing

The interest in parallel computing dates back to the late 1950's, with advancements surfacing in the form of supercomputers throughout the 60's and 70's. These were shared memory multiprocessors, with multiple processors working side-by-side on shared data. In the mid 1980's, a new kind of parallel computing was launched when the Caltech Concurrent Computation project built a supercomputer for scientific applications from 64 Intel 8086/8087 processors. This system showed that extreme performance could be achieved with mass market, off the shelf microprocessors. These massively parallel processors (MPPs) came to dominate the top end of computing, with the ASCI Red supercomputer computer in 1997 breaking the

barrier of one trillion floating point operations per second. Since then, MPPs have continued to grow in size and power.

Starting in the late 80's, clusters came to compete and eventually displace MPPs for many applications. A cluster is a type of parallel computer built from large numbers of off-the-shelf computers connected by an off-the-shelf network. Today, clusters are the workforce of scientific computing and are the dominant architecture in the datacenters that power the modern information age.

Today, parallel computing is becoming mainstream based on multi-core processors. Most desktop and laptop systems now ship with dual-core microprocessors, with quad-core processors readily available. Chip manufacturers have begun to increase overall processing performance by adding additional CPU cores. The reason is that increasing performance through parallel processing can be far more energy-efficient than increasing microprocessor clock frequencies.

In a world which is increasingly mobile and energy conscious, this has become essential. Fortunately, the continued transistor scaling predicted by Moore's Law will allow for a transition from a few cores to many.

The software world has been very active part of the evolution of parallel computing. Parallel programs have been harder to write than sequential ones. A program that is divided into multiple concurrent tasks is more difficult to write, due to the necessary synchronization and communication that needs to take place between those tasks. Some standards have emerged. For MPPs and clusters, a number of application programming interfaces converged to a single standard called MPI by the mid 1990's. For shared memory multiprocessor computing, a similar process unfolded with convergence around two standards by the mid to late 1990s: pthreads and OpenMP. In addition to these, a multitude of competing parallel programming models and languages have emerged over the years. Some of these models and languages may provide a better solution to the parallel programming problem than the above "standards", all of which are modifications to conventional, non-parallel languages like C ^[3].

5. Future of Parallel Computing

As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is to transition the software industry to parallel programming. The long history of parallel software has not revealed any “silver bullets,” and indicates that there will not likely be any single technology that will make parallel software ubiquitous. Doing so will require broad collaborations across industry and academia to create families of technologies that work together to bring the power of parallel computing to future mainstream applications. The changes needed will affect the entire industry, from consumers to hardware manufacturers and from the entire software development infrastructure to application developers who rely upon it.

Future capabilities such as photorealistic graphics, computational perception, and machine learning really heavily on highly parallel algorithms. Enabling these capabilities will advance a new generation of experiences that expand the scope and efficiency of what users can accomplish in their digital lifestyles and workplaces. These experiences include more natural, immersive, and increasingly multi-sensory interactions that offer multi-dimensional richness and context awareness ^[3].

6. von Neumann Architecture

John von Neumann (1903-57) was a Hungarian-American mathematician and computer scientist. von Neumann is well known for the computer architecture he named after his surname von Neumann architecture or also known as von Neumann model. He is also founder of the merge sort algorithm.

In von Neumann architecture there are three essential entities, a processing unit, an input-output unit and a storage unit. Processing unit can be broken down into couple of subunits as the arithmetic logic unit, the processing control unit and the program counter. Arithmetic Logic Unit or ALU has the job of doing the computations such as adding and subtracting. The

control unit or PCU controls the data flow through processor. The program counter simply keeps track of what instruction to run at the given moment and when to increments done.

John von Neumann is also known for his great mathematician skills as at his time first steps to the modern computers where taken. von Neumann architecture is one of these first steps which is the building block of our modern day computers.

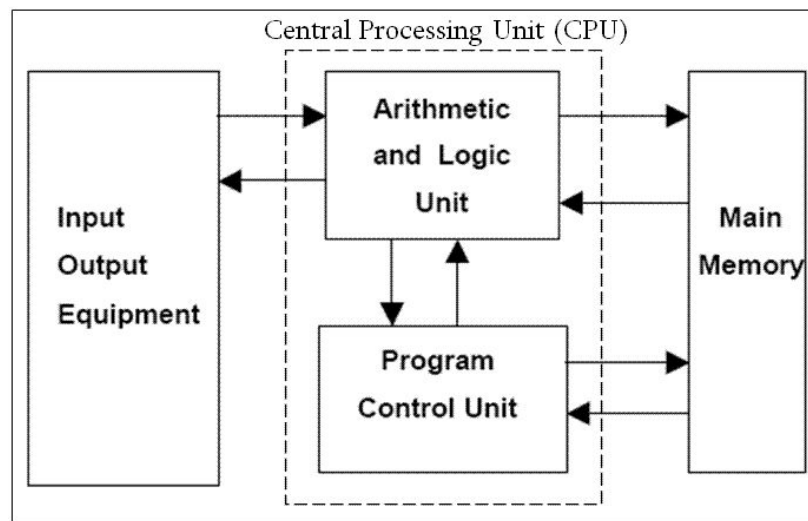


Figure 3. Von Neumann Architecture

7. Flynn's Classification

Michael J. Flynn (1934-current) is an American computer scientist. He worked in IBM and was the founding chairman of IEEE Computer Society. He is well known for Flynn's Taxonomy or Classification.

The four classifications defined by Flynn's classification for computer architectures are single instruction single data (SISD), single instruction multiple data (SIMD), multiple instruction single data (MISD) and multiple instruction multiple data (MIMD).

A sequential computer exploiting no parallelism both in instruction or in data is SISD. However, SIMD defines a parallel computer which exploits parallelism on the data streams against a single instruction. Multiple processing elements work under the control of a single control unit where all processing elements of this organization receive the same instruction broadcast from the control unit. Also, main memory can be divided into parts for generating

multiple data streams thus allowing all the processing elements to execute the same instruction simultaneously.

