EVALUATIONS OF THE PARALLEL EXTENSIONS IN .NET 4.0

Master's (one year) thesis in Informatics (15 credits)

Md. Rashedul Islam
Md. Rofiqul Islam
Tahidul Arafhin Mazumder

Autumn 2010:MI20

Title: Evaluations of the parallel extensions in .NET 4.0

Year: 2010

Author/s: Md. Rashedul Islam, Md. Rofiqul Islam and Tahidul Arafhin Mazumder

Supervisor: Håkan Sundell

Abstract

Parallel programming or making parallel application is a great challenging part of computing research. The main goal of parallel programming research is to improve performance of computer applications. A well-structured parallel application can achieve better performance in terms of execution speed over sequential execution on existing and upcoming parallel computer architecture. This thesis named "**Evaluations of the parallel extensions in .NET 4.0**" describes the experimental evaluation of different parallel application performance with thread-safe data structure and parallel constructions in .NET Framework 4.0. Described different performance issues of this thesis help to make efficient parallel application for better performance. Before describing the experimental evaluation, this thesis describes some methodologies relevant to parallel programming like Parallel computer architecture, Memory architectures, Parallel programming models, decomposition, threading etc. It describes the different APIs in .NET Framework 4.0 and the way of coding for making an efficient parallel application in different situations. It also presents some implementations of different parallel constructs or APIs like Static Multithreading, Using ThreadPool, Task, Parallel.For, Parallel.ForEach, PLINQ etc. The evaluation of parallel application has been done by experimental result evaluation and performance measurements. In most of the cases, the result evaluation shows better performance of parallelism like less execution time and increase CPU uses over traditional sequential execution. In addition parallel loop doesn't show better performance in case of improper partitioning, oversubscription, improper workloads etc. The discussion about proper partitioning, oversubscription and proper work load balancing will help to make more efficient parallel application.

Keywords: Parallel application, Decomposition, Multithreading, Parallel frameworks, Data Parallelism, Task parallelism, Parallel Performance.

Acknowledgements

We would like to express our sincere gratitude to our supervisor Dr. Håkan Sundell, Ph.D, Senior Lecturer, School of Business and Informatics, University of Borås for giving us his valuable time and proper guidelines about our thesis.

Thanks to Dr. Anders Hjalmarsson, PhD, Thesis Course coordinator for coordinating our thesis. Also thanks to Dr. Anders Gidenstam, Ph.D, for supporting us to improve our thesis.

We are thankful to the University of Borås for giving us the opportunity to use LAB resources during the thesis. We are also grateful to the other groups and our friends for co-operating with us by sharing information.

Finally, we would like to show our greatest admiration to our respected parents who always supported and encouraged us in all aspect of life to progress, cautiously.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

1.1 Background

In the past, people normally used sequential computers except super computers. All of the programs execute sequentially, in serial computers and the performance of serial computer was not so good. Gradually the manufacturers had tried to improve the performance of those serial computers. Now-a-days, parallel architecture computers like different multi core computers are available to our hand. On the other hand, the workloads are increasing day by day. People are trying to solve more complex problems using computers and computer programs for the benefit of the world. For example planetary movement, weather and ocean patterns, rush hour traffic, building a space shuttle or ordering hamburger at the drive through. In present, people using the parallel computer or parallel application in different areas. Historically, parallel computing has been used to model difficult scientific and engineering problems found in different areas in the real world. For example Bioscience, Biotechnology, Genetics, Atmosphere, Earth, Environment, Chemistry, Molecular Sciences, Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics, Mechanical Engineering - from prosthetics to spacecraft, Electrical Engineering, Circuit Design, Microelectronics, Computer Science, Mathematics etc.

Today, parallel computing is being used in the commercial sectors for better services to save time and increase the speed for processing of large amounts of data in sophisticated ways. For example Databases, Data mining, Medical imaging and diagnosis, Oil exploration, Web search engines, Web based business services, Web search engines, Web based business services, Financial and Economic modeling, Advanced Graphics and Virtual reality, particularly in the entertainment industry and multi-media technologies etc.

So, we should know parallel processing. Parallel processing is a way of dividing a large complex task among multiple processing units which operates simultaneously for achieving a common goal. This is a strategy by which we can do any complex task very quickly. The expected result is faster completion with the comparison of sequential processing. In the early days the computer program was serial, and it was a set of sequential instructions. The main purpose of parallelism is to decrease the execution time by maximum uses of CPU resources. In sequential processing one instruction can execute at a time and all other instruction execute one after another. On the other hand Parallel processing large instructions are decomposing in different parts and distributing the different parts to different processors for doing the small task simultaneously. It increases efficiency, speed and performance. We use the parallel execution for executing independent tasks in parallel, overlap the execution of dependent tasks, complete a job in less time, and increase usage of CPUs. The parallel computer is becoming a high performance application platform. However, it is generally recognized that designing applications for parallel computers is hard [1] [2] [3]. For developing parallel application we have different languages, tools and platforms. Every platform has a different kind of tools and APIs. Every platform has an identical pattern for parallel applications. From the above point of view we are using the resource of parallel architecture and evaluate the performance of parallel application in .NET 4.0 with visual C# programming language.

1.2 Problem Statement

Performance is the main considered factor in parallel applications. So, if we think about parallel application development in .NET then the final target should be better performance. There are so many terminologies of parallel computer architecture and developing parallel application. In this research we have tried to discuss briefly about those terminologies. This research involves finding the constructs and APIs of. NET 4.0 and making some simple parallel application using different parallel APIs for different problem. Perform experimental comparison of some parallel applications using parallel extensions with corresponding sequential versions. The study involves experiments using different numbers of CPU resources and workloads to scale programs. Also the study involves performance evaluation and identifying the factors for achieving good performance in parallelism.

1.3 purposes of the study

Gradually the uses of computer resources are increasing and we are using it for computing complex problems. In the past we had no parallel architecture, but now we had. Now we can use properly the parallel architecture applying the right pattern of parallel application. According to the modern demand we need to work with extreme large number of data within a minimum time frame. So understanding the parallel application performance, way of proper CPU usages and minimum time usages is the main purpose of our study. This study will help to know the factors for increasing performance of parallel application in parallel architecture.

Many developers are using different kinds of programming tools or platforms for developing parallel application. Describing right pattern of parallel application in .NET platform is another purpose of this study. The study will help to know about different APIs in .NET platform and also the usability of APIs in the right situation.

1.4 Research Questions

For doing research on a subject and achieving his/her goal the researcher should keep some question in his/her mind. According to the research purpose we have a main research question and several sub questions based on the main question.

> "What is the experimental performance of applications using parallel extensions with corresponding sequential versions in terms of different workload and different number of CPU resources used in some implementations using two types of parallel architecture?"

Sub questions:

a. Which parallel constructs and APIs we can use for making numbers of appropriate simple parallel applications?
b. Which performance issues involve for improving the performance of parallel application constructed by parallel extensions?
c. What improvements of CPU resource utilization can be made by parallel executions?

1.5 Target groups

In this study we are discussing about parallel computing, parallel computer architecture, parallel application APIs in .NET platform, parallel application performance etc. So, this research will help different categories of people in different areas. We can divide our target beneficiaries in two main categories.

a. Academia

   In academic arena students or researchers who want to study in a parallel application, can get help with the study. They can get a brief knowledge about parallel programming. If they want to do more research then the study will help them as a foundation.

b. Professional

   In the professional group the developers and developing firm who are working with parallel programming in C# .NET platform will be helpful by this study. Some organizations are working to research on large volume of data from the real world like population, weather forecasting, atmosphere, environments. This thesis paper can also be helpful for them.

1.6 Research strategies

A Research strategy is a systematic plan to reach the research goal quickly and efficiently. Research strategy refers to the plan that a researcher will pursue to execute an investigation to address the research questions. It specifies the source of data constraints that may hamper the research and how they will be addressed.

The research strategy is the general plan of the researcher how they will go about answering the research questions which has been set by the researcher. It contains the clear ideas and objectives, derived from research questions, and also identify the sources from which researcher be determined to collect data and consider the constraints that the researcher will predictably have the success of accessing data, time, location and money, ethical issue [49].

In our research the main goals are performance measurement of parallel applications with the traditional sequential execution, find out the factors of achieving performance in parallel applications in .NET Framework 4 also find out APIs for making parallel application in .NET Framework 4. For achieving our desired goal there are some plans and steps towards the target. These steps are as follows.

- Study literature in depth on Parallelism, Parallel Programming and Parallel Computer Architecture.
- Find different patterns and APIs in .NET Framework 4.0
- Find which API/pattern is better for which situation
- Study existing parallel application in .NET Framework 4.
- Create new sample sequential and parallel programs with timing code for measuring performance.
- Execute the code and evaluate the output.
- Evaluate performance from a different point of views
- Find the issues of parallel application performance

The following Figure-1 illustrates the strategy of our research.

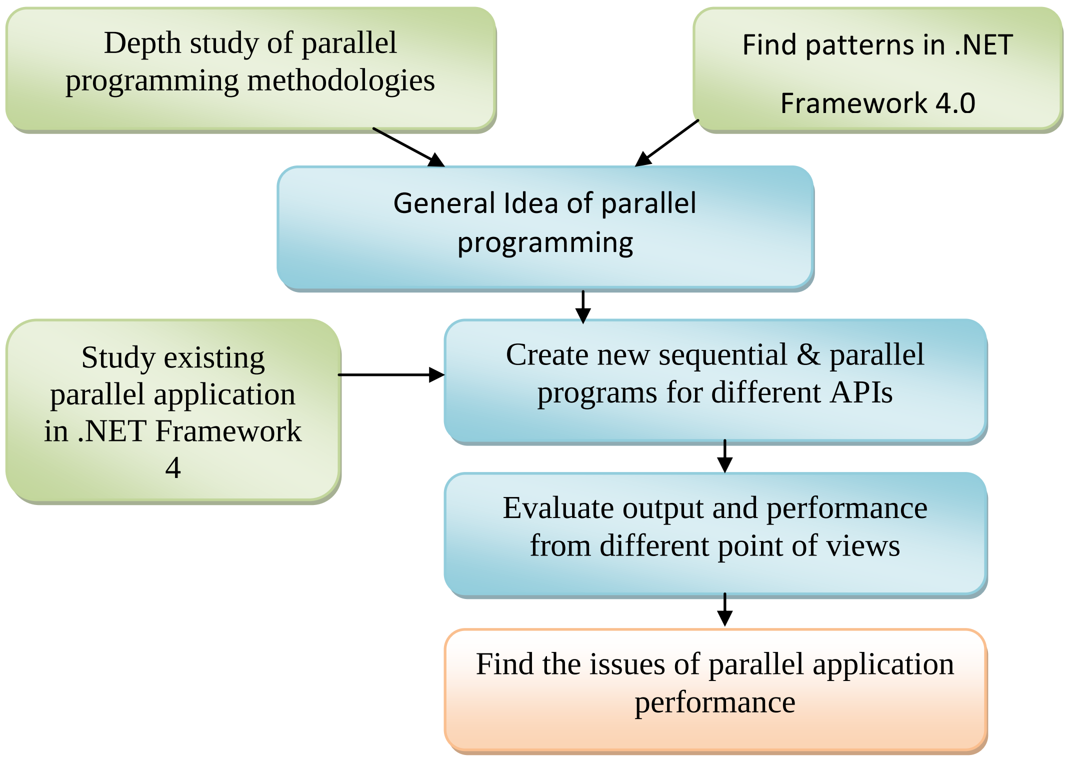

Depth study of parallel programming methodologies
Find patterns in .NET Framework 4.0
General Idea of parallel programming
Study existing parallel application in .NET Framework 4
Create new sequential & parallel programs for different APIs
Evaluate output and performance from different point of views
Find the issues of parallel application performance

Figure 1: Research Strategy.

2 PARALLEL PROGRAM DESIGNING METHODOLOGY

2.1 Introductions of Parallel Programming

In general the sequential execution is used for programming. In sequential programming one instruction is executed at a time and it uses one processor. Sequential programming has been written for serial computers. In sequential programming, processing speed depends on the hardware that how fast it can process the data. On the other hand, Parallel programming comes to overcoming the processing speed of sequential programming. For a complex task it decomposes an algorithm or data into different parts and divides those tasks into multiple computers and completes the tasks on multiple computers simultaneously.

2.2 Advantages of parallel Programming approach

The limitations of sequential program are execution time, speed and performance. For program execution in a sequential way it takes more time than parallel approach. Because it executes one instruction at a time and all instruction executes one after another. On the other hand in parallel programming approach multiple instructions are executed concurrently using multiple processing units. So it overcomes the limitations of sequential programming approach by improving processing speed of execution.

2.3 Why Parallel Programming is Difficult?

Generally parallel programming is seemed as an activity for people who are going to research or use the latest technology. It is very difficult because software engineers run away from parallel programming whereas modern general functions computer processors are of multi cores and the number of cores are increasing day by day. But these processors are not utilized properly. There are several difficulties of parallel programming as follows:

2.3.1 Managing complex data dependency is Non-Trivial

For parallel programming design, it is very important to well managed data dependency. For an example:

A1.m=20+10   A2.n=40-16   A3.q=m*n

In the above statements we can see that A1 and A2 can be executed in parallel in different cores but A3 will be executed only after m and n are available for the core executing statement A3. So in that case we need proper ordered data for execution [6].

2.3.2 Managing data access conflicts without full access control

Main memory of a system is shared across multiple cores. It is very useful if you want to maintain only particular data copy for multiple cores.

With the following statement we can see that a core might be incrementing a variable sequentially.

```
D1.p=q
D2.p+=1
D3.q=p
```

Since the memory is shared across multiple cores, q could be updated by another core after statement D1 and before the statement D3 are executed. The update is effectively lost when statement D3 is then executed writing value of p into q.

2.3.3 Managing multiple core executions driving one insane

Present state of parallel program is normally being executed a single program by multiple cores in parallel, i.e. every core executes a different copy of the same program. Every core performs different range of computations a data based on their IDs.

For simulating parallel program we have to make sure that the executions of different number of cores and also ensure that every core will execute properly even there is no data for some cores to compute.

2.3.4 Lack of Parallel library for general use

Programming becomes more manageable and simpler when we can use reusable libraries for our task. And most of the complex task is encapsulated inside software libraries. Parallel programming will be much easier if we use a parallel library as much as possible like parallel sorting, parallel binary tree search, parallel set, concurrent hashing etc. in our application. But there are not enough reusable parallel libraries. The .NET 4.0 has a rich parallel library for making parallel application. Such as Parallel LINQ or PLINQ, The Parallel class, The task parallelism constructs, The concurrent collections, SpinLock and SpinWait etc. Which are helping to data parallelism, Task parallelism and low level parallel constructs.

2.3.5 Interacting With Hardware

The area of Hardware interaction is usually the operating system, the compiler, libraries, or other software environment infrastructures. On the contrary, developers working with novel hardware features and method will often require working directly with such hardware. Also, through access to the hardware can be necessary when squeezing the last drop of performance out of a given system. For that reason, the developer may require modifying or configuring the function to the cache geometry, system topology, or interrelated procedure of the target hardware. In some cases, hardware may be well thought-out to be a resource which may be subject to partition or access control [6].

2.3.6 Legacy software is often used as the entry point to parallelization

Building software from scratch is very expensive. It is intelligent to make a parallel version based on the existing serial version. But, the software construction of the serial version might not appropriate for making a parallel version.

2.4 Parallel Computer Architecture

For improving performance of parallel programming there are several different methods. Flynn in 1972 [25] introduced four categories for the relationship of programming instruction to program data. These four Systems architectures are SISD, SIMD, MISD and MIMD. System architecture helps to understand parallelize coding methods and performance.

SISD: Single Instruction Single Data stream. This is not a parallel computer. In this model each instruction is used for every clock cycle and each instruction operates by single data element. Also there is a limitation of using instruction in a certain time period. The Figure-2 in the next page shows the SISD model.

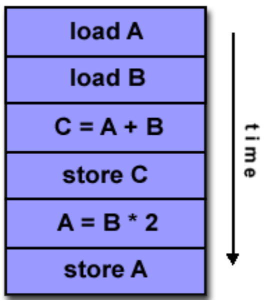


Figure 2: Single Instruction Single Data stream [5]

SIMD: Single Instruction Multiple Data stream. It is like SISD, but in this model one instruction can operate more than one data element. Each processing unit can execute multiple data, but in a given clock period the same instruction will be executed by all processing units. SIMD instruction and execution units are good for graphic processors in modern computers such as graphic/image processing. The following Figure-3 represents the SIMD model.

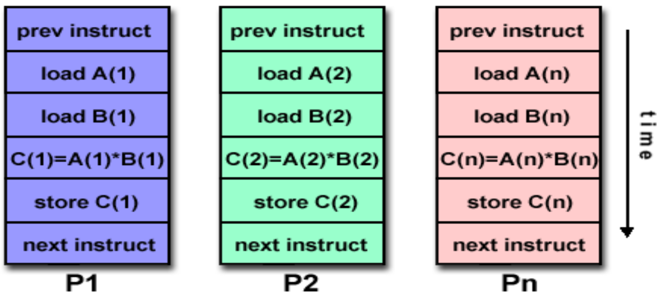


Figure 3: Single Instruction Multiple Data stream [5]

MISD: Multiple Instruction Single Data stream [26]. In this model multiple processing units joined with singe data stream, but every processing unit independently operates data via independent instruction stream. The Figure-4 in the next page represents the MISD model.

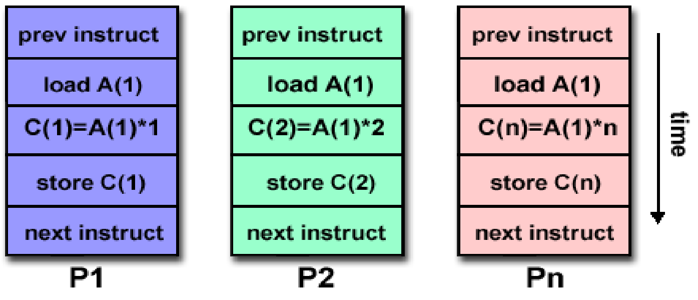


Figure 4: Multiple Instructions Single Data stream [5]

MIMD: Multiple Instruction multiple Data stream. Most of modern computers with multiple processors fall in this category. In this model multiple processors can operate multiple data streams and every processor can execute different instructions simultaneously. There can be deterministic or non-deterministic and synchronous or asynchronous instruction execution such as multi core PC and current super computer. The following Figure-5 represents the MIMD model.

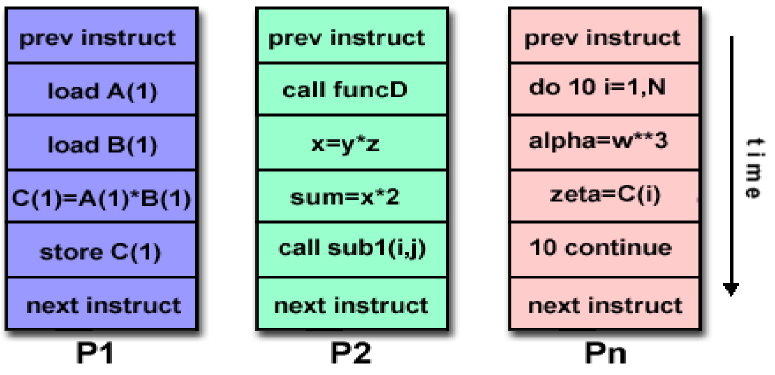


Figure 5: Multiple Instructions Multiple Data stream [5]

2.5 Memory Architecture

2.5.1 Shared Memory

In parallel computer architecture all processors can access all memory as a single global address space. Same memory resource shared by multiple processors during independent operation. It is visible to all the processors if any change in memory location affected by one processor. Based on memory access times shared memory machine can be divided into two main classes. These are UMA and NUMA. The figure-6 and figure-7 indicates the UMA and NUMA.