MISD is when multiple instructions operate on a single data stream. In other words, multiple processing elements are organised under the control of multiple control units, each control unit handling one instruction stream but each processing element is processing only a single data stream at a time. On contrast, MIMD has multiple processors executing on different instructions on different data streams in parallel manner. Multiple instruction streams operate on multiple data streams. In order to be capable of handling multiple instruction streams, multiple CUs and processing elements are organized.

8. Amdahl's Law

Gene M. Amdahl (1922-2015) was an American computer architect who worked in IBM is known for his work on mainframe computers. He is also formulated the famous Amdahl's law which describes the limitations on parallel computing.

Amdahl's law formulates the theoretical speedup and efficiency of a parallel computation. The speedup formulation is as follows;

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

Where S_{latency} is the theoretical speedup of the execution of the task, s is the speedup of the part of the task and p is the proportion of the execution time ^[5].

Speedup and efficiency calculation is also done for the implementation of the Odd/Even algorithm in the upcoming parts of the report.

B. Project Description

The objective of the project is to implement the odd even prefix algorithm to be distributed to at least two machines and for each machine to work in parallel. Initially, together with C++, Message Passing Interface (MPI) was planned to be used but, due to the unsupported

MPI library for the Windows operating system, some parts of the library was deprecated. Thus, rather than MPI, socket programming used to establish connection between hosts.

Completed project software can work in a single machine or in two computers as a server client application. The number of processing elements and size of the number set is not limited by the software but one should know that even if software does not have limitations usually hardware puts these limitations. An example for this would be the maximum positive integer value which is $2^{32}-1$ on a 32-bit system. When reaching this value, software will not stop proceeding to the further iterations of the algorithm but in case of adding another value of more than 0 the result will be an integer overflow jumping to -2^{32} and continuing addition from there.

After completion of implementation, the aim was to calculate an estimation of SpeedUp(p) and Efficiency(p). In order to do this, the algorithm was studied and some improvements were done. Section below describes the original odd even prefix algorithm and the changes made to it in order to get the best performance.

1. Algorithm Description

i. Odd/Even Prefix Algorithm (Original)

Prefix algorithms are used to recursively compute the prefix sum of a sequence. There are some types of prefix algorithms such as parallel prefix or upper lower prefix as well as odd even prefix algorithm. Similarity between the prefix sum algorithms is that they all have divide and conquer approach. In the original odd even prefix algorithm, inputs are divided into groups of odds and evens. Below figure shows the algorithm working on a set of length 8.

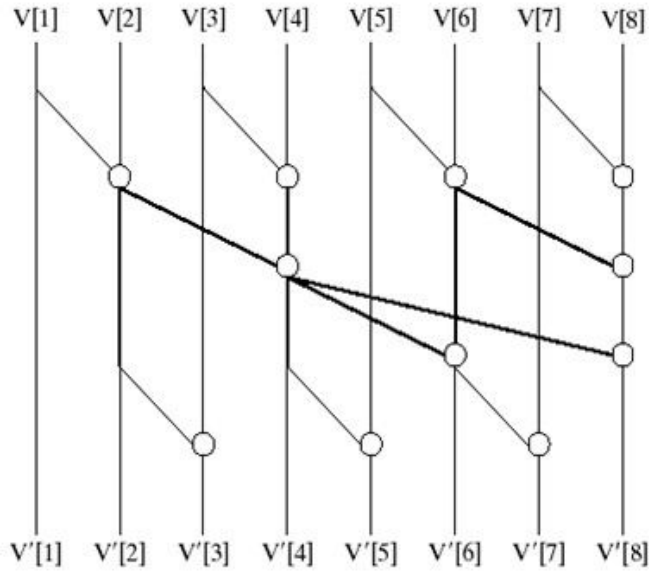


Figure 4. Odd Even parallel prefix algorithm for $N=8$.

After the work division, each odd with the next higher even is combined and this is carried out for the further reduced sets of evens. Finally, combining each even with next higher odd at the output stage. Thus, the problem is reduced to odds and evens and each odd and even is combined at each iteration level.

ii. Improved Algorithm

The aim of the project is to get the overall sum of the set. In order to achieve this with the best performance the original prefix algorithm was improved. The algorithm is made simpler by skipping unnecessary steps that are not needed for the overall sum. Going from the figure 4, these unnecessary steps would be addition of $V[4]$ and $V[6]$ at depth 3 and the all operations on depth 4. The demonstration of the improved algorithm is given below in figure 5 for $N = 8$.

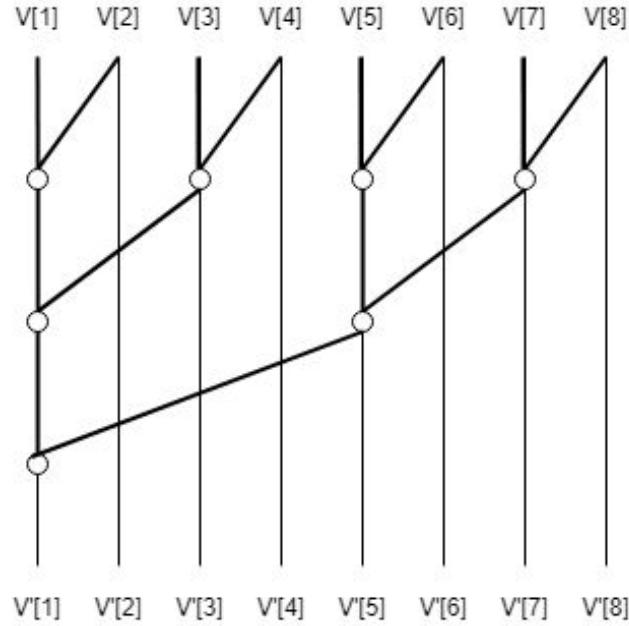


Figure 5. Odd Even improved parallel prefix algorithm for $N=8$.

Comparing the two figures, the unnecessary steps from the original algorithm was removed and the number of operations changed. If the size of the set would be much higher, the change in number of operations drastically reduces.

Generalization of the algorithm would be for a set with length of N . In this case the addition operations are as follows; at first depth $1^{st} = 1^{st} + 2^{nd}$, $3^{rd} = 3^{rd} + 4^{th}$, ..., at second depth $1^{st} = 1^{st} + 3^{rd}$, $5^{th} = 5^{th} + 7^{th}$, ... and for $\text{Depth}(N) = d$, $1^{st} = 1^{st} + \text{floor}(N / 2)^{th}$. If the set length is an odd number, than an extra operation is performed. For example for a set with length 5, the depth would be 3 where at final depth $1^{st} = 1^{st} + 5^{th}$ operation would be performed.