Uniform Memory Access (UMA):

- It is most commonly symbolized today with symmetric multiprocessor machines.
- Identical processors.
- Equal access and equal memory access time.
- Very often it is described as CC-UMA-Cache Coherent UMA. Cache Coherent means in a shared memory if one processor update a location then all other processors knows about that update.

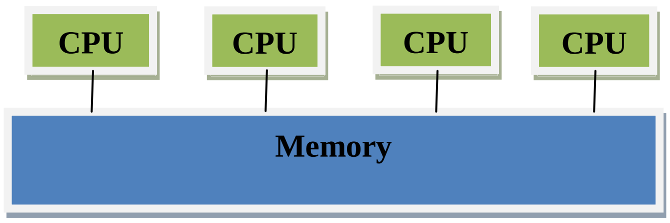


Figure 6: Shared Memory (UMA) [5]

Non-Uniform Memory Access

- Generally creates a physical link between two or more Symmetric Multiprocessors (SMPs) [5].
- In NUMA one SMP can straightforwardly access another SMPs memory.
- There is not equal access time for all processors to all memories.
- Memory access is slower across the link.
- It may be also called CC-NUMA-Cache Coherent NUMA if cache coherency is maintained.

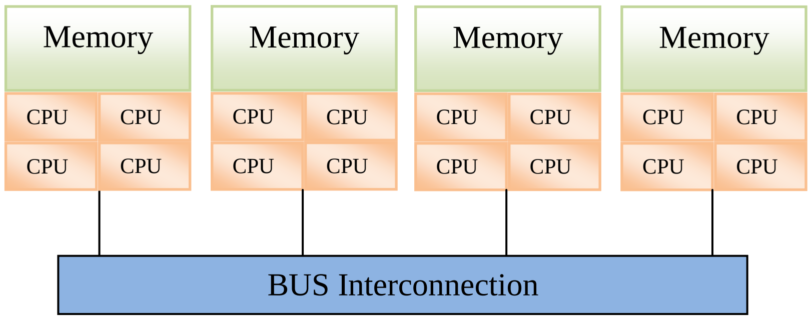


Figure 7: Shared Memory (NUMA)[5]

Advantages of shared memory

- Data sharing is very fast between tasks and uniform due to the proximity of memory to CPUs.
- Memory sharing as global address space provides a user-friendly programming perspective to memory.

Disadvantages of shared memory

- The main disadvantage is the lack of scalability between memory and CPUs.
- It is totally the responsibility of programmer for synchronization constructs which ensure the right use of global memory.
- It is expensive and difficult to design and produce shared memory machine with multi core processors.

2.5.2 Distributed memory

Distributed memory system is similar to shared memory system and it is varying widely but shares a common characteristic. To connect with inter processor memory it requires a communication network. Every processor has their own local memory and one processor memory address do not map with another processor memory address, so the concept of global address space across all processors is not available. It operates independently because every processor has its own local memory. There is no changing effect to another processor if any processor changes its local memory. So, there is no concept of cache coherency. It's totally programmer responsibility to explicitly define how and when data is communicated if one processor needs to communicate with another processor. The Figure-8 shows the distributed memory architecture.

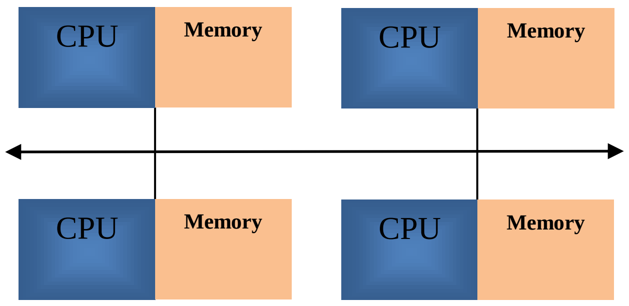


Figure 8: Distributed memory architecture [9]

Advantages of distributed memory architecture

- In distributed memory architecture processors number are scalable with memory. In this architecture the number of processors is increased in proportion with memory size.
- Without any interference each processor can rapidly access its own memory.
- It is cost effective.

Disadvantages of distributed memory architecture

- Programmer is responsible for data communication between processors and also many of its particular linking between processors.
- For memory organization based on its global memory is very difficult to map with existing data structure.

2.6 Parallel programming models

There are several models for parallel programming. Such as Shared Memory model, Message Passing model, Threads model, Data Parallel model, Hybrid Model. Any of those model can be implemented in parallel programming depends on the scope or demand of problem to be solved. According to the problem and algorithm sometimes two or more models need to introduce in same program. Generally thread model is used for making multiple execution paths of a single program. Sometimes beside the thread model the shared memory model is needed for access shared memory location from multiple threads. Sometimes data parallel model is needed for partitioning a large set of data.

2.6.1 Shared memory model

In shared memory model multiple processes or multiple threads share common memory space. The shared-memory systems allow multiple processors or threads to simultaneously read and write the same memory locations using some read and write instructions [64]. For controlling the access of shared memory various mechanisms are used like locks or semaphores [5]. The main disadvantage of shared memory model is performance and it is more difficult to understand and manage data area. The figure-9 illustrates the Shared memory model.

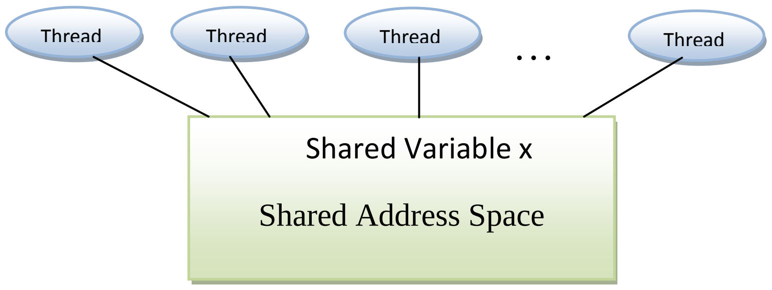


Figure 9: Shared memory model [9]

Implementations

- The native compilers of shared memory platforms translate user program variables into actual memory addresses.
- There are no platforms implementations of distributed memory that are currently exist.

2.6.2 Threads model

In parallel programming threads model a single process can have several parallel execution paths. Every execution path can complete execution independently. There is the detail description about threading in section 2.9 of this paper.

2.6.3 Message passing model

The following characteristics are demonstrated by the message passing model [8]:

- During computation in their own local memory a set of tasks is used. Same physical mechanism and or across random number of machines can reside multiple task.
- Data are exchanged by task by sending and receiving messages through communication.
- To perform data transfer by each process generally it requires cooperative operation.

Implementation

- Message passing implementations are usually covered subroutines of a library which are embedded in source code. The programmer is mainly responsible for determining parallelism.
- For task communication normally do not use MPI [24] implementations in a shared memory architecture. For performance factors they use shared memory.

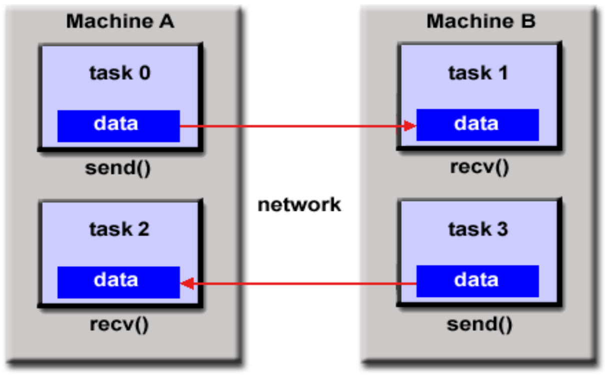


Figure 10: Message passing Model [5]

2.6.3 Data Parallel Model

The following characteristics are demonstrated by the data parallel model:

- On a data set the majority of parallel work spotlighted on performing operations. For an example an array or cube, the data set is normally structured into an ordinary structure.

- Together a set of task work on the similar data formation, though every task works on the same data format on a different partition.
- Through global memory all tasks may have access to the data structure in shared memory. On the other hand, the data structure is divide and existing as "chunks" in the local memory of every task in distributed memory architecture.

The following figure-11 shows the data parallel model.

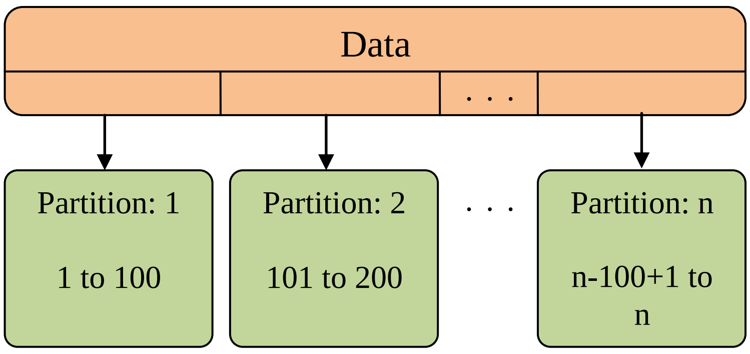


Figure 11: Data parallel model [5]

Implementation

- Write a program with data parallel constructs is normally accomplished by data parallel model.
- To specify the alignment and distribution of data are allowed by compiler directives for the programmer. And for the most general parallel platforms FORTRAN [5] implementations are available.
- Programmers are not able to see all message passing that is done.

2.6.4 Hybrid Model

- The Hybrid model is a combination of two or more other parallel programming models that are described in this section.
- A general example of hybrid model is message passing model is combined with shared memory model or the threads model.
- Another common example of a hybrid model is combining data parallel with message passing. As mentioned in the data parallel model section previously, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer [5].

2.7 Steps for Creating a Parallel Program

In general, sequential program construction is not so complex. But making a parallel program is complex and involves many factors and dependency. In a sequential program there is no extra headache for partitioning the task or data. But in a parallel program the whole work is partitioned in several subparts and different processors are responsible for executing one or more subparts. And different part of whole work execute concurrently.

There are three important concepts for creating a parallel program such as task, processes and processors [48].

A task is the smallest unit of work done by the program. A whole work can be divided into several tasks and different tasks can execute concurrently in a parallel program. On the other hand a process is an arbitrary entity. There are several cooperative processes can be exist in a parallel program, each process program subset of task. Finally every process executes its task on physical processors.
For making a parallel program, there are the four steps [48].

a) Decomposition
Decomposition is a process to break whole programs into different task. There is more about this in the next section.

b) Assignment
The assignment operation is responsible for assigning or distributing different tasks among processes. The assignment operation mentions the mechanism of task distribution. It task are distributed properly with proper load balancing and less inter process communication the overall performance of parallel program will increase.

c) Orchestration
In a parallel program, all the process is synchronized with each other for exchanging the data. The orchestration mechanism mainly depends on the programming model.

d) Mapping
The mapping step is mainly responsible for mapping all processes to processors. The operating system is mainly responsible for this mapping operation.

The following figure-12 illustrates different steps of creating parallel program.

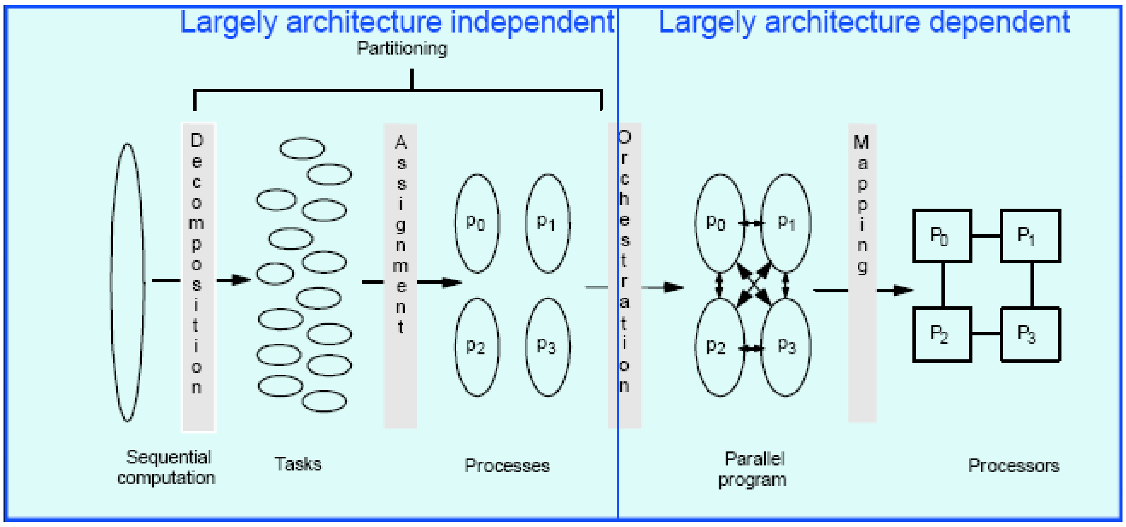


Figure 12: Step of creating a Parallel program [48]

According to the above figure we can see that the first step is decomposition or partitioning. Before partition of a sequential program we have to identify the different parts of the program which can be executed concurrently. After that the whole program can be partitioned in several tasks. Several tasks are distributed to different processes by assignment operation. In a parallel program different processes need to communicate and synchronize to each other for executing all the tasks. The communication and synchronization model can be defined by orchestration operation. And finally using the mapping operation all processes maps to the available processors for execution.

2.8 Decomposition

Decomposition is a process to break computing into different tasks and to assign among processors [48]. Decomposition in a parallel computer increased computing power. There are three methods of decomposing parallel computing into smaller task. These are functional decomposition, Producer/Consumer decomposition, Data Decomposition, or a combination of both [4].

2.8.1 Functional decomposition

In functional decomposition the problem is divided into different tasks and distributed to the multiple processors for simultaneous execution. Functional decomposition is better if the structure is not static or fixed optimized number of calculations for performing.

2.8.2 Producer/Consumer decomposition

Producer/Consumer decomposition is a type of functional decomposition where input of second thread is the result of first thread. In Producer/Consumer scenarios often allow several producers and/or consumers.

2.8.3 Data decomposition

It is data level parallelism. In data decomposition task are broken down by the working data instead of the characteristics of the task. Normally many threads are performing the same work which is broken down via data decomposition on the same program but different data items.

2.9 Threading

In an execution, a thread is a smallest unit of processing [58]. In any multithreading application there are several threads and the operating system is mainly responsible for scheduling them. The figure-13 illustrates the simple multithreading mechanism. In the implementation points of view threads and tasks are similar but thread and processes differs. A thread exists inside a process. In a process multiple threads can exist and they can share resources [11] [58].

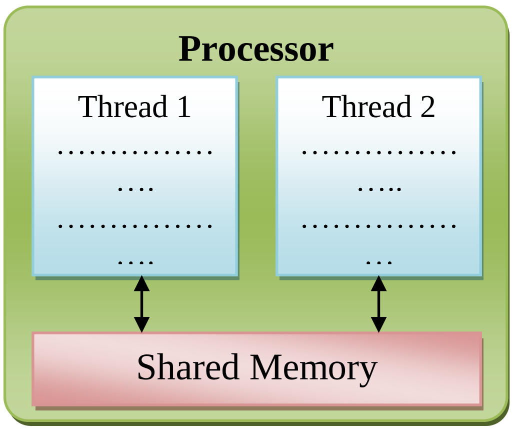


Figure 13: Simple Threading Model

In a multithreading application all threads are managed by a thread scheduler. A thread scheduler ensures CPU time allocation, waiting and blocking states of different thread. If you think about multithreading in a single processor then it is possible by Time Division Multiplexing. In general it seems as parallel but originally it is not. The processor switches between threads. But in multi core system multithreading [46] is implemented with genuine concurrency [11] and different threads can run in parallel in different core.
A main program runs in a single thread (main thread). The thread creates new threads [11]. The following figure-14 shows how the worker thread act under the main thread.

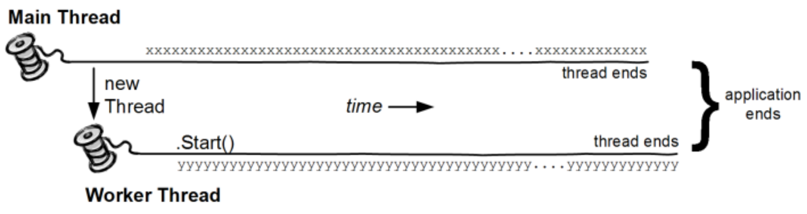

Main Thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxxxxxxxxx
thread ends
new Thread
time
application ends
.Start()
thread ends
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyyyyyyy
Worker Thread

Figure 14: Threading concept in application (from [11])

In a whole life cycle a thread has different states. In Programming Framework there are some methods and properties for identifying the different states of thread. All states of a thread can be divided mainly in layers [11]. In figure-15 on the next page shows the different states and layers.

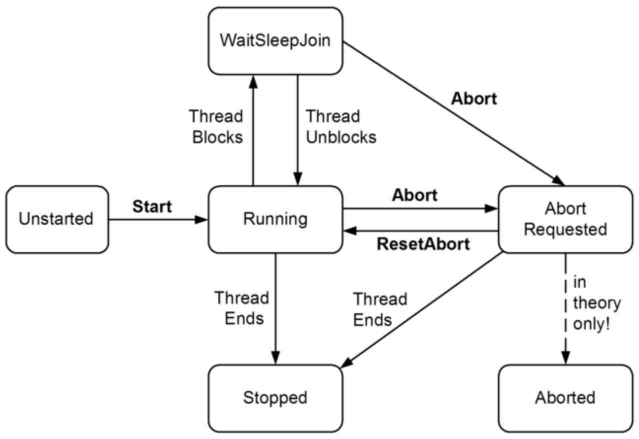


Figure 15: Different states and layers of thread

2.9.1 Threads per Processor

The number of threads per processor depends on the architecture. In general and from the different experimental study we can get that, in an old single core system there are 2 threads (1 thread per CPU x an arbitrary multiplier of 2). In modern multi core system generally supports 2 threads per core. In one experiment with Intel i7 quad core system 16 threads has been found (4 CPUs x 2 logical threads per CPU x the arbitrary multiplier of 2) [12]. Also some software thread can be possible using time sharing.

2.9.2 Uses of Multithreading

Multithreading has many uses. The followings are most common.

- Making a high functional user interface where the thread will work on the time consuming task and the interface will be free for keyboard and mouse event.

- Making such application where a thread will wait for getting response from other system or hardware and other thread will perform other reaming task.
- Making parallel application in which large and complex problem with large number of data can be handled by a different thread in multi core or multiprocessor computer.
- For making such kind of server application where the server will handle large number of client requests simultaneously.

2.10 Communications

Communication refers to the information exchange between threads as well as processes in a parallel application [59]. The majority number of parallel programming application requires task to share data with each other because all of those applications are not reasonably so simple. There are some protocols for sharing and synchronizing data during that communication. In a single application different processes co-operate with each other by accessing shared resources. The operating system manages the communication by some synchronization mechanisms. Sometimes the communication done by accessing shared memory space. In a client-server scheme can follow producer-consumer mechanism. In client server scheme the clients produce requests and servers consume them. In client server approach message passing mechanism is more applicable than the shared memory concept [59].

3 PARALLEL APPLICATION PATTERNS IN .NET FRAMEWORK 4.0

In different programming languages there are different ways of programming. In software development, the different ways or patterns help developers to establish a direction in a wide range of problem areas. Patterns mean successful, repeated and common solutions which developers apply in particular architectural and programming domains. Design patterns covers architectural structure or methodologies, coding patterns and building blocks also emerge, representing typical ways of implementing a specific mechanism [14]. When developers are trying to find better solutions, sometimes they succeed and sometimes they fail. After many times attempts for the solution it possible to find good patterns. Also in new problem areas finding patterns takes more time.

In an age of high-performance computing there were some experiences for supercomputer and clusters in the last few decades. But recently such experiences can be possible in our personal computer area. Manufacturers are building the multi-core machine as a personal computer for daily uses. So, computation is now in front of parallel architecture and large number of heavy workload. And it is very important to make efficient parallel application to ensure that all applications can build with much parallelism and scalability. Design parallel constructs and APIs for parallel computing help to add structure and discipline to the process of concurrent software development [16, 20, 21, 22].

> *"In general, a 'multi-core' chip refers to eight or fewer homogeneous cores in one microprocessor package, whereas a 'manycore' chip has more than eight possibly heterogeneous cores in one microprocessor package. In a manycore system, all cores share the resources and services, including memory and disk access, provided by the operating system."*
> *–The Manycore Shift, (Microsoft Corp., 2007)*

For the different reason (already discussed in previous sections) the parallel programming is difficult for the developers. In parallel programming many issues are involved like decomposition, synchronization, load balancing, two-step dance, lock, deadlock etc. Which are not in sequential programming? So it is a great challenge for the developer to understand the ways of parallel programming, avoid such difficult issues and make quality patterns.

The .NET Framework 4.0 has a large collection of API to handle common needs for parallel programming. The .NET Framework 4.0 helps developers make parallel applications for solving large and complex problems in parallel architecture like multi-core and many-core machines. The new key features of .NET Framework 4.0 support for parallel patterns [14]. In this chapter the following discussion will provide depth knowledge about common parallel constructs and APIs for parallel programming in .NET Framework 4.0. This following discussion covers language-based support of Visual C# in Visual Studio 2010 for different key parallel patterns.

3.1 PFX (Parallel Framework Extensions) Concepts

The Parallel Framework Extensions are called PFX. Previously single-threaded code was being used in single core machine. But in multi-core machine the single-threaded code will not run faster automatically. In most server applications multi-thread has been used. Each thread handles different client request. Now multi-thread pattern are implemented in everywhere. So you should do the followings in your code [11]:

1. Partition it into small chunks.
2. Execute those chunks in parallel via multithreading.
3. Collect the results as they become available, in a thread-safe and performance manner.

But at the time of partitioning and collecting another problem occurred like locking and thread-safety. Because, communication is needed when many threads work on the same data set at a time. There are two methods for partitioning work for threads [25]

1. Data parallelism
   When a large number of data values will be operated by a set of tasks then we can parallelize by multiple threads. Each thread will perform (same) set of task of subset data. This approach called data parallelism [13]. In this approach we are partitioning data between different threads.

2. Task parallelism
   On the other hand tasks are partitioned in different threads. In Task parallelism [13] or Functional parallelism distributing the execution process or work in different computing nodes or threads.

For the highly parallel hardware the data parallelism is better. The data parallelism has less shared data. And the data parallelism is also structured parallelism [11] means that the parallel work unit starts and finish in the same place in a program. On the other hand the task parallelism is unstructured.

3.2 PFX Components

In PFX (Parallel Framework Extensions) there are mainly two functionality layers [11]. The top layer consists of two *structured data parallelism[13]* APIs: **PLINQ** and the **Parallel** class. And the bellow of top layer consists of *unstructured task parallelism classes* and some additional constructs to help with low-level parallel programming activities. The following figure-16 shows the components of PFX.

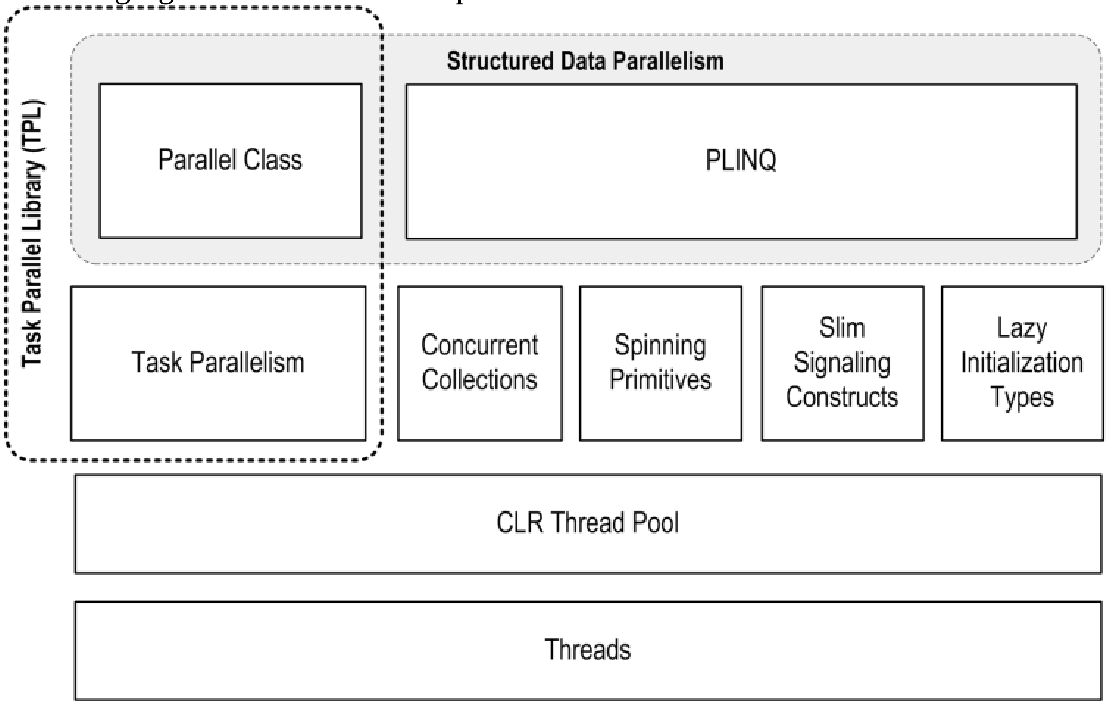

Task Parallel Library (TPL)
Structured Data Parallelism
Parallel Class
PLINQ
Task Parallelism
Concurrent Collections
Spinning Primitives
Slim Signaling Constructs
Lazy Initialization Types
CLR Thread Pool
Threads

Figure 16: Components of PFX [11]

The following APIs are collectively known as PFX (Parallel Framework Extensions). Those multithreading APIs [46] are used in .NET Framework 4.0.

- Parallel LINQ or PLINQ
- The Parallel class
- The task parallelism constructs
- The concurrent collections
- SpinLock and SpinWait

The task parallelism constructs are called the Task Parallel Library or TPL. The .NET Framework 4.0 also has some lower-level threading constructs:

- The low-latency signaling constructs (SemaphoreSlim, ManualResetEventSlim, CountdownEvent andBarrier)
- Cancellation tokens for cooperative cancellation
- The lazy initialization classes
- ThreadLocal<T>

The PLINQ automates all the steps. PLINQ includes partitioning into task, execution and collecting the result. Parallel classes are responsible for partitioning and execution rather than collecting the result. In this case you have to collect the result yourself. In the task parallelism constructs, you have to partition the work and collect the result yourself. The following table shows the functionality. The concurrent collections and spinning primitives help low-level parallel programming activities. In PFX the concurrent collections and spinning primitives are not only important for the today's parallel architecture but also future architecture.

	Partitions work	**Collates results**
PLINQ	Yes	Yes
The **Parallel** class	Yes	No
PFX's task parallelism	No	No

Table 1: Characteristics of PFX's Component

3.3 .NET Framework 4.0 & Parallel Computing

Nowadays people are using personal computers and workstations with multi core (CPUs). Unlike the previous used single core machines the multi core machines are capable to execute simultaneously using multiple threads. The multi core featured computers are increasing rapidly. Proper utilization of modern multi core features we have to introduce the parallelism in our application. This parallelism technique helps us to distribute work across multiple processors. Visual Studio 2010 and the .NET Framework 4 enhance the past low level threading support for parallel programming by providing a new runtime, new class library types, and new diagnostic tools [33]. In .NET Framework 4 provides so many features help us to develop parallel application very easily. It helps us to write scalable, fine-grained parallel application for executing on those multi core computers. The following figure-17 shows a high-level overview of the parallel programming architecture of the .NET Framework 4 [33].

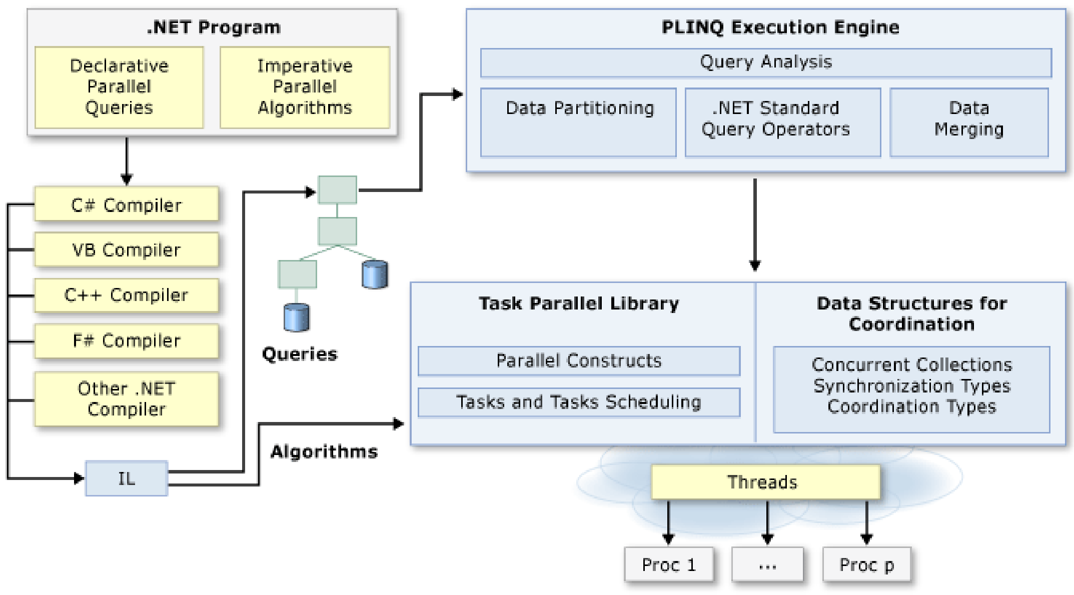


Figure 17: High-level overview of the parallel programming architecture (in the .NET Framework 4([33])

3.4 Parallel Looping constructs Using Static Partitioning

This is very primitive pattern for creating a parallel application. Here you can create a thread whatever you want depending on the existing number of cores on your machine. The thread Vs processor already described in the previous chapter.

Here is an example of making a simple program for creating one-thread-per core. In that case at first we have to identify that how many cores have in our machine. In .NET Framework there is a class *System. Environment* and its property named *ProcessorCount* will helps to retrieve the number of logical processors you have. When you know the number of processors and number of threads you can create threads and assign work to each thread. This implementation will help to gather a basic idea of partitioning work and implement parallel programs. In this implementation each core is used for executing specific partition. Here is an example.

```
public static void StaticThread()
        {
        // Set the total work size
int size=n;
        // Determine the number of iterations to be processed, the number // of cores to
        use, and the approximate number of iterations to // process in each thread.

int numProcs = Environment.ProcessorCount;
int range = size / numProcs;
        var threads = new List<Thread>(numProcs);
        for ( int p = 0; p < numProcs; p++ )
                {
        int start = p * range;
        int end = (p == numProcs - 1) ? size : start + range;
        threads.Add(new Thread(() =>
                {
        for (int i = start; i < end; i++)
                                //Process statement
                }));
                }

        foreach (var thread in threads) thread.Start();
        foreach (var thread in threads) thread.Join();

}
```

In the above example we have used one thread per core. Also we can create two threads per core. There are some positive and negative impacts of this implementation.

- The positive impact is that we can assign thread resource for specific loop partition. In that case the operating system is responsible to schedule the threads across the hardware in a suitable way.
- One of the negative impacts is the cost of thread. In general in .NET Framework 4, one thread needs approximately a megabyte of stack space [14]. If we assign a small operation to a thread then the space will not use efficiently.

- Another negative impact is that, repeatedly it creates a new thread and turn down is an extra load for the processor. If we assign a small loop and little work per loop then it might not be better in performance with respect to a sequential process.
- Oversubscription is another negative impact and a major performance issue. If we assign many threads over the limited hardware resources then the system forces to the operating system to spend more time to context switching between components. This context switching has a very negative effect on caching system on the machine.

3.5 Thread Pool

In previous discussion there were some negative impacts and cost factors like oversubscription, context switching, caching, accessing data from main memory etc. We can reduce those kinds of cost factors by creating the pools of threads [13] [60]. In this approach the system manages all the threads into a pool. Several tasks wait in a queue. When a thread in a pool completes current task it requests another task from the queue. After complete a task and start a new task the thread returns to the waiting thread's queue. The Thread Pool ensures the maximum number of threads. When all threads of pool are busy the tasks wait in queue [60]. The Thread Pool [13] manages the cost of life cycle of all thread. The thread pool ensures an upper-limit of thread number, limit of total memory consumed and oversubscription.

The .NET Framework 4.0 has the **System.Threading.ThreadPool** class which provides such kind of thread pool. The .NET Framework maintains a pool of threads. The static **QueueUserWorkItem** method is mainly used for making this. The following example [14] demonstrates the simple implementation of ThreadPool.

```
public static void StaticThreadPool()
   {
      // Determine the number of iterations to be processed, the
      // number of cores to use, and the approximate number of
      // iterations to process in each thread.
     int size = n;
     int numProcs = Environment.ProcessorCount;
     int range = size / numProcs;
     // it will track the number of threads remaining to complete.
     int remaining = numProcs;
     using (ManualResetEvent mre = new ManualResetEvent(false))
     {
        for (int p = 0; p < numProcs; p++)
        {
           int start = p * range;
           int end = (p == numProcs - 1) ? size : start + range;
           ThreadPool.QueueUserWorkItem(delegate {
              for (int i = start; i < end; i++)
                 //peocess ststement
              if (Interlocked.Decrement(ref remaining) == 0)
                                   mre.Set();

              });
        }
```

```
                // Wait for all threads to complete.
                mre.WaitOne();
            }
             }
```

In the above example, the other operations like to retrieve the number of processor and partitioning is like as previous one. In the above example the **ManualResetEvent** allow threads to communicate with them [28].

3.6 Dynamic Partitioning

In static partitioning less synchronization is needed. But we think about the load balancing between processors then we have to think about dynamic partitioning. In dynamic partitioning the system can adapt changing workload quickly. So in static partitioning there is less synchronization but in dynamic partitioning much load-balancing is possible [14]. For a simple and particular problem the static partitioning may be suitable. But in general, for minimizing the overhead and sufficient load balancing the dynamic partitioning is better. In .NET Framework 4 has different implementation patterns and APIs for such implementation.

3.7 Parallel.For

Here we will discuss about the **Parallel.For** which is the most common pattern in .NET Framework 4 for parallel programming. The **System.Threading.Tasks** namespace of .NET Framework 4 provides a static Parallel class. Before going to **Parallel.For** here we are showing a simple, single thread application:

```
for (int i = 0; i < 100; i++)
        {
                //statements in loop
        }
```

Now we can see a basic structure of Parallel.For which requires three arguments: the lower bound, the upper bound and the delegate to be invoked for each iteration.

```
Parallel.For(0, 100, delegate(int i)
   {
          Workstatement(i);
          //here program call a process of work
   }
   );
```

Here is another pattern of Parallel.For. This pattern shows how to use thread-local variables to store. This pattern also regains state in its separate task that is created by a Parallel.For. In this pattern thread-local data helps to avoid the overhead of synchronizing a large number of accesses to shared state. And finally write the final result once to the shared resource, or pass it to another method. The following example [62] illustrates the implementation of Parallel.For.

```
static void Main()
  {
  int[] nums = Enumerable.Range(0, 1000000).ToArray();
  long total = 0;
  // Use type parameter to make subtotal a long, not an int
  Parallel.For<long>(0, nums.Length, () => 0, (j, loop, subtotal
            ) =>
        {
           subtotal += nums[j];
           return subtotal;
        },
           (x) => Interlocked.Add(ref total, x)
        );
        Console.WriteLine("The total is {0}", total);
        Console.WriteLine("Press any key to exit");
       }
   }
```

By default, the Parallel.For uses the .NET Framework 4 ThreadPool as a work queue to execute the loop. Also the Parallel. For and its overload provides a lot of features:

- **Exception handling**: The Parallel. For has the ability for exception handling. If any iteration of parallel loop throws an exception then all of the threads in the loop attempt to stop processing as soon as possible. If any iteration is currently executing then it will not be stopped but new loop will not start. At the last of all unhandled exception will aggregate in an AggregateException instance. This exception type provides support for multiple "inner exceptions," whereas most .NET Framework exception types support only a single inner exception [29].
- **Breaking out of a loop early**: Like the break keyword in C#, Parallel. For has a feature of breaking out of a loop.
- **Long ranges**: The Parallel.For support Int32-based ranges to work also overload provide for working with Int64-based ranges.
- **Thread-local state**: Several overloads provide support for thread-local state.
- **Configuration options**: It is possible to control multiple aspects of a loop's execution.
- **Nested parallelism**: If there is a Parallel.For loop within another Parallel.For loop then the both loops coordinate with each other and share threading resources. Also there are two Parallel. For loops concurrently in the same pool then they can share threading resources in the same pool.
- **Dynamic thread counts**: Parallel.For is suitable for accommodating workloads in execution time. It is good to process thread based load rather than static assignment.
- **Efficient load balancing**: Parallel.For also supports load balancing. This helps to ensure quality of load balancing at the time of few numbers of iteration also many numbers of iterations.

3.8 PARALLEL.FOREACH

In for loop iteration work on a specific range or specific kind of data set which data set built by number which represents the range. But Parallel.Foreach in C# is a such kind of pattern which works on any data set. The structure of Parallel.Foreach is as follows [14]:

```
Parallel.ForEach(datalist, dataitem =>
                {
                // statements in iteration
        }
        );
```

Also we can use the Enumerable class from LINQ. The **IEnumerable<int>** represent the same range which use in **Parallel.Foreach**.

```
foreach(int i in Enumerable.Range(0, 10))
{
//statements for i.
}
```

Using this Parallel.Foreach we can achieve much more complex parallel patterns for unknown range of data set. The Parallel Foreach implementation concept is similar to Parallel For. Like previous partition the data and assign each partition in the thread for execution.