Discussions about size, depth and time complexity and estimation of fraction of sequential work is in Tests and Results part of the report.

2. User Documentation

i. System Requirements

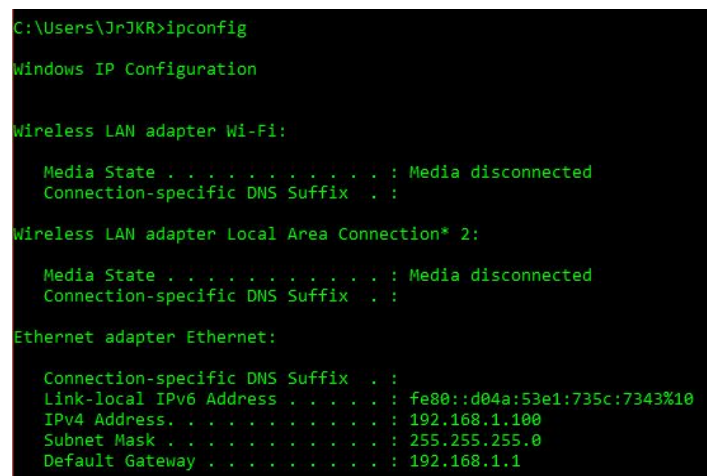
Minimum system requirements are Windows operating system, Visual Studio and Visual C++ compiler, Intel Core 2 Duo or above processor. Since the development environment was Windows 10, Visual Studio 2017 and C++11 it is highly recommended that these are found in

the system. Processor equivalent or above Intel Core 2 duo is recommended due to multithreading since otherwise, the program will be run sequentially and benefits of the algorithm will not be observed. As for the storage and memory, there are no requirements but at least a level of modern storage and memory should be found in the system. Finally, two machines connected to the same network or having a connection to world wide web is required. Details about network requirements and configurations are explained in the next part.

ii. Network Configurations

Project software uses ip address to establish connection between server and client. In order to correctly do this, there are few points to consider. First and easiest way for both machines to connect to each other is for them to be in the same network. However, it is possible to establish connection from an external network. Bare in mind that if network has some sort of port blocking policy, port “27015” should be free and set to allow data transmission. If both client and server are in the same network, this is usually not a problem, otherwise, port forwarding rules should be applied on both networks.

Port forwarding allows a router to let traffic pass through a socket from an incoming external data transmissions. In order to do this, default gateway and IPv4 address should be known. To get the IP configurations command prompt is used. Next command “ipconfig” is entered. There might be many network configurations shown however currently joined network is what is important at this step. Below screenshot shows a similar configuration output.



```
C:\Users\JrJKR>ipconfig

Windows IP Configuration

Wireless LAN adapter Wi-Fi:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Wireless LAN adapter Local Area Connection* 2:

    Media State . . . . . : Media disconnected
    Connection-specific DNS Suffix  . :

Ethernet adapter Ethernet:

    Connection-specific DNS Suffix  . :
    Link-local IPv6 Address . . . . . : fe80::d04a:53e1:735c:7343%10
    IPv4 Address. . . . . : 192.168.1.100
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.1.1
```

Figure 6: Screenshot of IP configurations.

Since, the machine uses ethernet to join to network, “Ethernet adapter Ethernet:” contains the required details. Now that we have IPv4 address and default gateway, we can move on to the next step.

Each router usually has its own configuration page that is accessible from any browser. To access this page, default gateway is entered to the URL section of a browser. In order to access this page router username and password is required. If these credentials are not changed, they often can be found on the back of the router. Below screenshot is an example for this configuration page with steps including how to do port forwarding. Please note that, this particular page belongs to TP-Link TD-W8961N and each router can have its own port forwarding steps thus incase of a different router brand or model, one should look for the specific configuration settings of that particular router.

The screenshot shows the TP-Link web interface for a 300Mbps Wireless N ADSL2+ Modem Router. The browser address bar shows 192.168.1.1/rpSys.html. The page has a navigation menu with tabs: Gelişmiş, Hızlı Kurulum, Arayüz Ayarları, Gelişmiş Ayarlar (selected), Erişim Yönetimi, Bakım, Durum, and Yardım. Under Gelişmiş Ayarlar, there are sub-tabs: Firewall, Yönlendirme, NAT (selected), QoS, VLAN, and ADSL. The NAT configuration page is shown with the following settings:

- Sanal Sunucu : Tekli IP hesabı
- Kural Dizini : 1
- Uygulama : Odd Even
- Protokol : TCP
- Başlangıç Portu : 27015
- Bitiş Portu : 27015
- Yerel IP Adresi : 192.168.1.100

Below the configuration fields is a table titled "Sanal Sunucular" showing the list of virtual servers. The first row is highlighted in red, indicating the current configuration.

Kural	Uygulama	Protokol	Başlangıç Portu	Bitiş Portu	Yerel IP Adresi
1	Odd Even	TCP	27015	27015	192.168.1.100
2	WoL	UDP	60069	60069	192.168.1.100
3	-	-	0	0	0.0.0.0
4	-	-	0	0	0.0.0.0
5	-	-	0	0	0.0.0.0
6	-	-	0	0	0.0.0.0
7	-	-	0	0	0.0.0.0
8	-	-	0	0	0.0.0.0
9	-	-	0	0	0.0.0.0
10	-	-	0	0	0.0.0.0
11	-	-	0	0	0.0.0.0
12	-	-	0	0	0.0.0.0

At the bottom of the page, there are buttons: KAYDET, SİL, GERİ, and İPTAL. The page is annotated with red arrows and text indicating the steps: Step 1 (browser address bar), Step 2 (Gelişmiş Ayarlar tab), Step 3 (NAT sub-tab), Step 4 (configuration fields), Step 5 (KAYDET button), and Result (table row).

Figure 7: Router configurations, port forwarding.