3.8.1 PROCESSING NON-INTEGRAL RANGES

In .NET Framework 4.0 the Parallel class provides overloads for working with ranges of Int32 and Int64 values [14]. The parallel For and Parallel **Foreach** can work on non integer value. Parallel.For has no overloads for **Node<T>**, its direct usage with **Parallel.ForEach**. To work with **Node<T>** we can make a method as follows [14], and then we can use it with **Parallel.ForEach**

```
public static IEnumerable<Node<T>> Iterate(Node<T> head)
        {
        for (Node<T> i = head; i != null; i = i.Next)
                {
                yield return i;
                }
        }

Parallel.ForEach(Iterate(head), i =>
{
// ... Process node i.
});
```

Here is a simple example using Parallel.ForEach for better understanding. This example downloads the webpage content from different websites. Here the example [63] shows the pattern of normal ForEach and the Parallel.ForEach.

```
static void Main(string[] args)
{
        string[] urls =  { "http://towardsnext.wordpress.com",
        "http://yahoo.com",
        "http://microsoft.com",
        "http://google.com",
        "http://aol.com",
        "http://rediff.com"
        };

// download content Using foreach
Console.WriteLine("Foreach");
foreach (string url in urls)
{
        WebClient client = new WebClient();
        client.DownloadString(url);
}

Console.WriteLine("Parallel Foreach");
Parallel.ForEach(urls, url =>
{
        WebClient client = new WebClient();
        client.DownloadString(url);
});
}
```

3.8.2 Breaking out of loops early

Like Parallel.For, the Parallel.ForEach supports some mechanism for breaking out for loop. Such as:

- **Planned exit**
  In both Parallel.For and Parallel.ForEach have two methods Stop and Break, and properties IsStopped and LowestBreakIteration for supporting the planned exit. The Parallel.For and Parallel.ForEach surpass a ParallelLoopState request to the body delegate. At the time of program execution if any iteration calls Stop then loop control logic will not start any new iterations. The ParallelLoopResult.IsCompleted helps to identify that the iteration is completed or not.
- **Unplanned exit**
  In Unplanned exit anytime it can be possible to terminate loop from outside. The System.Threading.CancellationToken in the .NET Framework 4 helps to make this cancellation. Overloads of all methods accept a ParallelOptions instance. CancellationToken property of ParallelOptions monitors the cancellation. When the loop finds the cancellation it stops launching new iteration and wait for other existing iteration [14].
- **Multiple Exit Strategies**
  Also the multiple exist strategies can be employed together. Because after applying exception handling some unhandled exception can occur.

3.9 PARALLEL.INVOKE

The Parallel.For and the Parallel.ForEach mainly support overloads for data parallelism in the System.Threading.Tasks.Parallel class [32]. On the other hand the **Parallel.Invoke** provides the task parallelism [13] in the Task Parallel Library. The **Parallel.Invoke**supports to execute concurrently any number of arbitrary statements. In such a way of operation a shared memory can be used as a data source. Here is a simple structure for using Parallel.Invoke [43]:

```
static void Main(string[] args)
{
   Parallel.Invoke(
                        ()=>F1,
                        ()=>F2,
                        ()=>F3,
                        ()=>F4);
}
public static void F1()
{
      //statement
}
public static void F2()
{
      //statement
}
public static void F3()
{
      //statement
}
public static void F4()
{
      //statement
}
```

The different methods: F1, F2, F3 and F4 will be assigned to different threads and execute it in parallel.

3.10 PLINQ

Language-Integrated Query (LINQ) was introduced in the .NET Framework version 3.0. It features a unified model for querying any System.Collections.IEnumerable or System.Collections.Generic.IEnumerable(Of T) data source in a type-safe manner [34]. Parallel implementation of the LINQ pattern is **Parallel LINQ** (PLINQ) [13]. If the sequential LINQ queries operate on IEnumerable or IEnumerable(Of T) data source with multiple execution then we can say it PLINQ [11]. In PLINQ the data source partitioned into different segment and all partition execute parallel by a different thread on a multiprocessor machine. The ultimate result is that the PLINQ ensures the maximum uses of CPU resource. PLINQ distribute the CPU work as a way that the all threads get a fair distribution [53].

Almost all of PLINQ's functionalities are exposed by the System.Linq.ParallelEnumerable class. In PLINQ It has been implemented all standard query operations of LINQ. And

System.Linq has the additional operation for parallelism. PLINQ is a combination the power of parallelism and simple, readable syntaxes of LINQ [11].

3.10.1 Degree of Parallelism

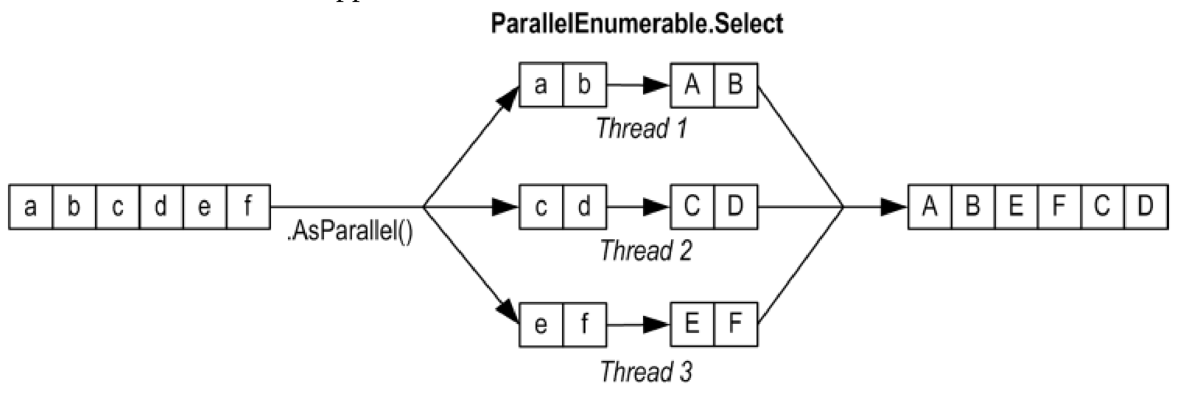
The PLINQ uses all available cores or CPU resources and show the significant better performance over LINQ and others. By default, PLINQ use all of the processors on the host computer up to a maximum of 64 [53]. By using WithDegreeOfParallelism(Of TSource) method it is possible to specify the number of processors for PLINQ [54]. This PLINQ gives the opportunity to share a certain amount of CPU time to other processes running on the computer.

Parallel LINQ in the .NET Framework 4 provides all .NET Framework query operators such as Select (projections), Where (filters), OrderBy (sorting), and a host of others. PLINQ also provides several additional operators such as AsParallel and ForAll. The AsParallel enable parallel processing of a LINQ-to-Objects query and ForAll support for Partitioning data like Parallel.For and Parallel.ForEach. Also the PLINQ has a great support for merging data after execution.

PLINQ can implement simply by calling AsParallel() on the input sequence after that continue the LINQ query as usual. The following example illustrates the calculation of prime numbers between 3 and 100,000.

```
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);
var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;
int[] primes = parallelQuery.ToArray();
```

The whole work is done by partitioning the input data in different groups then execute using different threads and finally collect all the results and combine in a single output. Figure-18 shows normal .AsParallel approach.



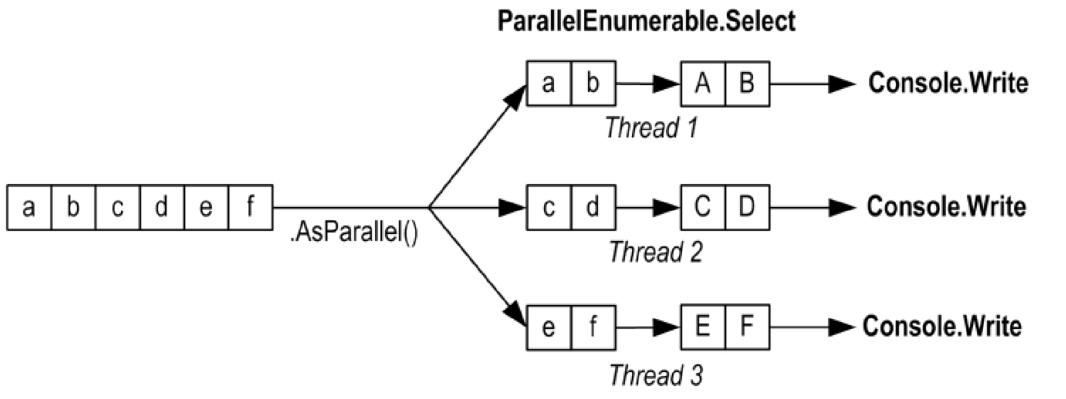
```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

Figure 18: PLINQ .AsParallel Approach (From [11])

3.10.2 Output-side optimization of PLINQ

In parallel programming, partitioning and merging is an overhead or extra cost of the processor and it is comparable to the sequential programming. If we want to make the parallel application then we can't avoid the partitioning. But sometime we need to avoid merging. The **ForAll** operator of PLINQ supports to avoid merging and executes a delegate for each output element. The figure-19 on the next page shows the approach with merging result.

```
List<InputData> inputData =”a, b, c, d, e, f”;
inputData.AsParallel().Select(i => new OutputData(i)).ForAll(o =>
        {
        ProcessOutput(o);
        });
```



```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write)
```

Figure 19: PLINQ withoutmerging result (From [11])

The following illustration (Figure-20) shows the difference between foreach and ForAll<TSource> with regards to query execution [34].

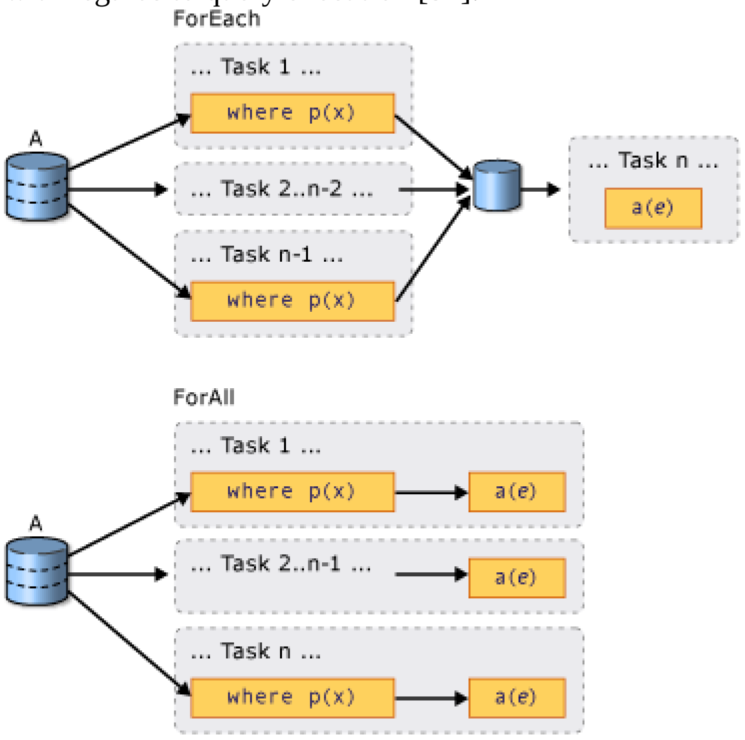


Figure 20: difference between Foreach and ForAll<TSource>

3.10.3 Input-side optimization of PLINQ

The general concept of the PLINQ is to take the total input data, partition it into different parts, and assign the different part to different available processing cores. The different parts will processed by different cores and create an output. And finally merge or aggregate the result or execute another operation on the final result. The following figure-21 shows the PLINQ concept.

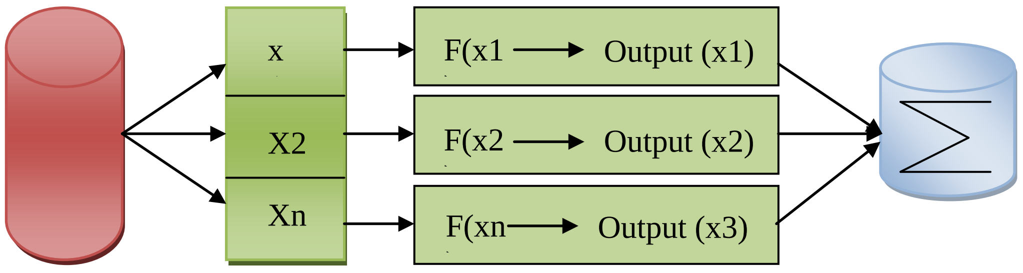


Figure 21: General concept of PLINQ

The Input side optimization of PLINQ means apply different strategies for input partitioning. In PLINQ there are 4 main strategies that we use for partitioning [19].

- **Range Partitioning** – Range partitioning is very common partitioning. This partitioning is only workable with indexed data source like array or list. In this partitioning the different ranges of data source assigned to different threads. he main benefit is that we know the exact length of data range and it is easy to access any element from the range directly. Figure-22 illustrates the range partitioning.

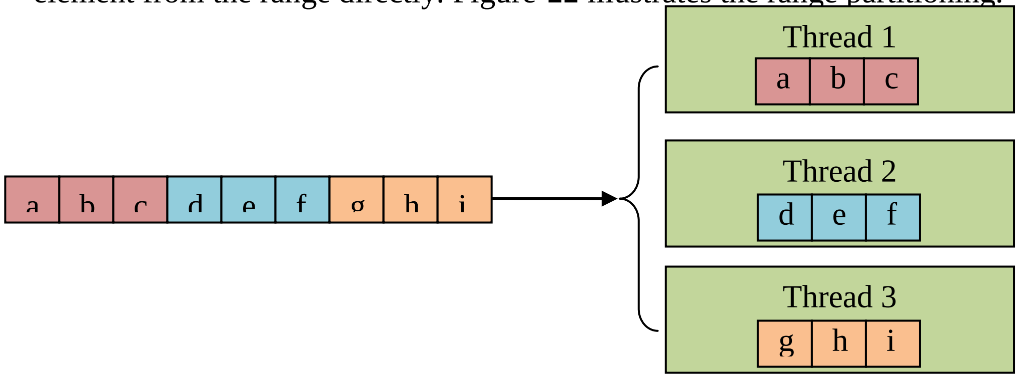


Figure 22: Range Partitioning in PLINQ

- **Chunk Partitioning** – Chunk partitioning is mainly for non-indexed data source. The data source partitioned in chunk and served to the requested thread. IEnumerables and IEnumerable<T>s do not have fixed Count Properties, so there's no way to know when or if the data source will enumerate completely. The chunk partitioning strategy is quite general and PLINQ's algorithm had to be tuned for best performance on a wide range of queries. In terms of optimization the Chunk Partitioning is important for load balancing among the cores. Figure-23 illustrates the Chunk partitioning.

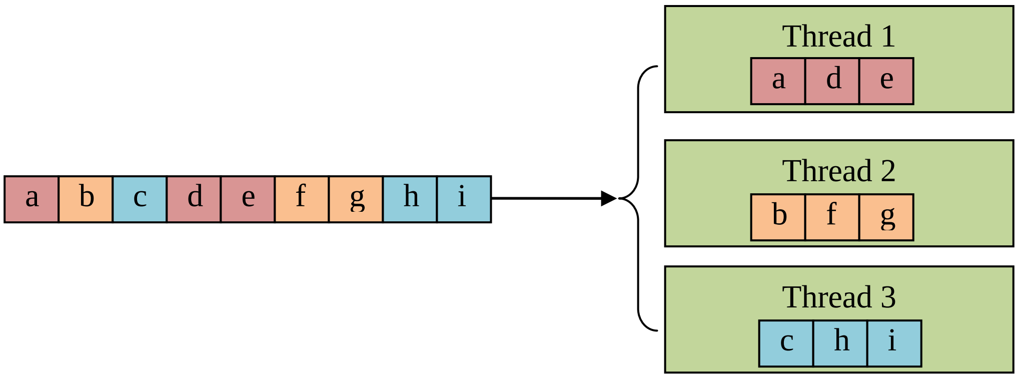


Figure 23: Chunk Partitioning in PLINQ

- **Striped Partitioning –** This Striped Partitioning is used for SkipWhile and TakeWhile and also useful for processing items at the head of a data source. In striped partitioning, the data source divided into a number of blocks. Each block has the same number of data elements. Each thread is allocated one or a small number of items from each block. Figure-24 illustrates the striped partitioning.

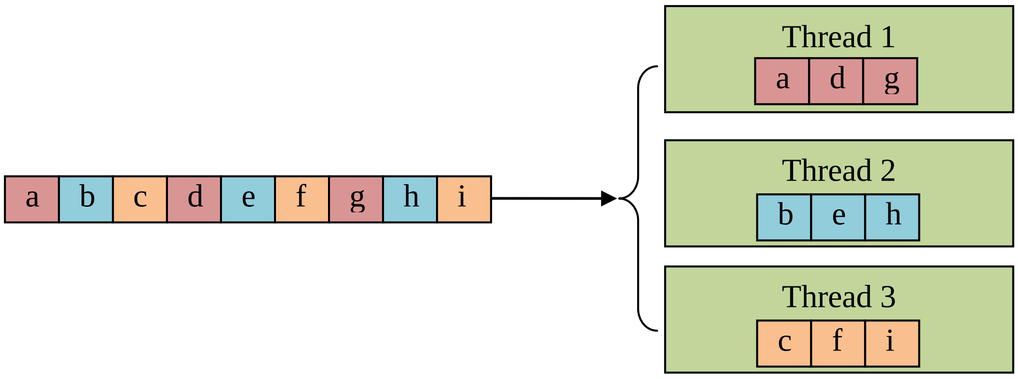


Figure 24: Striped Partitioning in PLINQ

- **Hash Partitioning –** This is a special type of partitioning. The hash partitioning is used by the query operators such as Join, GroupJoin, GroupBy, Distinct, Except, Union, Intersect. At the time of hash partitioning, all data is processed and assigned to threads with different hash codes. Each thread processes data item with identical hash-codes. Based on element key hash partitioning assigns every element to an output partition. And this is useful for creating the hash table. Figure-25 illustrates the Hash partitioning.

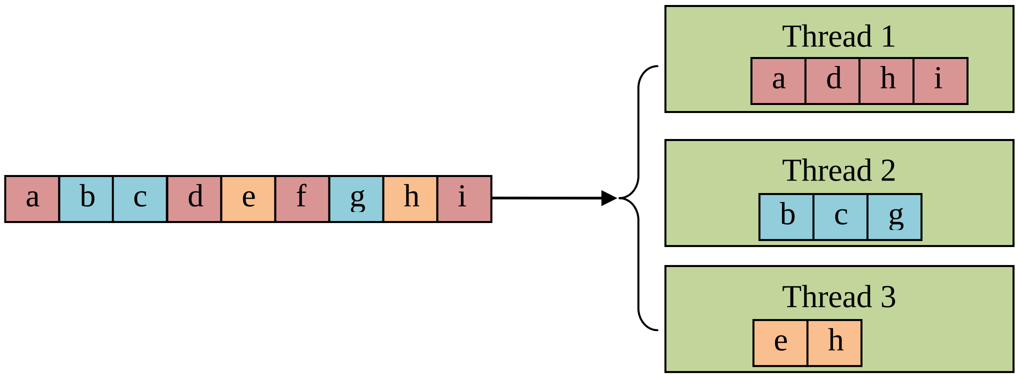


Figure 25: Hash Partitioning in PLINQ

The hash partitioning scheme is used for PLINQ. In the same partition the hash partitioning helps to make all possible matches. This mechanism makes shared data less and hash table size small. If the ordering is involved in query then the hash partitioning is not so speedy [19].

3.10.4 Data Sets of Unknown Size

In many real-world problems it is very difficult to know the data set size in advance. Because data comes from different external source and store the data in data structure. But doesn't keep track the size of data set.

Parallel.ForEach and PLINQ patterns are able to parallelize such problems. The Parallel.ForEach and PLINQ constructs work on data streams in the form of enumerables [14]. **Enumerables**, however, are based on a pull-model, such that both Parallel.ForEach and PLINQ are use enumerable form which has "move next" to get the next element. However, in producer/consumer pattern we can use **BlockingCollection**. BlockingCollection's **GetConsumingEnumerable** method provides data for Parallel.ForEach or PLINQ. Here is an example of BlockingCollection [14].