After entering the gateway URL and getting access to the configurations page, following steps should be followed in order, Advanced Settings > NAT > Virtual Server and the next page should be similar to the above screenshot. IPv4 address found earlier, port number “27015” and protocol “TCP” should be entered as details and once save button is clicked, the router should be allowing data transmission for the specific IPv4 address and port number combination over TCP protocol.

iii. Firewall Configuration

One final note on establishing connection is to allow each machines’ local firewall to let the software have internet connection. At the first time running of the program, a windows is prompted for the user asking if firewall should allow connection for the specific program which should be answered as “Allow access”. Figure 8 is an example for this prompt window.

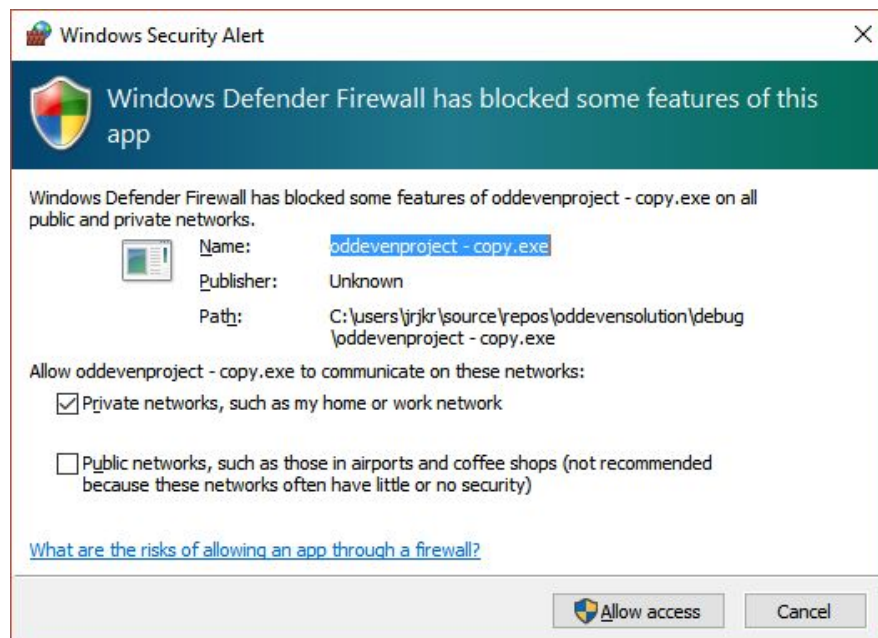


Figure 8: Windows Security Alert

In case this windows is not prompted for the user, user can set the rules manually by going to the firewall settings and using add rule functionality.

iv. Running the Software

There are no installation or setup files included regarding the software but system requirements should be met in order to run the program. Software is a command line application. For running the program, copy of the *OddEvenProject.exe* file should be found on a local directory. From an open command prompt window, either this local directory should be reached or full directory location can be used to run the software. The “-help” command can be used to get the list of commands and their explanations. A screenshot of running helper function is found in appendix, figures 9-12.

Software has three user modes, *local*, *server* and *client*. Each mode has their own function which also can use “-help” function to get the list of functions. See appendix # for the details about methods. Parameters for *local* and *server* user modes are; size of set (*-n <size>*), number of processing elements to use (*-p <value>*), debugging (*-d*) or getting the result (*-res*) can be given as commands.

Type of set generation (*-dist <type> <value1> <value2>*) is also another parameter that could be given by *local* or *server* users. Three set types are included, 0th set is created as {1,1,1,...}, 1st set is created as {1,2,3,...} and 2nd set type is a random generation of values from *<value1>* to *<value2>*.

There are 4 algorithms implemented in the software. SEQ stands for sequential, REC stands for recursive and PAR stands for parallel which can only be accessed from *local* mode. DIS stands for distributed which is the only algorithm that is used by *server* and *client* modes. Distribution function (DIS) establishes server-client connection and load distribution also integrates parallel function (PAR) as its local worker function for performing sum prefix.

Local user have access to SEQ, REC and PAR algorithms by providing *-algorithm <value>* where *<value>* being 0, 1 and 2 respectively. *Server* and *Client* users are restricted to DIS function which they do not need to provide algorithm type. *Client* user has to give *[ip]* parameter as “*client [ip]*” strictly as a first command and number of processing units, debugging or result commands are also provided for *client* users.

3. Technical Documentation

There are three files included as part of implementation, “*main.cpp*”, “*Prefix.cpp*” and “*Prefix.h*”. The “*Prefix.cpp*” and “*Prefix.h*” forms a library for the driving functions and everything is done inside a Prefix object while “*main.cpp*” only has the job of calling this library.

i. Sequential, Recursive and Parallel Prefix Implementations

Both sequential and recursive algorithm implementations are straight forward. Both uses the closest biggest power of 2 to the length of array as the upper limit of iterations which would be the maximum amount of iterations required in a Depth(N) algorithm. The difference between two is that sequential algorithm works as bottom to top, starting from initial depth, whereas recursive algorithm works from top to bottom but the starting point for addition operation is same which is the initial depth.. Below, both sequential and recursive algorithm implementations are given.

```
//SEQUENTIAL Odd/Even Prefix
void Prefix::prefixSeq() {
    if (n == 0) { result = 0; return; }
    for (int j = 2; j <= (int)pow(2, ceil(log2(n))); j = j * 2) {
        int r = j / 2;
        for (int i = j - r; i < n; i += j)
            A[i - (j - r)] += A[i];
    }
}
```

```

//RECURSIVE Odd/Even Prefix
void Prefix::prefixRec() {
    if (n == 0) { result = 0; return; }
    if (n > 1) {
        int j = (int)pow(2, ceil(log2(n))), r = j / 2, i = j - r;
        prefixRec(i, j);
    }
}

void Prefix::prefixRec(int i, int j) {
    int jt = j / 2, rt = jt / 2, it = jt - rt;
    int r = j / 2;
    if (j > 2 && i == (j-r))
        prefixRec(it, j / 2);
    if (i + j < n)
        prefixRec(i + j, j);
    A[i - (j - r)] += A[i];
}

```

Notice that, in both algorithms, “ j ” represents the $2^{\text{CurrentLevel}}$ where the limit of iterations for “ j ” is $2^{\text{Depth}(N)}$. Basically, “ j ” is taken as a reference for the depth and on every iteration is multiplied by 2 in order to get $2^{\text{NextLevel}}$. “ r ” is the increment amount on first index to get the second index for the addition operation. One disadvantage of the recursive algorithm is since it has to keep track of all previous steps to be performed in the future, if N is too large, software will give error due to reaching the limit of the stack.

Following with the parallel prefix algorithm, parallelization process is done with creation of threads which is carried out by the main thread. According to the current depth of algorithm, which is referred as levels, where 0th level would be initial depth, main thread creates and waits for the threads to finish. This process is continued until the final level. Due to the waiting part of the algorithm, no synchronization is required as at each level, no interference in the critical section is required.

```

//Main Thread
void Prefix::prefixPar() {
    while (level < depth) { //Thread creations
        vector<thread> threads;
        for (int i = (r * 2), j = 1; i < n - 1 && j < thrCnt; i +=
            (r * 2), j++) {
            threads.push_back(thread(&Prefix::prefixParWorker,
                *this, i));
        }
        prefixParWorker(0);
        for (auto& th : threads) th.join(); //Waiting for children
        level++, r *= 2;
    }
}

```

Notice that “ r ” is the iteration index of the sum operation. For example, at the first depth, 0th index would contain the sum of 0th and (0+1)th indexed values. Thus, “ r ” at this stage would be 1. Also, the “ $thrCnt$ ” indicates amount of threads to be used. This is the parameter taken from command prompt entered by user initially with “ $-p <thrCnt>$ ” command.

Since, main thread also plays part in the computation process, it also proceeds to the computation function. Driving function here is the “ $prefixPar()$ ” but the actual computation is done in “ $prefixParWorker()$ ” which is has the below implementation.

```

//Worker Thread (Computation Function)
void Prefix::prefixParWorker(int id) { //Child Threads
    for (int i = id;
        ((i % (r * 2)) == 0 && (i + r < Prefix::n));
        i += ((thrCnt)*(r*2)))
        A[i] += A[i + r]; //Addition operation
}

```

“ id ” parameter start from 0 (main thread) and is given to each worker by increment of two, so 1st worker would have $id = 2$. Each worker proceeds to addition operation if $(id+r)$ th index is reachable and if $thrCnt$ is lower than maximum required for the current depth, than some threads proceed for extra iterations of for addition operation (s.t if $n=8$ and $thrCnt=2$ at initial depth, main thread would be performing both 0th+1th and 4th+5th and similarly worker would perform 2th+3th and 6th+7th operations).

ii. Server/Client-Side Socket Programming

In order to establish both-way connection between server and client, “*winsock2.h*”, “*ws2tcpip.h*” and “*windows.h*” libraries are used. Mostly, implementation details for both server-side and client-side codes are similar. The difference is that client will be given the ip address of the server since client is the connection initiator. Below code is the implementation of client and server side connection establishment.

```
// Initialize Winsock
WSAStartup(MAKEWORD(2, 2), &wsaData);

// Resolve the server address and port
getaddrinfo(client ? addr : NULL, DEFAULT_PORT, &hints,
&result);

If (server) {
//Server only section
// Create a SOCKET for connecting to server
socket(result->ai_family, result->ai_socktype,
result->ai_protocol);

// Setup the TCP listening socket
::bind(ListenSocket, result->ai_addr, (int)result->ai_addrlen);
} else {
//Client only section
// Create a SOCKET for connecting to server
ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,

// Connect to server.
connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);

}
freeaddrinfo(result);

iRecResult = listen(ListenSocket, SOMAXCONN);

// Accept a client socket
accept(ListenSocket, NULL, NULL);
```

Notice that server creates a listening port initially. This is a must since when client is trying to connect it should be able to find this particular socket. The parameter “*addr*” when trying to resolve server address in server-side implementation is set to “*NULL*” as it will get its

own server-side ip address. The “*DEFAULT_PORT*” parameter is same in both sides and was set to “27015”.

```
#define DEFAULT_PORT "27015"
```

Finally, the connection is established between server and client. The send and receive operations can be done.

```
// RECEIVE  
  
recv(Socket, receive_buf, receive_buflen, 0);  
  
// SEND  
  
send(Socket, send_buf, send_buflen, 0);
```

iii. Load Distribution and Distributed Prefix

The distribution process is done by the server. First server generates the array according to the user input taken from the command prompt, “*iniArr()*” function is used for generation of array and values. Client has no clue about the length and content of the array. In case of random number generation “*ranRange[0]*” and “*ranRange[1]*” holds the lower and upper bound for the random number;

```

void Prefix::iniArr() {
    if (n == 0) { A = NULL; return; }
    delArr();
    A = new int[n];

    random_device rd;
    mt19937 mt(rd());
    uniform_real_distribution<double> dist(ranRange[0],
ranRange[1]);

    for (int i = 0; i < n; i++) {
        switch (arrType) {
            case 0: A[i] = 1; break;
            case 1: A[i] = i; break;
            case 2: A[i] = (int)dist(mt); break;
            default: A[i] = 1; break;
        }
    }
}

```

After creating the array, server divides this array into two halves, if the length of array is odd then the length of first half of the array is set to $\text{floor}(N/2)$ and the length of second half is set to $\text{ceil}(N/2)$. This division is computed by the following two functions.

```

int * Prefix::arrFirstHalf(int * arr, int n) {
    int *arr1, n1 = (int)floor((double)n / 2);
    arr1 = new int[n1];
    for (int i = 0; i < n1; i++) {
        arr1[i] = arr[i];
    }
    return arr1;
}

int * Prefix::arrSecondHalf(int * arr, int n) {
    int *arr2, n2 = (int)ceil((double)n / 2);
    arr2 = new int[n2];
    for (int i = 0; i < n2; i++) {
        arr2[i] = arr[i+n2-1];
    }
    return arr2;
}