```
private BlockingCollection<T> _streamingData = new BlockingCollection<T>();
// Parallel.ForEach
Parallel.ForEach(_streamingData.GetConsumingEnumerable(),
item => Process(item));
// PLINQ
var q = from item in _streamingData.GetConsumingEnumerable().AsParallel()
...
select item;
```

3.10.5 Parallel.ForEach Vs PLINQ for Threading

The Parallel.ForEach and PLINQ are use threading for parallelism. But the Parallel.ForEach and PLINQ use slightly different threading models in the .NET Framework 4.

By default, the PLINQ pattern uses a fixed number of threads to execute a query, means it uses the all logical cores in the machine. On the other hand, Parallel.ForEach can use variable number of threads. It can recognize the best accommodation of current workloads over time using ThreadPool's at the time of injecting and retiring threads in Pool. The Parallel.ForEach continually monitor new available threads, and the ThreadPool is continually trying out injecting new threads into the pool and retiring threads from the pool to see whether more or fewer threads is beneficial[14]. However, Parallel.ForEach block the threads used by the loop and Parallel.ForEach may not release a blocked thread to back to the ThreadPool for other uses. That's why sometime the total number of threads increases dramatically. And total memory uses of oversubscribed thread makes a negative performance. For addressing this, we can set an explicit limit on the number of threads in Parallel.ForEach. This is possible using specifically MaxDegreeOfParallelism field of ParallelOptions.

3.11 Parallelism through Task Class

In PFX (Parallel Framework) the Task parallelism is the low-level approach of parallelization. In .NET Framework 4 the **System.Threading.Tasks** has the following classes for working at that level of parallelism [11] [61].

Class	**Purpose**
Task	For managing a unit of work
Task<TResult>	For managing a unit of work with a return value
TaskFactory	For creating tasks
TaskFactory<TResult>	For creating tasks and continuations with the same return type
TaskScheduler	For managing the scheduling of tasks
TaskCompletionSource	For manually controlling a task's workflow

Table 2: Task Parallelism Classes

Task is another important API for managing parallelism. In **ThreadPool** has an overhead of starting a thread. Using the Task we can avoid that overhead [11]. The **Tasks** of .NET Framework 4 has a significant efficient over the previous **ThreadPool.QueueUserWorkItem**. Tasks do more easy and efficient way whatever the Thread Pool can do. Tasks also provide powerful features for managing unit of work. Feature of Task are [35]:

- Tune a task's scheduling
- Establish a relationship of parent/child when one task is started from another
- Implement cooperative cancellation
- Wait on a set of tasks—without a signaling construct
- Attach "continuation" task(s)
- Schedule a continuation on the basis of multiple antecedent tasks
- Propagate exceptions for continuations, parents, and task consumers

3.11.1 Task.Factory

Task. Factory provides access for factory methods for creating a Task and Task<TResult> instances [36]. The Task. Factory makes a static property on Task that returns a default TaskFactory Object. Basically the common use of this TaskFactory is to create three kind of new tasks.

a) "Ordinary" tasks (via StartNew)
b) Continuations with multiple antecedents (via ContinueWhenAll and ContinueWhenAny)
c) Tasks that wrap method that follow the asynchronous programming model (via FromAsync)

After creating and executing several tasks you can wait for multiple tasks using the static methods Task.WaitAll (wait for all tasks to finish) and Task.WaitAny (waits for just one task to finish). WaitAll to similar as Wait();

3.11.2 Creating and Starting Tasks

We can create and start Task using Task.Factory.StartNew, passing in an Action delegate:

```
Task.Factory.StartNew(()=> Console.WriteLine("Hello from a task!"));
```

The generic version is as follows with get result back [11].

```
Task<string>task = Task.Factory.StartNew<string>(()=>
// Begin task
{
  using(varwc =new System.Net.WebClient())
    return wc.DownloadString("http://www.linqpad.net");
});

// We can do other work in parallel...
RunSomeOtherMethod();

// Wait for task to finish and fetch result.
string result =task.Result;
```

A created tasks can also be run synchronously using Run Synchronously instead of Start.

3.11.3 Child tasks

We can also run one task inside the other task with a parent-child relationship using TaskCreationOptions.AttachedToParent [11].

```
Taskparent = Task.Factory.StartNew(()=>
{
  Console.WriteLine("I am a parent");

  Task.Factory.StartNew(()=>        // Detached task
  {
    Console.WriteLine("I am detached");
  });

  Task.Factory.StartNew(()=>        // Child task
  {
    Console.WriteLine("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

If program waits for any parent task to complete, it also waits for its child task as well.

3.11.4 Task Schedulers and UIs

A task scheduler allocates tasks to threads. This is represented by the abstract **TaskScheduler** class. The .NET Framework 4 provides two types of implementation. The first one is *the default scheduler* that works with the CLR thread pool, and the second one is *synchronization context scheduler*. The Task Schedulers helps to work with the threading model for WPF and Windows Forms. For example, an application which will fetch some data from a web service in the background and the result will show in a form. We can use the *synchronization context scheduler* for getting the features [11].

```
Public partial class MyWindow: Window
{
 TaskScheduler _uiScheduler;
// Declare this as a field so we can use
 // it throughout our class.
 Public MyWindow()
 {
  InitializeComponent();

  // For form creation getting the UI scheduler for the Thread.
  _uiScheduler =TaskScheduler.FromCurrentSynchronizationContext();

  Task.Factory.StartNew<string>(SomeComplexWebService)
    .ContinueWith(ant => lblResult.Content = ant.Result,_uiScheduler);
 }
 String SomeComplexWebService(){...}
}
```

3.12 Parallelism through Concurrent Collections

The System.Collections.Concurrent namespace has following tools which helps to implement the low level data structure in concurrent way [37].

Concurrent collection	**Nonconcurrent equivalent**	**Description**
ConcurrentStack<T>	Stack<T>	Represents a thread-safe last in-first out (LIFO) collection.
ConcurrentQueue<T>	Queue<T>	Represents a thread-safe first in-first out (FIFO) collection.
ConcurrentBag<T>	(none)	Represents a thread-safe, an unordered collection of objects.
BlockingCollection<T>	(none)	Provides blocking and bounding capabilities for thread-safe collections that implement IProducerConsumerCollection<T>.
ConcurrentDictionary<TKey,TValue>	Dictionary<TKey,TValue>	Represents a thread-safe collection of key-value pairs that can be accessed by multiple threads concurrently.

Table 3: Concurent collections classes ([37])

In general when users write parallel application typically they need to implement synchronization mechanism with their own data structure. Such as, reading/writing data in shared data structure concurrently. In .NET Framework 4.0 has several new collections in the System.Collections.Concurrent namespace for controlling data structures [37].

In System.Collections.Concurrent namespace the ConcurrentStack is a lock-free[1] thread-safe implementation. Using this ConcurrentStack we can implement the standard stack operation. In a normal stack functions are Push, Pop and Peek and it performs operation in LIFO (Last in First out). The ConcurrentStack<T> provides Push. It does not provide Pop and Peek function. Instead of those it provides TryPop and TryPeek. A Push method normally puts a new item on the top of the stack. And the structure is

```
        ConcurrentStack<string>myst = new ConcurrentStack<string>();
        myst.Push("Test")
```

The TryPop helps you to pop the top item of the stack. The structure of TryPop is

```
             string currentData = "";
      if (myst.TryPop(out currentData))
             {
         //ConsumeData(currentData);
             }
```

Here the TryPop returns a true value if there is a data to pop otherwise false. It is very difficult to know in multi-threaded implementation that there is an item in stack to pop. Because there are different threads working on the same data structure at the same time.

Sometime user needs to look not pop the top item of a stack. In that case the "TryPeek" of ConcurrentStack will provide the support.

```
             string currentData = "";
      if (myst.TryPeek(out currentData))
             {
         //ConsumeData(currentData);
             }
```

Here is an example of ConcurrentStack<T> in a producers/consumer scenario [40].

```
      ConcurrentStack<string>mystack= new ConcurrentStack<string>();
  const int COUNT = 10;
   public void ConsumeData()
     {
      //create the producers
      for (int i = 0; i < COUNT; i++)
      {
             Thread cProducer = new Thread(new ThreadStart(delegate()
         {
         for(int cIndex = COUNT; cIndex > 0; cIndex--)
         {
           mystack.Push(
          Thread.CurrentThread.ManagedThreadId.ToString()+ "_
                                    " +cIndex.ToString());
           }
         }));
         cProducer.Start();
```

[1]"lock-free" mean that they don't use traditional locks using the "lock" keyword or using Monitors. They perform compare and swap operations by using the System.Threading.Interlocked.

```
            }
            Thread.Sleep(500);
            //consume data
            string cData = "";
            while (mystack.TryPop(out cData))
            {
               //ConsumeData(cData);
            }
        }
```

Like ConcurrentStack the .NET Framework 4.0 has the ConcurrentQueue in System.Collections.Concurrent namespace for implementing the standard queue data structure. And also it is concurrent. The queue work as FIFO (First in first out) and the main operations are enqueue and dequeue. The c# Enqueue() method is using for insert item into the queue and TryDequeue() method for remove and return the object at the beginning of the ConcurrentQueue<T>. Also there is TryPeek to return an object from the beginning of the ConcurrentQueue<T> without removing it. The syntaxes are. [18]

```
            var cqueue = new ConcurrentQueue<string>();
            cqueue.Enqueue("test");
            string cData = "";
    if (cqueue.TryDequeue(out cData))
            {
               //pop data
            }
            string cData = "";
    if (cqueue.TryPeek(out cData))
            {
               //peek data
            }
```

Besides the ConcurrentStack and ConcurrentQueue the .NET 4 has ConcurrentBag, BlockingCollection, ConcurrentDictionary etc in System.Collections.Concurrent for lower-level parallel programming activities.

4 PERFORMANCE FACTORS OF PARALLEL APPLICATION

4.1 Amdahl's law

Amdahl's law is very essential rules for establishing the theory for achieving maximum speed-up of a parallel program. Amdahl's law places a strict limit on the speedup that can be understandable by using multiprocessors [7] [38].

"*The Amdahl's law states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization*"[10].

Amdahl's Law can be shortly expressed by using the following equation, where f is the fraction of time spent on serial operations and p is the number of processors [39]:

$$\text{Speed up} \leq \frac{1}{f + (1-f)/p}$$

When the number of processors increases the term (1-f)/p approaches to 0. So, from the above equation we can get the following equation [39]:

$$\lim_{p \to \infty} \text{speed up} \leq \frac{1}{f}$$

From the above speed-up equation we can see that, the maximum potential speedup for a parallel program depends on how much a program can be parallelized.

"*Amdahl's Law clearly demonstrates the law of diminishing returns: as the number of processors increases, the amount of speedup attain by adding more processors decreases*" [39].

For example, there are three programs and the values of f of those three programs are 0.1, 0.01 and 0.001. According to the above equation the possible maximum speed-up of three programs are respectively 10, 100 and 1000. Figure-26 in the next page represents the speedups of above discussion for up to 1024 processors [39].

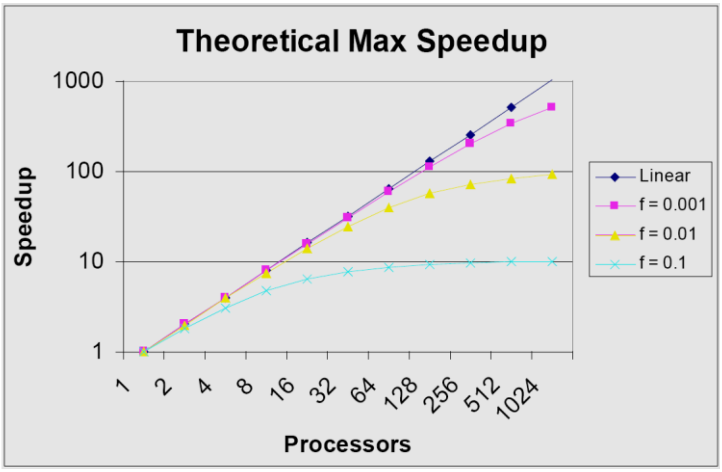


Figure 26: Theoretical maximum speedups for three parallel programs with different values of f (From [39])

4.2 Load Balancing

Load balancing represents the proper way of workload distribution. Proper load balancing will distribute the work to all tasks and all tasks will be busy at maximum time [5]. Load balancing minimizes the execution time of the program by assigning the task to each processor's work proportion to its performance. Two major categories of load balancing are static and dynamic load balancing [41].

4.2.1 Load Balancing Algorithms

There are various algorithms of load balancing which are proposed in the literature based on some specific application domain. Some algorithms are for large parallel jobs, where as some works for short and quick jobs. Some are handling heavy data tasks while other parallel task does compute very fast. The common four basic types of load balancing algorithms are as follows [41]:

- Workstations performance monitoring (Load monitoring)
- Information exchange between workstations (Synchronization)
- Making the work movement decision and calculating new distributions (rebalancing Criteria)
- Real data movement (Job migration)

4.2.2 Load Balancing Strategies

There are three main questions usually classify the strategy of load balancing algorithm. Three important questions of load balancing strategies are: I) who will make decision of load balancing, II) what information is needed to make load balancing decision, and III) where the decision of load balancing is made.

4.3 Granularity

"The granularity of parallel program is the average computation cost of a sequential unit of computation in the program"[42]. Granularity of parallel program is known as the ratio between computation and communication of different processors working on the same problem.

This definition based on overhead computation cost eliminates the specific to the run time. Further execution cost of a program is imposed by the creation and management of parallel threads. For an example thread creation need operation like stack allocation. For achieving better parallel performance, it requires to maintain the computation to proper use of available processor and to provide the possibility of overlapping computation and communication. The efficiency of different strategies for creating parallel threads has to consider the overhead for task creation, communication on the both ends and cleaning up completed threads. The performance of a coarse-grained computation is better for high latency machine.

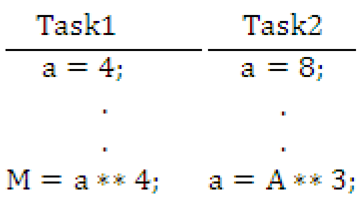
4.4 Data Dependency

Data dependency exists when the same storage locations are used by multiple processors in parallel programming application. It is important for parallel programming because data dependency is one of the primary inhibitors to parallelism and the result is multiple uses of the same location in storage of different tasks. Some examples of data dependency are:

Example 1: Loop carried data dependence [5]

```
                    for(i=1; i<=end_limit; i++)
                              {
                              x[i]=x[i-1]*4;
                              }
```

The above mentioned code has data dependency. The x[i-1] Must be computed before the calculation of x[i]. Therefore x[i] demonstrates a data dependency on x[i-1].
Example 2: Loop independent data dependence [5]



Task1	Task2
a = 4;	a = 8;
.	.
.	.
M = a ** 4;	a = A ** 3;

In the above equation the value of M depends on the value of a.
However, data dependency is very important to design parallel programs and loops are probably the most well-known target of parallelization efforts.

4.5 Static and Dynamic Partitioning

Partitioning is another important performance factor. Static partitioning and Dynamic partitioning [45] applicability depends on the problem to be solved. Sometimes static partitioning is good and sometimes dynamic. If the workload per iteration is approximate same then static partitioning represents an efficient way to partition data set [14]. On the other hand if the workload per iteration is not equivalent then the dynamic partitioning is better. Because of work nature and improper partitioning sometime falls the execution in load imbalance. And in this situation some parts are finished in short time and some part takes more time.

If the total workload is small and the program creates many partition/threads automatically then synchronization between threads become an overhead for the CPU than the original work execution. But if the total workload is very large then the synchronization is negligible.

4.6 Deadlock

Parallel application performance can be hampered due to deadlock. Deadlock is a situation in parallel programming application where two or more processes is unable to continue because of circular wait to do something [14].

Remember that a node has to insure that no information received later can change its output; this may involve waiting for other inputs, the cycle of waiting nodes results in a deadlock. For an example, a task is waiting for getting output from the server while the server is waiting for more input from the scheming task before outputting anything, then deadlock exist. Parallel program should design such a way so that deadlock cannot occur.

4.7 Communication Patterns

For some cases if the number of processors is increased then the execution time attributable computation will decrease, but it will also increase the execution time attributable to communication. The required communication time depends upon a given system's communication parameters and bandwidths.
Generally when number of processors increase to execute one task the communication frequently trend to increase while the message size will decrease, when more processors share same data set and every processor handle a smaller amount of total data. In an application, if the overall communication time is large corresponding to the main computation time then the performance will be hampered. And the communication time will be the bottleneck in the performance of these applications [44].

5 EXPERIMENTAL RESULT EVALUATION & PERFORMANCE MEASURE

"Performance refers to the responsiveness of the system — the time required to respond to stimuli (events) or the number of events processed in some interval of time"[27].

Performance is the main concern on parallel program [17]. In general, the parallel application has better performance than the sequential program in a parallel architecture environment or parallel application platform. The expected performance depends on the available parallel architecture, sequential computation time in each process and their difficulties [51]. Also the performance of parallel application [30] [31] varies according to different approaches of parallel application and workload scalability on available architecture. Chapter-3 introduced the parallel programming constructs and APIs offered by .NET Framework 4 and we are using C# programming language. For our experiment and performance evaluation we have tried to build up several parallel applications for above discussed APIs in C#.

We have done our experiment in two machines with different configuration. The Computer configurations are as follows

Machine-A:
Intel® Core(TM) 2 Duo CPU E8400 @ 3GHz 2.99 GHz with 3.5GB internal memory.

Machine-B:

Processor: Intel® Xeon® CPU x5660 @ 2.80 GHz, 2.79GHz (2 Processor, 12 cores each) with 12GB internal memory.

We have tested those programs in different ways and different data ranges. In this chapter there are 9 different experiments. 1) Sequential Vs Static Multithreading for calculating simple equation using machine A and B, 2) Sequential Vs Parallel.For for calculating simple equation using machine A and B, 3) Sequential ForEach Vs Parallel.ForEach for downloading web contents using machine B, 4) Sequential Vs Task Parallelism using Parallel.Invoke for text file processing in machine A, 5) Parallel application performance in terms of increasing number of threads in two types of parallel environment for finding prime numbers using machine A and B, 6) PLINQ Performance with different patterns for downloading web contents using machine B, 7) PLINQ Performance with different workload for finding prime numbers using machine B, 8) LINQ Vs PLINQ Performance with different number of cores for finding baby name popularity using machine B and 9) Speed-up for calculating simple equation using static multithreading and ThreadPool in machine B. We have got different result as well. At the time of result evaluation we have got some better performances and also achieved some bad experiences. We are describing several experiments for evaluating performance.

5.1 Sequential Vs Static Multithreading for calculating simple equation (experiment 1)

In our 1st experiment we had several aims. We have tried to make a concept, how sequential and parallel applications utilize the existing processor resources of different cores. Secondly we have tried to compare the performance of sequential and statically divided multithread parallel program. Also find out the efficiency of ThreadPool corresponding to the normal multithreading parallel application. This is a very primitive example and we have gotten better performance of parallel execution.