```

Finally, the first half of the array is converted into *string*. After client completes the computation and sends the result, the received data is again converted from *string* to *int array*

and merged. Following three function implementations served for conversion to *string*, conversion to *int array* and lastly merging the arrays respectively.

```
//Int array to string
string Prefix::arrToStr(int * arr, int n) {
    string str;
    for (int i = 0; i < n; i++) str += to_string(arr[i]) + ",";
    return str;
}

//String to int array
int * Prefix::strToArr(string str) {
    string delimiter;
    size_t pos = 0;
    string token;
    delimiter = "n";

    if ((pos = str.find(delimiter)) != string::npos) {
        token = str.substr(0, pos);
        Prefix::n = stoi(token);
        str.erase(0, pos + delimiter.length());
    } delimiter = ",";
    int *arr = new int[n];
    for (int i = 0; (pos = str.find(delimiter)) != string::npos;
i++) {
        token = str.substr(0, pos);
        arr[i] = stoi(token);
        str.erase(0, pos + delimiter.length());
    }
    return arr;
}

//Array merging
int * Prefix::arrMerge(int * arr1, int * arr2, int n) {
    int n1 = (int)floor((double)n / 2);
    int n2 = (int)ceil((double)n / 2);
    int *arr3 = new int[n1 + n2];
    for (int i = 0; i < n1; i++) arr3[i] = arr1[i];
    for (int i = n2 - 1; i < n1 + n2; i++) arr3[i] = arr2[i - n2];
    return arr3;
}
```

After client has been joined to the server, server divides the array into two halves, converting array to suitable format and sends the data to client. Client and server proceed to

computations. When client is finished with the computation, sends the resulting array back to server, server merges the two halves of the array and finalizes the overall sum prefix by adding first index of second half of the array to the first index of the first half. Below is both client and server side code for the steps explained above.

```
/*(Server) Dividing array into two halves*/
int *arr1 = arrFirstHalf(A, localn), *arr2 = arrSecondHalf(A,
localn);

/*(Server) Converting array into char * array*/
char * sendbuf;
string arrStr = to_string(n1) + "n" + arrToStr(arr1, n1);
sendbuf = new char[arrStr.size() + 1];
strcpy_s(sendbuf, arrStr.size() + 1, arrStr.c_str());

/*(Server) Sends array to client and starts working on the
computation of sum of second half*/
send(ClientSocket, sendbuf, (int)strlen(sendbuf) + 1, 0);

/*(Client) Converting received data into int array and start working
on first half of computations*/
recv(ConnectSocket, recvbuf, recvbuflen, 0);
Prefix::A = strToArr( string(recvbuf));

/*(Both) Start working on the task
prefixPar();

/*(Client) Converts resulting array and sends data back to server*/
string arrStr = arrToStr(Prefix::A, Prefix::n);
char * sendbuf = new char[arrStr.size() + 1];
strcpy_s(sendbuf, arrStr.size() + 1, arrStr.c_str());
send(ConnectSocket, sendbuf, (int)strlen(sendbuf) + 1, 0);
```

```

/*(Server) After completing computation for the second half, receives
the array, converts and merges. Performs the last step of sum prefix
computation.*/
recv(ClientSocket, recvbuf, recvbuflen, 0);
arr1 = strToArr(recvbuf);
arr1[0] += arr2[0];
Prefix::A = arrMerge(arr1, arr2, localn);

```

iv. Libraries and Header File

“*Prefix.h*” is the header file that hold all the partner object variables, member functions and constructor declarations. The complete representation of the header file is as follow;

```

private:
    //Member Variables & Constants
    int *A = NULL;
    int n, result = -1;
    int algorithm = 0; double ranRange[2] = { 1, 10 };
    string algName[5] = { "Seq", "Rec", "Par", "Dis", "NULL" };
    int arrType = 4, thrCnt = -1;
    bool debug = false, debugRes = true;

public:
    //CONSTRUCTORS & DECONSTRUCTORS
    Prefix();
    Prefix(int n, int thrCnt, int arrType, bool debug);
    ~Prefix();

    /*****DRIVER FUNCTIONS*****/
    //DISTRIBUTED Odd/Even Prefix
    void prefixDis();
    void prefixDis(char * addr);
    void prefixDisServer();
    void prefixDisClient(char * addr);

    //Parallel Odd/Even Prefix
    void prefixPar();
    void prefixParWorker(int id);

    //Recursive Odd/Even Prefix
    void prefixRec();
    void prefixRec(int i, int j);

    //Sequential Odd/Even Prefix
    void prefixSeq();

```

```

//Usefull Functions
void menu(int argc, char* argv[]);

//GETTERS & SETTERS
int * getArr();
int getVal(int i);
void setArr(int * arr);

//Internal Functions
void iniArr();
void delArr();
string arrToStr(int * arr, int n);
int * strToArr(string str);
int * arrFirstHalf(int * arr, int n);
int * arrSecondHalf(int * arr, int n);
int * arrMerge(int * arr1, int * arr2, int n);
void printArr();
void printRes();

```

The member variables are not public, thus they are not reachable from outside the “*Prefix*” class, as they should be for the security reasons. However, some getter and setter functions have been implemented in order to have some flexibility.

The complete list of libraries used for mathematical operations, printing on console, string parsing, keeping the time for finding run time, random number generation etc. is as follow;

```

#include <windows.h> //Socket Programming
#include <winsock2.h> //Socket Programming
#include <ws2tcpip.h> //Socket Programming

#include <stdlib.h> //Converting char to string etc.
#include <stdio.h> //Read, write access (printf)
#include <iostream> //Read, write access (cout, cin)
#include <string> //String operations (parsing)
#include <math.h> //Log, pow etc.
#include <vector> //Vector operations
#include <thread> //Creating, joining threads
#include <mutex> //Synchronization
#include <chrono> //Measuring time
#include <random> //Generating random numbers

```

C. Tests and Results

Let's jump into the size, depth and time complexity of the problem. When parallel algorithm is run with 1 processing unit (simulating sequential algorithm), the size and depth of the algorithm would be;

$$Size_l(N) = Depth_l(N) = N-1$$

However with the help of parallelism (when processing unit count is maximized), size and depth are;

$$Size_p(N) = N-1 ; Depth_p(N) = \text{ceil}(\log_2 N)$$

Notice that size of the problem did not change even though parallelism introduced. This is simply because each value should at least be visited once even if the number of processors are increased. Nevertheless, the issue never was the size, since the time complexity of the algorithm is proportional to the depth. By reducing depth, new running time of the algorithm is;

$$O(\text{ceil}(\log_2 N))$$

All the tests are carried out with set generation is set to random integers from 10 to 1000. The value T_1 is for the 1 processor, T_p is for the maximum number of processors which is $\text{floor}([n/2])$ and T_{dp} is for the server client with maximum number of processors.

Calculation of f, Speedup and Efficiency is as follows;

$$f = \frac{T_p - T_1 / p}{T_1(1 - 1/p)}$$

$$\text{Speedup } S(p) = T_1 / T_p$$

$$\text{Efficiency } E(p) = S(p) / p$$

N	10000	20000	50000	100000	1000000	10000000	100000000
$T_1(\text{ms})$	2.4553	2.7282	3.2102	3.0961	10.529	81.525	811.36
$T_8(\text{ms})$	10.197	9.4727	12.200	14.142	23.828	68.806	723.43
$T_p(\text{ms})$	2.6070	2.1167	2.5983	3.0016	9.4723	80.731	809.63
$T_{dp}(\text{ms})$	302.30	485.51	796.39	1300.4	-	-	-
$S(8)$	0.2408	0.2880	0.2631	0.2189	0.4418	1.1849	1.1215
$S(p)$	0.9418	1.2889	1.2355	1.0315	1.1115	1.0098	1.0021
$S(dp)$	0.0081	0.0056	0.0040	0.0023	-	-	-
$E(8)$	0.0301	0.0360	0.0328	0.0274	0.0552	0.1481	0.1402
$E(p)$	1.8E-4	1.2E-4	4.9E-5	2.1E-5	2.2E-6	2.0E-7	2.0E-8
$E(dp)$	6.7E-4	4.7E-4	3.3E-4	1.9E-4	-	-	-
f_8	0.1171	0.4184	0.4594	0.5553	0.2672	0.0899	0.0958
f_p	2.1E-4	7.6E-6	4.0E-5	1.9E-5	1.8E-6	2.0E-7	2.0E-8
f_{dp}	10.253	14.823	20.666	34.994	-	-	-

Table 1: Speedup, Efficiency and estimation of fraction f of sequential part.

Processing Units	p(1)	p(8)	p(floor(N/2))	p _d (12)
Big O	O(N)	O(N)	O(ceil(log ₂ N))	O(N)

Table 2: Running time of $p = 1, 8, \text{floor}(N/2)$ (maximum processing units) and $p_d = 12$.

For references to the algorithm working is in appendix, figures 13-19. Notice that even though the “big O” for f_p is the best, from table 1 it is clear that when $p(8)$ algorithm is most efficient. This is because the algorithm is run on 8 core CPU. If the tests could have done for even larger amount of N , algorithm with 8 processing units will surpass algorithm with maximal processing units.



Graph 1-4: Running time, speedup, efficiency and fraction of sequential work representations as a line graph.

For low number of N , T_1 is better but when the number of N increased multithreaded application is more efficient. This also means if more number of physical processing units were found in the system, the running time for large N would have been even smaller. In the graphs and table, the distributed algorithm took too much time to respond due to the waiting for

client/server and sending array through network thus even tests couldn't be carried out for large and was not included in some of graphs.

D. Conclusion

The project was tested and results were gathered using one server and one client machine. The number N was chosen according to the differentiation of the completion time of the algorithm. Since the numbers are generated randomly, it is possible that the execution time is not uniform. Results of the experiments show that, in a perfect environment where infinite number of processing units and machines were found with minimal amount of time spent on operations such as server client communication, distributed algorithm would be the obvious choice to perform odd even prefix algorithm. Point of moral here is, in reality, such infinite resources are not available and according to the available processing elements, the algorithm should be informed.

Appendix

```
C:\Users\JrJKR>C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe -help
Usage: C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe <host_type> [options]

<host_type>
Values | Description
local  | Running prefix algorithms locally.
server | Running a server for distributed prefix algorithm.
client | Joining to a server for distributed prefix algorithm.

For more information: <host_type> -h.
```

Figure 9: Help for `host_type` choice.

```
C:\Users\JrJKR>C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe local -help
Length of Array.
Option: -n [value].

Number of Processing units.
Option: -p [value] (Default: POSSIBLE_MAX).

Algorithm Type
Option: -algorithm / -a [value]
Values | Description
0      | Sequential Odd/Even Prefix (Default).
1      | Recursive Odd/Even Prefix.
2      | Parallel Odd/Even Prefix.

Distribution Type: defining values of the array.
Option: -distribution / -dist [value1] [value2]
Values | Description
0      | All values are 1.
1      | Starting from 0, values are incremented by 1.
2      | Randomly generated integers from [value1] to [value2]

Debug.
Option: -debug / -d

Print Result.
Option: -res
```

Figure 10: Help function for local user.


```

C:\Users\JrJKR>C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe server -help

Server: Distributed and Parallel Odd/Even Prefix algorithm will be performed.
In order to proceed, client must be joined

Usage: C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe server -n [value] [options]

Length of Array.
Option: -n [value].

Number of Processing units.
Option: -p [value] (Default: POSSIBLE_MAX).

Distribution Type: defining values of the array.
Option: -distribution / -dist [value1] [value2]
Values | Description
0      | All values are 1.
1      | Starting from 0, values are incremented by 1.
2      | Randomly generated integers from [value1] to [value2]

Debug.
Option: -debug / -d

Print Result.
Option: -res

```

Figure 11: Help function for server user.

```

C:\Users\JrJKR>C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe client -help

Client: Distributed and Parallel Odd/Even Prefix algorithm will be performed.
In order to proceed, must be joining to waiting server.

Usage: C:\Users\JrJKR\source\repos\OddEvenSolution\Debug\OddEvenProject.exe client [ip] [options]

Number of Processing units.
Option: -p [value] (Default: POSSIBLE_MAX).

Debug.
Option: -debug / -d

Print Result.
Option: -res