In this program we are calculating simple equation S=Cos(x)+1/Cos(x)+Sin(x)+1/Sin(x). The value of x from 0 to $1*10^8$ changed by the loop control variable. We have calculated the equation in 2 ways. One is using normal For() loop. The loop iterates 100000000 times. Another is using multiple threads. In multi thread approach we divided whole task to several number of threads (same number of CPU cores) and each thread assigned to each core [14]. The threading approach is as follows:

```
int size=100000000;
int numProcs =  Environment.ProcessorCount;
int range = size / numProcs;
double sum=0;
var threads = newList<Thread>(numProcs);
for ( int p = 0; p < numProcs; p++ )
   {
int start = p * range;
int end = (p == numProcs - 1) ? size : start + range;
      threads.Add(newThread(() =>
      {
for (int i = start; i < end; i++)
            //process Statement
      }));
   }

Stopwatch watch = Stopwatch.StartNew();
foreach (var thread in threads) thread.Start();
foreach (var thread in threads) thread.Join();
   watch.Stop();
Console.WriteLine(String.Format("Entire process took {0} milliseconds",
watch.ElapsedMilliseconds));
```

We have run the program in two multi core environments several times and we have got approximately the same result with little variation. Here we represent single execution result. First we have run the program in Machine-A (Core 2 Duo CPU). In sequential way it takes 53828 milliseconds. But in multi threading way it takes only 29952 milliseconds. The figure-27 shows the CPU uses of Machine-A (Core 2 Duo CPU) for sequential and parallel execution.

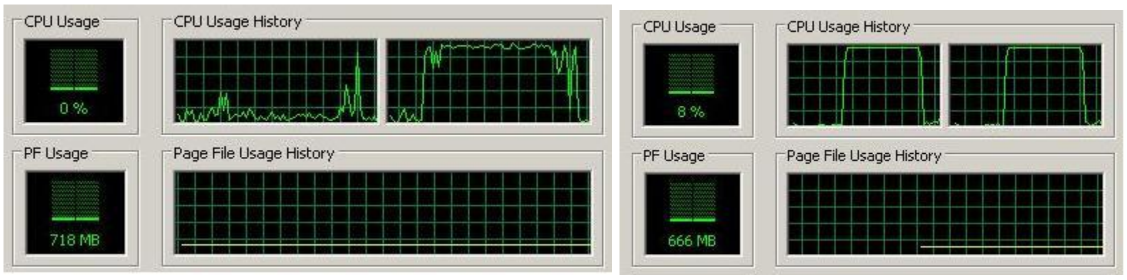


Figure 27: CPU Usage with Machine-A (Core 2 Duo CPU) for (b) sequential execution (b) parallel execution

From the above experiment we can see that in a sequential way two cores were not properly used. But in a parallel way two cores were used properly at the time of execution. Also sequential approach has been taken more time.

After that we have tested the same program in machine-B (24 cores). And we have got better performance. Total execution time was 5244 millisecond and all cores were worked simultaneously at the time of execution. This achieved performance addressed the first part of main research questions. The following Figure-28 shows the CPU uses of Machine-B (24 cores) and figure-29 shows the performance of sequential, multithreading in Machine-A (2 cores) and multithreading in Machine-B (24 cores).

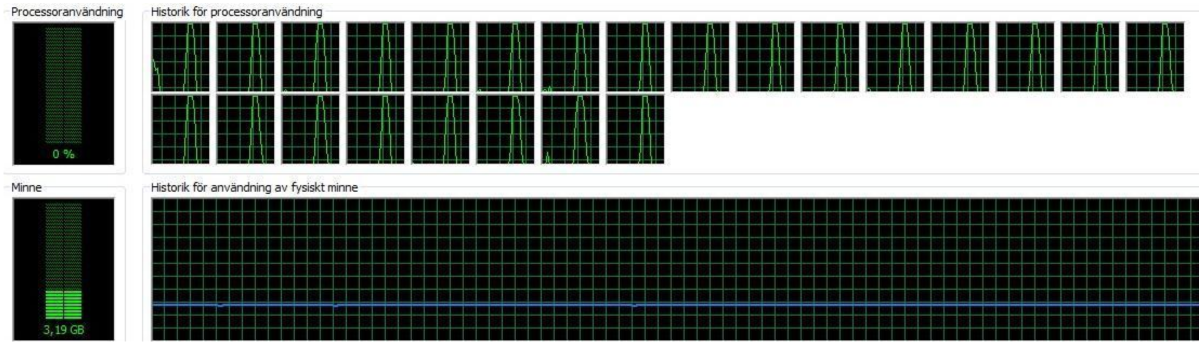
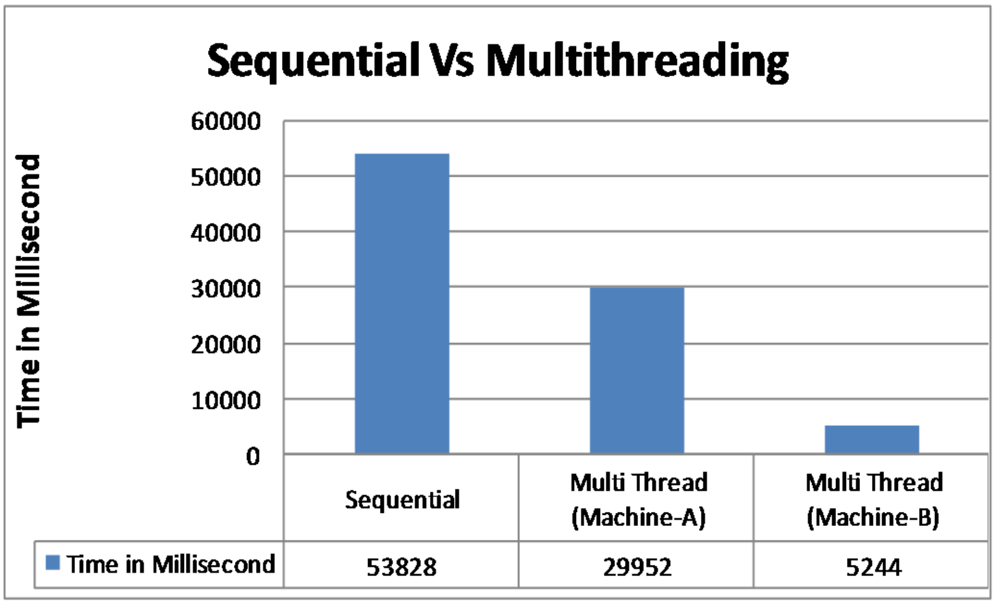

Processoranvändning
0 %
Historik för processoranvändning
Minne
3,19 GB
Historik för användning av fysiskt minne

Figure 28: CPU Usage of Machine-B (24 Cores) for parallel execution



	Sequential	Multi Thread (Machine-A)	Multi Thread (Machine-B)
🟦 Time in Millisecond	53828	29952	5244

Figure 29: Sequential Vs Static Multithreading for calculating simple equation

From the above experiment we got the improved performance of multi threading parallel application compare with sequential execution.

There were some cost factors in the previous approach. We have already introduced the ThreadPool in section 3.5. The ThreadPool merges all the threads into a pool and allows the threads to access work item queued for their processing. The ThreadPool manages the cost of life cycle of all thread, ensures an upper-limit of thread number, limits of total memory consumed. So, for reducing the cost factors of previous approach we can use the ThreadPool. Here is a part of sample code [14] for introducing ThreadPool.

```
ThreadPool.QueueUserWorkItem(delegate {
for (int i = start; i < end; i++)
                //Statement
if (Interlocked.Decrement(ref remaining) == 0) mre.Set();

    });
```

When we have tested this program, and we have seen, the total execution time is slightly less than the normal multithreading. Figure-30 shows Performance difference between Normal multithreading and using ThreadPool for both environments.

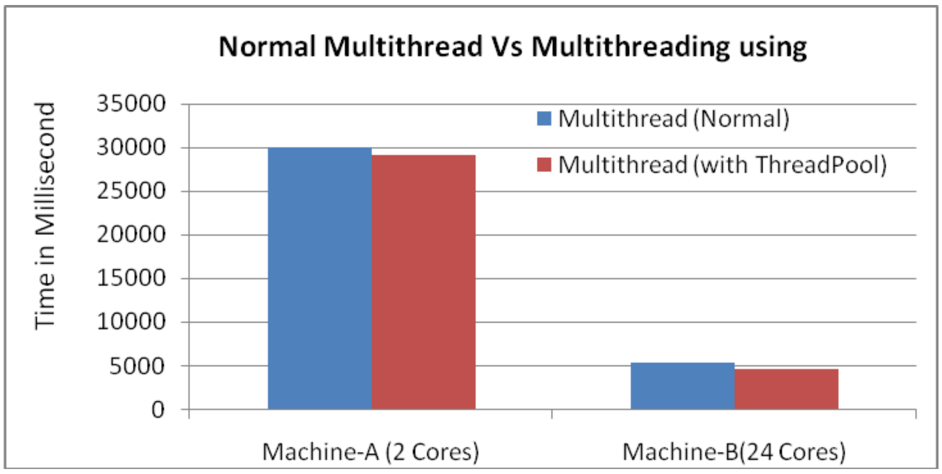


Figure 30: Normal Multithread Vs Multithreading using ThreadPool for calculating simple equetion

In this experiment we have got small performance improvement of multithreading (with ThreadPool) over normal multithreading in both machines.

5.2 Sequential Vs Parallel.For for calculating simple equation (Experiment 2)

In this 2nd experiment our aim was to evaluate the performance of parallel application using the Parallel.For API of .NET 4.0 comparing with sequential execution. We have also tried to find out the performance difference of Parallel.For using ThreadPool and Task.Factory.
In another experiment we have used **Parallel.For** for data parallelism**.** In this example we have seen parallel computing performance using Parallel.For corresponding to sequential execution. We have done the experiment in Machine-A (2 Cores) and Machine-B (24 cores). In this example we have calculated another mathematical equation

R=1/Cos(p)*tan(x,y) and R=R + Cos(q) *1/cos(q)

Where p, q, x and y are random numbers. This calculation has been done 20,000 times. In Parallel.For the above work has been done 100 times by creating multiple threads and assigning to different core. Here is the main function for calculating the equation in the iteration

```
static void calculation(int instance)
      {//Calculate initial result
         for (int i = 0; i < 20000; i++)
         {
            //Calculate result in itaration
         }
      }
```

For sequential execution the above function called by traditional for loop

```
for (int i = 0; i < 100; i++)
         {
            calculation(i);
         }
```

For parallel execution the calculation function called by Parallel.For loop.

```
static void parallelexe(object state)
      {
         Parallel.For(0, 100, i =>
         {
            calculation(i);
         });
      }
```

For combining and synchronizing all threads generated by Parallel.For we have used the ThreadPool and Task.Factory

```
ThreadPool.QueueUserWorkItem(new WaitCallback(parallelexe));

Task.Factory.StartNew(() => { paralleexe(null); });
```

In sequential execution every loop executes in sequential. But in Parallel.For different threads execute randomly and in parallel. We have tested this experiment in Machine-A and Machine-B several times. Here we mentioned the results of sequential execution, Parallel.For(using ThreadPool) and Parallel.For(using Task.Factory) in both environments.

Approach	Sequential	Parallel.For(Using ThreadPool)	Parallel.For(Using Task.Factory)
2 Cores (Millisecond)	13011	6723	6624
24 Cores (Millisecond)	12825	1737	1672

Table 4: Execution time in millisecond sequential and Parallel.For with ThreadPool and Task.Factory

The figure-31 in represents the data of table-4.

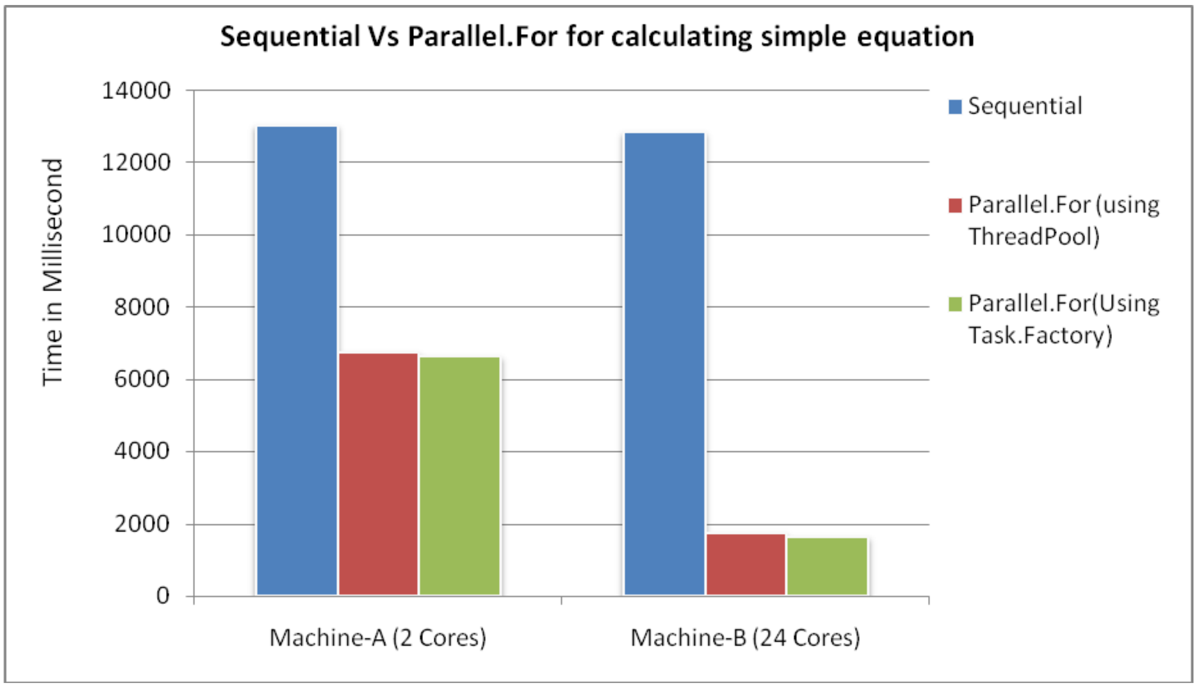


Figure 31: Sequential Vs Parallel.For for calculating simple equation

From the above figure-31, we can see that Parallel.For shows better performance over sequential execution. Also we can see that Parallel.For(using Task.Factory) has a little performance improvement than Parallel.For(using ThreadPool).

5.3 Sequential ForEach Vs Parallel.ForEach for downloading web content (Experiment 3)

For data parallelism we can also use Parallel.ForEach. Parallel.ForEach is useful for unknown data size. The aim of this experiment was to find out the performance of Parallel.ForEach API over sequential ForEach.

Here in an example of Parallel.ForEach. The main task of this example is to download the webpage contents using WebClient.DownloadString() method of System.Net from C# Language. And already we have discussed about this example in chapter-3. In C# there is a sequential looping tools named ForEach() also have the Parallel.ForEach for parallel execution. The sequential ForEach() download the contents of listed URLs in sequential way. On the other hand the Parellal.ForEach performs the same job of same list of URL in parallel way using multiple threads.

```
staticvoid Main(string[] args)
        {
        string[] urls =  { // List of URL
                                };

        //Use ForEach
        foreach (string url in urls)
                {
                        WebClient client = newWebClient();
                        client.DownloadString(url);
                }
        //Use Parallel.ForEach
        Parallel.ForEach(urls, url =>
            {
                WebClient client = newWebClient();
                client.DownloadString(url);
            });
    }
```

We have done this experiment in several times on Machine-B. After execution of this program we have got that the Parallel.ForEach took less time than the sequential ForEach. In a single execution the sequential approach has taken 7065 milliseconds, but Parallel. ForEach has taken 2205 milliseconds. The execution time varied in different tests (figure-32). Because of the downloading contents also depends on the internet speed. But there is a good performance difference between sequential and parallel application. From our opinion, the internet speed effects to both serial and parallel approaches. Because, we have tested the combine application on the same computer and approximately at the same time.

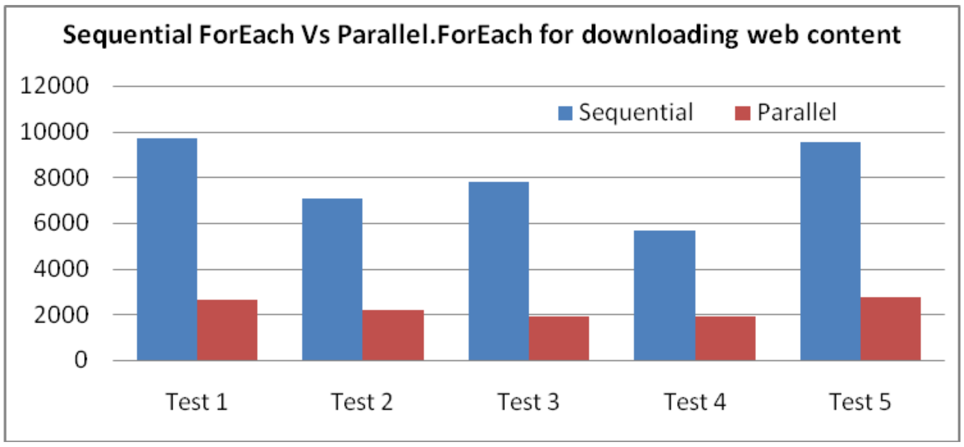


Figure 32: Sequential ForEach Vs Parallel.ForEach for downloading web content

5.4 Sequential Vs Task Parallelism using Parallel.Invoke for text file processing (Experiment 4)

In previous, all patterns such as static threading, Parallel.For or Parallel.ForEach were for data parallelism. Now we will discuss about an example of task parallelism. In task parallelism divides the whole task in different parts rather than divides the data set. And each part is executed in different threads by different cores. After assigning the task part to thread, the operating system is responsible for scheduling threads for execution. The Parallel.Invoke is the right API for task parallelism. In the following example the program reads words from a big text file and performs the following three tasks on the file. A) Finding the largest word, B) Finding the most common words and c) Finding the number of occurrence of a specific word [52]. In our experiment, we have done those three tasks in traditional sequential way and in parallel using Parallel.Invoke. The sequential approach executes those three tasks one after another. But the Parallel.Invoke executes above three tasks in concurrent using different threads.

We have done this experiment in Machine-A(2 cores). The experimental text file size was 61KB. In that file the total number of words was 9,185. The program read all words from file into memory then the above operation executes on those words. The file reading operation is out of the timing code.

We have tested five times in Machine-A (2 cores). For sequential approach the execution time was 198 milliseconds. And using the Parallel.Invoke we have got different times such as 128 ms, 130 ms, 127 ms, 130 ms and 135 milliseconds. The following figure-33 shows the experimented execution times.

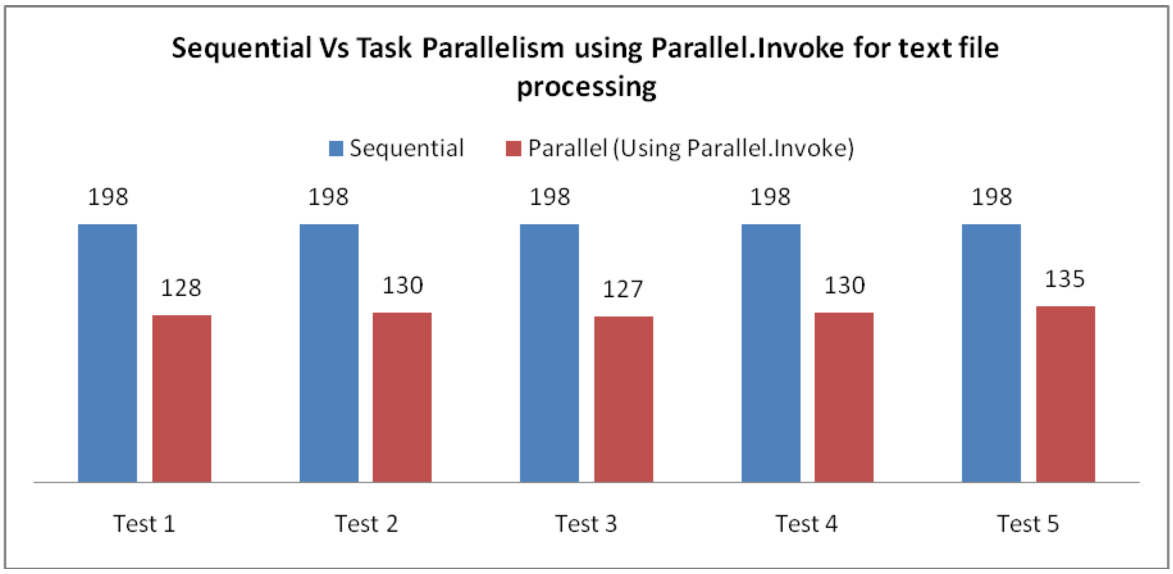


Figure 33: Sequential Vs Task Parallelism using Parallel.Invoke for text file processing

From the above experiment we can see that the parallel execution takes less time then sequential execution. But the time differences are not so much. During those five executions we have recorded the required time for individual task execution in a parallel approach. And we have got the following results.

	Test 1	Test 2	Test 3	Test 4	Test 5
Task 1	105 ms	106 ms	103 ms	103 ms	110 ms
Task 2	20 ms	21 ms	20 ms	23 ms	20 ms
Task 3	102 ms	102 ms	102 ms	93 ms	95 ms

*ms= millisecond.

Table 5 : Execution times for different Tasks in different tests

From the above table-5 we can see that the Task-1 and Task-3 have taken more times than the Task-2. In parallel execution the 2nd task has been completed early but the 1st and 3rd task has been taking more time to complete. So the workload was not well distributed between the threads. For the unbalanced workload the parallel execution times were not much shorter than sequential execution.

5.5 Parallel application performance in terms of increasing number of threads in two parallel environments for finding prime numbers (Experiment 5)

Now we are discussing data parallelism of parallel application. In this experiment we are trying to show the parallel application performance when the number of threads increases in different parallel architectures. For our experiment we have two computers Machine-A (2 cores) and Machine-B (24 cores). The program of this experiment finds prime numbers in a range from 0 to 2000000. We are finding those prime numbers using simple sequential algorithm and parallel application with different number of threads. We are using different number of threads and partitioning range between threads for finding prime numbers.

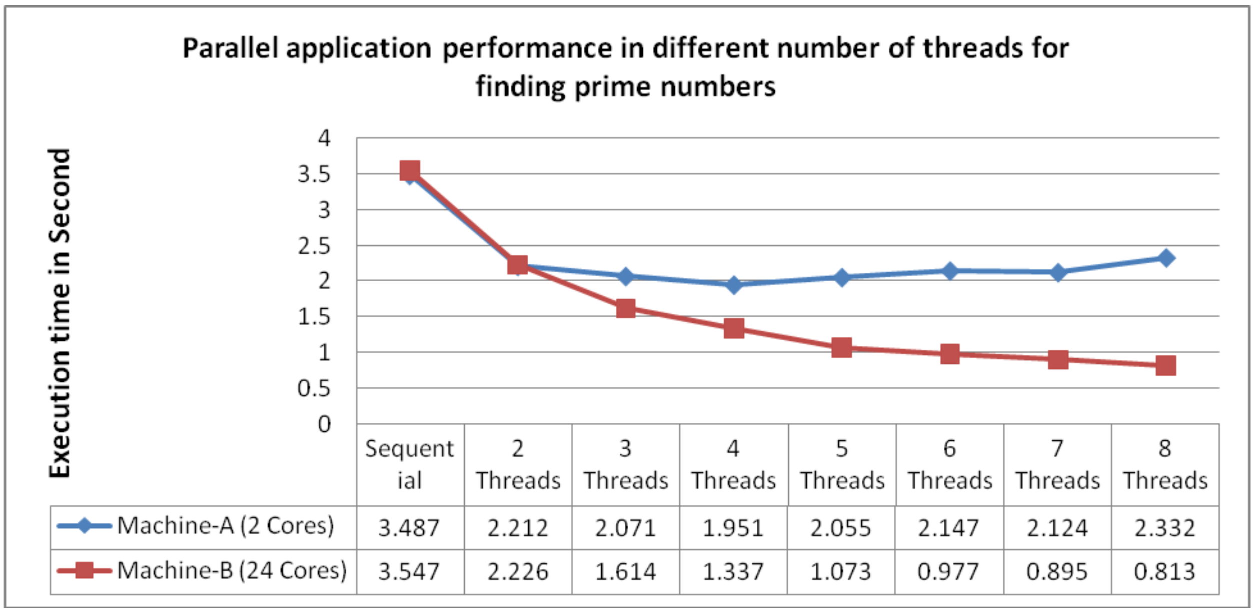
In this example there is a function *FindPrimes* which has two parameters indicate the range of values for finding prime numbers. In sequential approach we have used the sequential for loop for calling that function.

```
FindPrimes(long from, long to)
   {
     //Statement for finding Prime number

   }
```

On the other hand in parallel approach the *FindPrimesParallel* with 2 arguments are responsible for creating number of threads according to the value of THREADS variable (2 to 8). And the created threads with *PrimesThreads* help to find prime numbers using *FindPrimes* function.

```
FindPrimesParallel(long from, long to)
{
long range = (to - from) / THREADS;

ManualResetEvent[] mre = new ManualResetEvent[THREADS];
List<long>[] results = new List<long>[THREADS];

long b = from;

for (int i = 0; i < THREADS; i++)
     {
PrimesThreadInfo info = new PrimesThreadInfo();

       info.evt = mre[i] = new ManualResetEvent(false);
       info.result = results[i] = new List<long>();

       info.from = b;
       info.to = (i == THREADS - 1 ? to : b + range);
       b += range + 1;

newThread(PrimesThreads).Start(info);
     }

WaitHandle.WaitAll(mre);

foreach (List<long> lst in results)
foreach (long p in lst)
yield return p;
   }
```

We have run this application in both environments (Machine-A and Machine-B) and we have got the following results. The following results (figure-34) show the execution times for different numbers of manually created threads in both machines-A and Machine-B.



	Sequential	2 Threads	3 Threads	4 Threads	5 Threads	6 Threads	7 Threads	8 Threads
◆ Machine-A (2 Cores)	3.487	2.212	2.071	1.951	2.055	2.147	2.124	2.332
■ Machine-B (24 Cores)	3.547	2.226	1.614	1.337	1.073	0.977	0.895	0.813

Figure 34: Parallel application performance in different number of threads for finding prime numbers

From the above experiment, we can see the better performance of 2 threads over sequential in both environments. In Machine-B(24 cores) we increase the threads up to 8 and performance has been increased gradually. But in a Machine-A(2 cores) we have got the better performance up to 4 threads. After that additional increment of threads did not show the better performance in Machine-A(2 cores). From the above experiment we can say that the over subscription of thread on CPU resources sometimes decreases the parallel application performance.

5.6 PLINQ performances with different patterns for downloading web contents (Experiment 6)

In experiment 6 our aim was to find the performance of PLINQ with sequential execution. Also less number of threads will create less I/O request and it might reduce the I/O overlapping. Here we have tried to evaluate the PLINQ performance using .AsParallel() and .WithDegreeOfParallelism(). We have enhanced the webpage content download program discussed in experiment 3. Here we have done that work for another website in 3 different ways.

a) Normal sequential using ForEach
b) Parallelism of PLINQ using .AsParallel()
c) Parallelism of PLINQ using .AsParallel().WithDegreeOfParallelism()

In sequential approach the ForEach will go to all websites in the list.

```
var sitelist = from site in new[]
            {
            //several URL
}
```

```
let p = WebRequest.Create( new Uri(site)).GetResponse()
 select new
 {
    site,
    Length = p.ContentLength,
    ContentType = p.ContentType
 };

foreach (var siteresult in sitelist)
            {
 //do something
}
```

In PLINQ using .AsParallel() the total works have been done by parallel in machine B. The whole work divides in number of threads and use the number of cores. Generally the PLINQ attempt to uses all available processors in a computer [53]. PLINQ distribute the CPU work as a way that the all threads get a fair distribution. Also the PLINQ prevents creating a lot of threads which will be overhead for system [55]. According to this concept the PLINQ (with .AsParallel()) of this experiment might create 24 threads for work simultaneously. But we don’t know each of these operations blocking and waiting on I/O operation. Also we don’t know how much amount of CPU work. And the parallel application performance might be hampered.

```
      }
.AsParallel()
let p = WebRequest.Create( new Uri(site)).GetResponse()
```

In this section we have discussed about the degree of parallelism in PLINQ. The .WithDegreeOfParallelism() ensure the maximum number of processors used for PLINQ simultaneously [54].

In this experiment we used .WithDegreeOfParallelism(8) to ensure the maximum 8 concurrently execution and use of processor. The less number of threads will create less I/O request and it might reduce the I/O overlapping.

```
      }
.AsParallel().WithDegreeOfParallelism(numofthread)
let p = WebRequest.Create( new Uri(site)).GetResponse()
```

In our experiment we got some different results. We have executed the program 5 times and average the output times.Figure-35 shows PLINQ performance using different approaches.

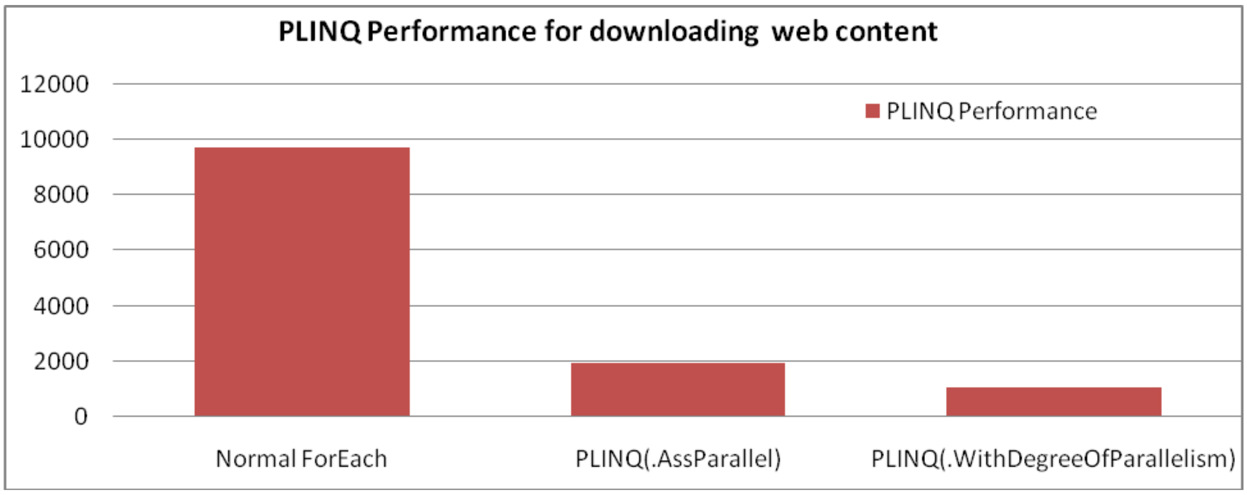


Figure 35: PLINQ performance using different approaches

5.7 PLINQ performances with different workload for finding prime numbers (Experiment 7)

The goal of experiment 7 was to evaluate parallel application performance for different workload (different data range) on CPU using PLINQ. In previous we have calculated the execution time of parallel application for a fixed set of data but we have used a different number of threads. But in this experiment we were not using different number of threads. We were using the all available CPU resources and we execute for different range of data.

In the previous example we have calculated prime numbers from 0 to 2000000 using different number of threads. But now we are using the full CPU resources for calculating prime numbers from different ranges such as 10, 100, 500, up to 150000.
In sequential approach *FindSequential* function will take all numbers list within the given range and it will check in sequential loop which numbers are prime number.

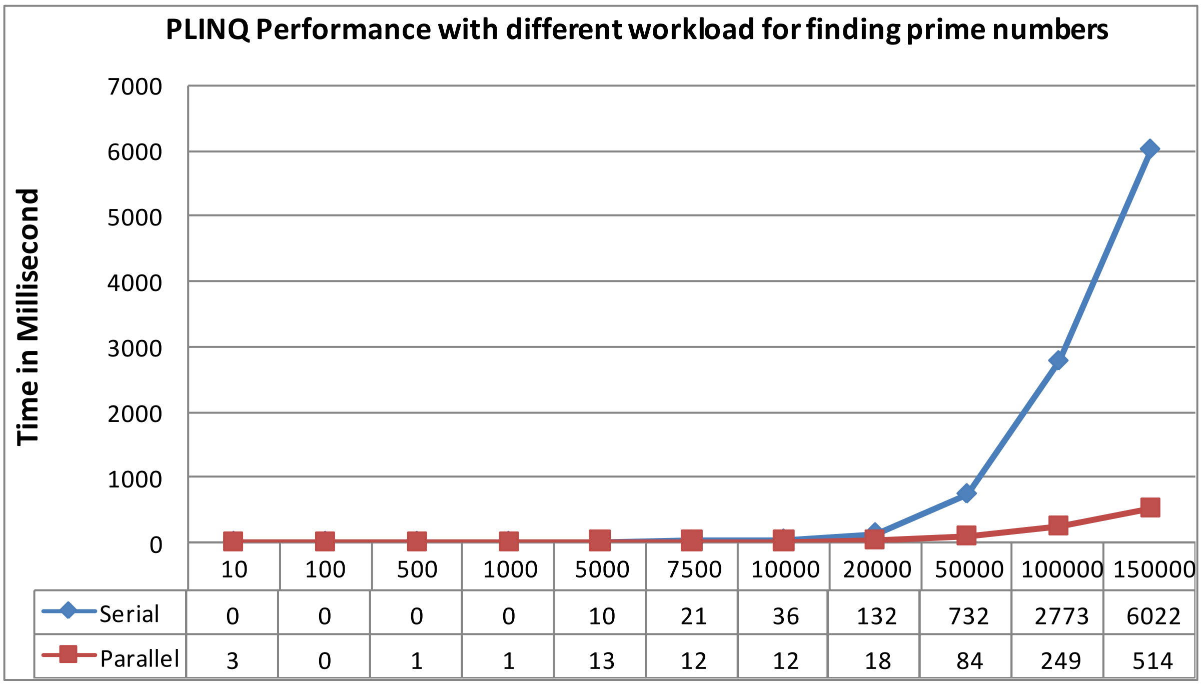
```
privatelong FindSequential(List<PotentialPrime> numbersinRange)
    {
                Stopwatch watch = newStopwatch();
        watch.Start();

                foreach (Primes n in numbersinRange)
                        {
                        var q = PrimeCheck(num.Value);
                n.PrimeCheck = q;
                }
                return watch.ElapsedMilliseconds;
      }
```

In parallel approach *FindParallel* function again takes all number's list within the given range and it will check PLINQ query which numbers are prime number.

```
privatelong FindParallel(List<PotentialPrime>numbersinRange)
      {
                Stopwatch watch = newStopwatch();
        watch.Start();
```

```
                    //PLINQ query for finding numbers which prime number.
                    var query = from n in numbersinRange.AsParallel()
                            select new { N = n, PrimeCheck = PrimeCheck(n.Value)
                                    };

                    Parallel.ForEach(numbersinRange, n =>
                    {
                            var q = PrimeCheck(n.Value);
                    n.PrimeCheck = q;
                    });

                    return watch.ElapsedMilliseconds;
            }
```

The above sequential and parallel approaches called a function named PrimeCheck(num) for checking a prime number. This PrimeCheck() function follow general algorithm. In this function there is no parallel mechanism.
We are executing in sequential algorithm and parallel using PLINQ. When we have executed this program in Machine-B (24 cores), we have got different results of execution time. The following figure-36 represents the time performance of sequential execution and parallel execution. The X-axis represents the number ranges in which we have to find the prime numbers and the Y-axis represents the required time of several ranges for serial and parallel.



	10	100	500	1000	5000	7500	10000	20000	50000	100000	150000
Serial	0	0	0	0	10	21	36	132	732	2773	6022
Parallel	3	0	1	1	13	12	12	18	84	249	514

Figure 36: PLINQ Performance with different workload for finding prime numbers

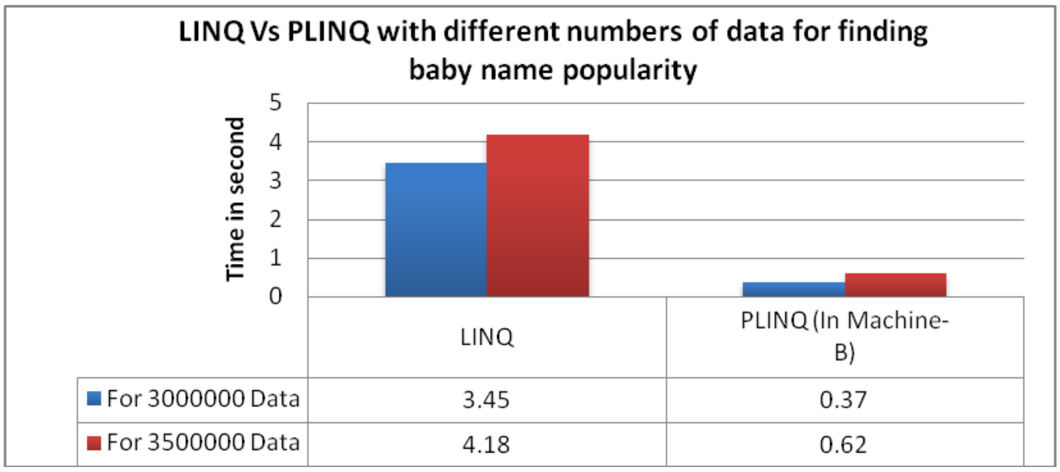
From the above experimental results we can see that, initially with the small workload (less range of data) the parallel execution and sequential execution does not show the performance difference. Because the total works are so small that it just misuses the parallel power. Creating, destroying several threads and synchronization between threads is extra overhead for the CPU, in terms of low workload and small process execution. But for the more workload (large number of data set) the PLINQ shows it performance corresponding to sequential execution. For large data range like 50000, 100000 or 150000 the parallel execution (PLINQ) and sequential execution time difference is increased.