```

Figure 12: Help function for client user.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 10000
Par | Result | 5010676
Total time: 2455310 ns
Total time: 2 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 20000
Par | Result | 10103210
Total time: 2728298 ns
Total time: 2 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 50000
Par | Result | 25306384
Total time: 3210273 ns
Total time: 3 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 100000
Par | Result | 50573408
Total time: 3096100 ns
Total time: 3 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 1000000
Par | Result | 504037837
Total time: 10529189 ns
Total time: 10 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 10000000
Par | Result | 752346893
Total time: 81525760 ns
Total time: 81 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 1 -n 100000000
Par | Result | -1089952319
Total time: 811369012 ns
Total time: 811 ms

```

Figure 13: Local parallel algorithm working at 1 processing units (simulating sequential runtime).

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 10000
Par | Result | 5050960
Total time: 10197337 ns
Total time: 10 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 20000
Par | Result | 10066769
Total time: 9472794 ns
Total time: 9 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 50000
Par | Result | 25206357
Total time: 12200301 ns
Total time: 12 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 100000
Par | Result | 50409614
Total time: 14142030 ns
Total time: 14 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 1000000
Par | Result | 504390993
Total time: 23828552 ns
Total time: 23 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 10000000
Par | Result | 750770191
Total time: 68806348 ns
Total time: 68 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -p 8 -n 100000000
Par | Result | -1088247321
Total time: 723434163 ns
Total time: 723 ms

```

Figure 14: Local parallel algorithm working at 8 processing units.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 10000
Par | Result | 5052356
Total time: 2607013 ns
Total time: 2 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 20000
Par | Result | 10113091
Total time: 2116742 ns
Total time: 2 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 50000
Par | Result | 25205812
Total time: 2598322 ns
Total time: 2 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 100000
Par | Result | 50260942
Total time: 3001680 ns
Total time: 3 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 1000000
Par | Result | 504562808
Total time: 9472399 ns
Total time: 9 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 10000000
Par | Result | 750907520
Total time: 80731686 ns
Total time: 80 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe local -a 2 -res -dist 2 10 1000 -n 100000000
Par | Result | -1090433771
Total time: 809639826 ns
Total time: 809 ms

```

Figure 15: Local parallel algorithm working at maximum number of $(N / 2)$ processing units.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe server -res -dist 2 10 1000 -p 8 -n 10000
Dis | Result | 2568127
Dis | Result | 5089076
Total time: 302309256 ns
Total time: 302 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe server -res -dist 2 10 1000 -p 8 -n 20000
Dis | Result | 5000436
Dis | Result | 10005739
Total time: 485147846 ns
Total time: 485 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe server -res -dist 2 10 1000 -p 8 -n 50000
Dis | Result | 12605691
Dis | Result | 25203670
Total time: 796393796 ns
Total time: 796 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe server -res -dist 2 10 1000 -p 8 -n 100000
Dis | Result | 25276123
Dis | Result | 50452182
Total time: 1300413354 ns
Total time: 1300 ms

```

Figure 16: Servers view of distributed algorithm.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe client 192.168.1.100 -res -p 4
Dis | Result | 2520949
Total time: 95533075 ns
Total time: 95 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe client 192.168.1.100 -res -p 4
Dis | Result | 5005303
Total time: 183499134 ns
Total time: 183 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe client 192.168.1.100 -res -p 4
Dis | Result | 12597979
Total time: 496659949 ns
Total time: 496 ms

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe client 192.168.1.100 -res -p 4
Dis | Result | 25176059
Total time: 1044828068 ns
Total time: 1044 ms

```

Figure 17: Clients view of distributed algorithm.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe server -res -dist 2 10 1000 -p 8 -n 8 -d
Dis | Init |
DIS SER | Received: clientready Bytes: 12
DIS SER | Sent: 4n189,290,246,858, Bytes: 12

Par | Init |
Par | Creating Child 2
Par | id:2 r:1 ind:[2]=667
Prefix destroyed.

Par | Creating Parent
Par | id:0 r:1 ind:[0]=909
Par | Creating Parent
Par | id:0 r:2 ind:[0]=1576
Par | Completed.
Dis | Result | 1576
DIS SER | Received: 1583,290,1104,858, Bytes: 19
DIS SER | Sent: serversuccess Bytes: 19
DIS SER | Received: clientleaving Bytes: 14
DIS SER | Sent: serverleaving Bytes: 14
Dis | Result | 3159
Total time: 297423920 ns
Total time: 297 ms

Prefix destroyed.

```

Figure 18: Example for debugging on server side.

```

C:\Users\JrJKR\source\repos\OddEvenSolution\Debug>OddEvenProject.exe client 192.168.1.100 -res -p 4 -d
DIS CLI | Sent: clientready Bytes: 0
DIS CLI | Received: 4n189,290,246,858, Bytes: 19

Par | Init |
Par | Creating Child 2
Par | id:2 r:1 ind:[2]=1104
Prefix destroyed.

Par | Creating Parent
Par | id:0 r:1 ind:[0]=479
Par | Creating Parent
Par | id:0 r:2 ind:[0]=1583
Par | Completed.
Dis | Result | 1583
DIS CLI | Sent: 1583,290,1104,858, Bytes: 19
DIS CLI | Received: serversuccess Bytes: 14
DIS CLI | Sent: clientleaving Bytes: 14
DIS CLI | Received: serverleaving Bytes: 14
Total time: 17156352 ns
Total time: 17 ms

Prefix destroyed.

```

Figure 19: Example for debugging on client side.

References

- [1]"Introduction to Parallel Computing", Computing.llnl.gov, 2017. [Online]. Available: https://computing.llnl.gov/tutorials/parallel_comp/#Abstract. [Accessed: 25- Dec- 2017].
- [2]"IBM Knowledge Center", Ibm.com, 2017. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.2.0/com.ibm.cics.tx.doc/concepts/c_wht_is_distd_comptg.html. [Accessed: 25- Dec- 2017].
- [3]"The History of the Development of Parallel Computing", Webdocs.cs.ualberta.ca, 2017. [Online]. Available: <https://webdocs.cs.ualberta.ca/~paullu/C681/parallel.timeline.html>. [Accessed: 25- Dec- 2017].
- [4]"Von Neumann architecture", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Von_Neumann_architecture. [Accessed: 25- Dec- 2017].
- [5]"Amdahl's law", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed: 25- Dec- 2017].