5.8 LINQ Vs PLINQ performances with different number of cores for finding baby name popularity (Experiment 8)

The aim of this experiment was to performance comparison of LINQ and PLINQ for different data ranges and using different number of cores with a fixed number of data. The experiment finds the baby name popularity from a large baby name database using LINQ and PLINQ. We have tried to show the parallel application (PLINQ) performance using various numbers of cores from 2 to 24. This program we collect from released code sample of The Parallel Framework team at Microsoft at http://code.msdn.microsoft.com/ParExtSamples. In this example the program creates a large number of baby name's database in main memory from the combination of states, names and years. The program gets a name and state (name: Stephen, state: NH) and queries the popularity of baby name in the states within different years from a large number of generated baby information. Finally it sorts the result and displays in a user interface [50]. Here LINQ and PLINQ have been used for doing this job. Here are both queries of LINQ and PLINQ [50]. Here the both query get Stephen as _userQuery.Name and NH as _userQuery.State.

```
// SEQUENTIAL QUERY
_sequentialQuery = from b in _babies
where b.Name.Equals(_userQuery.Name,
StringComparison.InvariantCultureIgnoreCase) &&
b.State == _userQuery.State &&
b.Year >= YEAR_START && b.Year <= YEAR_END
orderby b.Year select b;

// PARALLEL QUERY
_parallelQuery = from b in _babies.AsParallel().WithDegreeOfParallelism(numProcs)
where b.Name.Equals(_userQuery.Name,
StringComparison.InvariantCultureIgnoreCase) &&
        b.State == _userQuery.State &&
        b.Year >= YEAR_START && b.Year <= YEAR_END
orderby b.Year select b;
```

We have tested this program for 3000000 and 3500000 data in Machine-B(24 cores). We have got the following results (in figure-37) and we can see that the PLINQ (use available CPU resource of machine-B) has better performance than LINQ (Sequential).



	LINQ	PLINQ (In Machine-B)
🟦 For 3000000 Data	3.45	0.37
🟥 For 3500000 Data	4.18	0.62

Figure 37: LINQ Vs PLINQ with different numbers of data for finding baby name popularity

Also we have tested and recorded the PLINQ execution time for 3000000 data and different number of cores from 2 to 24. Here we can see that the LINQ takes 3.45 Seconds, the PLINQ with 2 cores takes only 1.87 Seconds, 3 cores takes 1.30 Seconds and so on. Finally 24 cores take only 0.37 Sec. The execution times are only for main calculation not for the whole program with sequential and parallel parts. At the time of increasing the core initially the performance was very good but with more processor the performance is not so good (shows in figure-38). Actually work on less data with huge parallel resources just misuses the power.

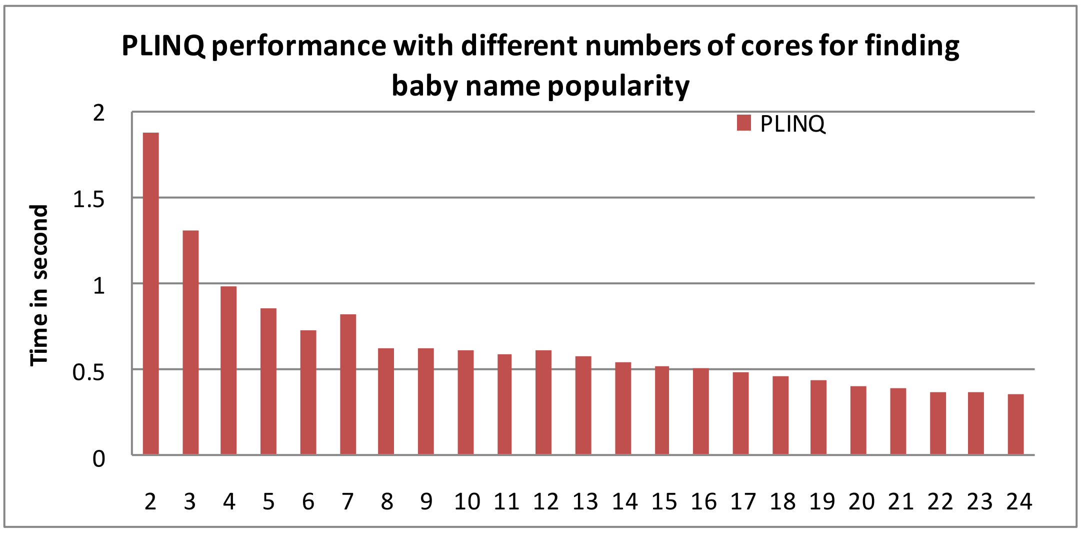
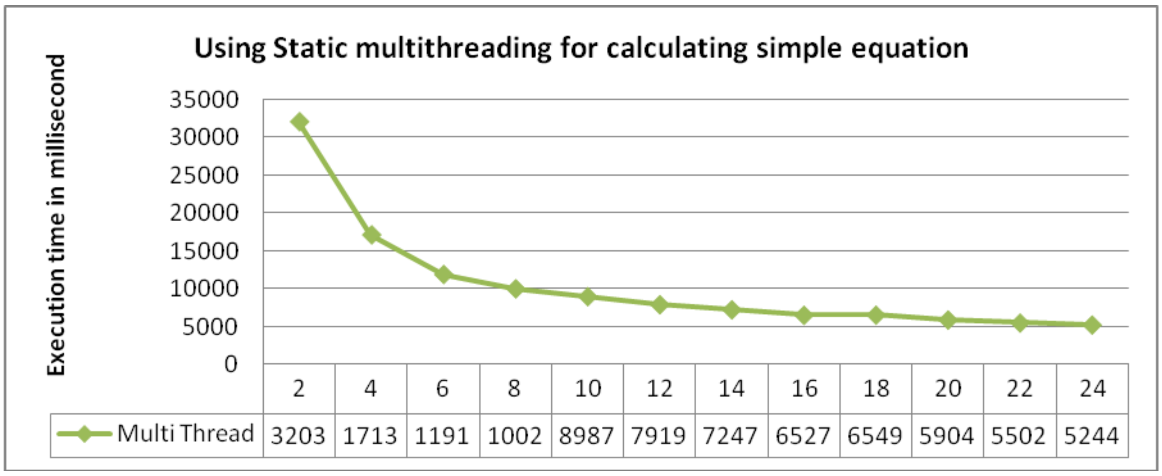


Figure 38: PLINQ performance with different numbers of cores for finding baby name popularity

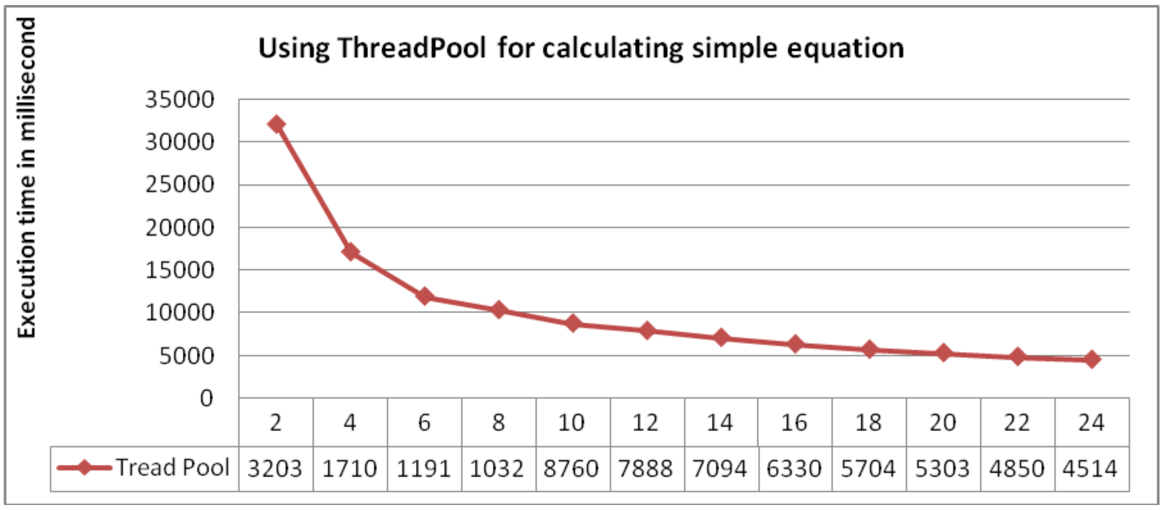
5.9 Speed-up for calculating simple equation using static multithreading and ThreadPool (Experiment 9)

The aim of this experiment was find out the speed-up of static multithreading parallel application on the basis of different numbers of cores use. Here we are enhancing the Experiment-1. In this experiment the program evaluates that simple mathematical equation in 100000000 times. This execution done by multiple threads (use a different number of cores) also using the ThreadPool. We have used a different number of cores. And have recorded the result in both approaches for machine-B. The result is in figure-39, 40.



	2	4	6	8	10	12	14	16	18	20	22	24
Multi Thread	3203	1713	1191	1002	8987	7919	7247	6527	6549	5904	5502	5244

Figure 39: Use Static multithreading for calculating simple equation ucores different number of core.



	2	4	6	8	10	12	14	16	18	20	22	24
Tread Pool	3203	1710	1191	1032	8760	7888	7094	6330	5704	5303	4850	4514

Figure 40: Use ThreadPool example for calculating simple equation using different number of cores

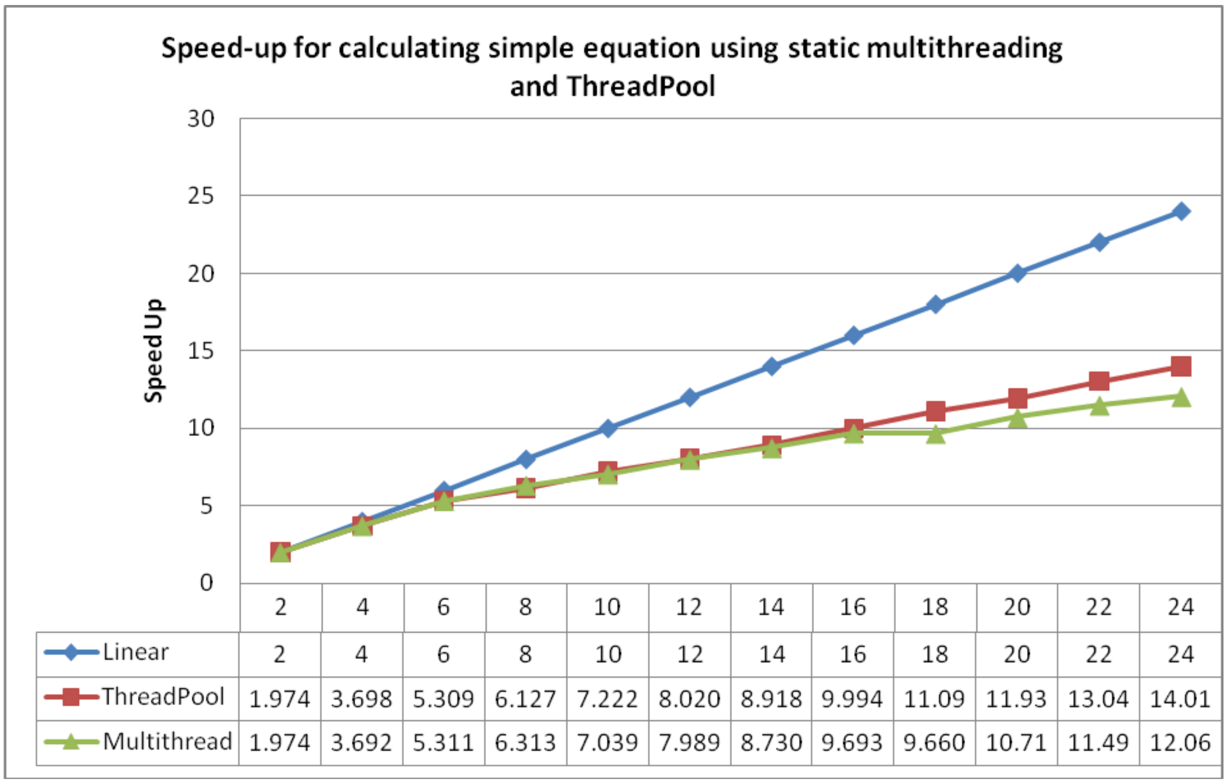
From the above experimental results we have calculated the execution speed up according to execution times. The speed-up of different cores has been calculated corresponding to the execution time of sequential execution.
Speed-up is defined by the following formula [46]:

$$S_p = \frac{T_1}{T_p}$$

Where:

- p is the number of processors
- T1 is the execution time of the sequential algorithm
- Tp is the execution time of the parallel algorithm with p processors

Here the execution time of Sequential algorithm T1= 63268 Milliseconds. So according to the equation the speed-up of Multithread approaches and for 2 cores Sp=(T1/Tp)=(63268/ 32036)=1.9749. Ideally for 2 cores the speed-up supposes to be 2 times. The speed should increase linearly. Theoretically the speed-up would not be linear. According to the Amdahl's law every program might have a small sequential part which cannot be parallelized [10]. Also in our experiment the speed-up is not linear. The following figure-41 shows the speed-up curve for two approaches. The above all experiments show different performance in different cases and this achievement addressed our research questions.



	2	4	6	8	10	12	14	16	18	20	22	24
Linear	2	4	6	8	10	12	14	16	18	20	22	24
ThreadPool	1.974	3.698	5.309	6.127	7.222	8.020	8.918	9.994	11.09	11.93	13.04	14.01
Multithread	1.974	3.692	5.311	6.313	7.039	7.989	8.730	9.693	9.660	10.71	11.49	12.06

Figure 41: Speed-up for calculating simple equation using static multithreading and ThreadPool

6 DISCUSSION AND CONCLUSION

6.1 Discussion

According to several experiments, in many cases the parallelism shows performance improvement over traditional sequential execution. But in some cases the parallel loop works slower than sequential execution. In parallel execution there is so much complexity which does not exist in sequential execution. All of the parallel constructs and APIs of .NET Framework 4.0 have great contribution for making parallel applications. In our small number of experiments we have got the good performance of parallel application corresponding sequential execution in terms of execution time. We have tried to compare the performance of different parallel constructs and APIs with sequential execution. Sometimes we have got better performance. Beside the achievement there are some bad experiences as well. From those bad experiences and studies of parallel application performances, we are discussing some issues which should we keep in mind for improving the performance of parallel application.

6.1.1 Inappropriate uses of threading

Threading use the CPU resources. Creating, destroying and Scheduling between threads is the cost factor. So, because of inappropriate implementations sometime multithreading will not show the better performance. If there is heavy I/O waiting then many worker threads cannot work simultaneously [11]. In the previous chapter the experiment-6 focuses a little bit on this idea.

6.1.2 Oversubscription of threads

Sometimes the over subscriptions of thread over CPU resources slow down the execution [56]. The benefits of parallelization are also depends on the number of processors on the computer. For example, if one has 2 cores which are capable of handling 4 threads simultaneously and the program creates more than 4 threads. Then switching and synchronization between threads will be an overhead for the CPU. Processor has to expense some time for that. In that case if one creates fixed number of parallelism and the target system has enough processors to handle that number of threads then it will be better. In experiment-5 we have got an effect of oversubscription. Oversubscription of threads did not show the better performance in Machine-B(2 cores).

6.1.3 Unbalanced workload distribution

In many cases, unbalanced workload distribution becomes an issue of slow down execution [4] [14]. Mainly it happens in static work partitioning. If a program creates 3 threads in which 1st thread is responsible for 2 second jobs the 2nd one is responsible for 3 seconds jobs but the 3rd one for 20 seconds jobs. Then the 1st and 2nd will complete their works and main program needs more time to finish. In experiment-4 the three tasks have been executed by different threads in parallel. But the assigned works of three tasks were unbalanced.

6.1.4 Low workload

If the system has enough parallel power but the total numbers of data or works is not so high then each CPU no need to do enough work. For large workload the time of synchronization between threads is negligible. But for the small amount of data the synchronization cost of CPU can be so expensive (relatively) than the main execution. So the performance might be slow down. This issue has been addressed in experiment-7.

6.1.5 Using shared memory location

In sequential execution it is quite easy to access static variables. But in parallel execution it is a complex task. Lock and synchronization can hurt to performance. So it is recommended to avoid or less use of shared memory [56].

6.1.6 Low physical memory of system

If we think about the parallel processing performance, then the uses of physical memory are another concern. In many parallel processing applications need huge amounts of memory [57]. If physical memory of computer is low and the program needs large number of spaces for executing parallel application then the memory swapping is an overhead for the processor. In that case performance might be downward.

6.1.7 Inappropriate use of parallel application patterns

Appropriate use of parallel constructs and APIs is another challenge for making an efficient parallel application. Developers have to recognize the appropriate API and right way for executing the task. We can think about the static and dynamic partitioning (already discussed in section-3.6). For the particular problem the static partition may be suitable and sometimes dynamic partitioning is suitable for load balancing. The experiment-1 shows the example of static partitioning. For data parallelism we can use the Parallel.For, Parallel.ForEach, PLINQ etc. But the Parallel.Invoke is suitable for task parallelism. The experiment-4 shows the example of task parallelism. The Parallel.ForEach or PLINQ is suitable for unknown size of dataset.

At the beginning of thesis, had some research questions. The whole thesis discussed different points and experiments and realized the answers of those research questions.
The research sub question (a) was “Which parallel constructs and APIs we can use for making numbers of appropriate simple parallel applications?”.
We have described details about different parallel constructs and APIs in chapter-3 for understanding the uses of those patterns in different situation. Also we have done many experiments using those constructs and APIs. From our experiment we have illustrated some notable experiments in the previous chapter. In some particular problems the static partition may be suitable and sometimes dynamic partitioning for proper load balancing. The experiment-1 shows how we can make static partition in parallel application using parallel constructs. For data parallelism and task parallelism we have different APIs. The Parallel.For, Parallel.ForEach, PLINQ etc. can be used for data parallelism and the Parallel.Invoke is suitable for task parallelism. The experiment-4 illustrated the task parallelism example using Parallel.Invoke. The experiment-3 and experiment-6 demonstrated Parallel.ForEach and PLINQ which are suitable for unknown size of dataset. The experiment-2 shows the implementation of Parallel.For which is the basic constructs for making parallel application.

The PLINQ has different methods like .AsParallel, .WithDegreeOfParallalism etc which help to make efficient parallel application. And the above discussions address the research sub question (a).

The research sub question (b) was “Which performance issues involve for improving the performance of parallel application constructed by parallel extensions?”
In this chapter we have discussed different issues for improving the performance of parallel application and some of them support our experiments. The experiment-5 shows the parallel application performance when the number of thread increase in different parallel architecture. The experiment-6 evaluated the PLINQ performance using .AsParallel () and .WithDegreeOfParallelism (). And this focused that the less number of threads will create less I/O request and it might reduce the I/O overlapping. The experiment 7 illustrated PLINQ which shows parallel application performance for different workload (different data range) on CPU. The experiences and discussions address the research sub question (b).

The research sub question (c) was “What improvements of CPU resource utilization can be made by parallel executions?”
During our experiment we have seen that the sequential execution does not use the full available CPU resources. Most of the parts of CPU have to stay idle. But mostly the parallel application has tried to use maximum resources of different cores. In our 1st experiment, figure-27 and 28 demonstrated the CPU uses the history of sequential and parallel execution.

The main research was “*What is the experimental performance of applications using parallel extensions with corresponding sequential versions in terms of different workload and different number of CPU resources used in some implementations using two types of parallel architecture*?”

From all experiments in chapter-5 we have got different performances for different parallel constructs and APIs. Rather than some exceptions of parallel application performances over sequential program are proper CPU utilization and less execution time. According to the Amdahl’s law the speed-up of parallel execution is not exactly linear [10]. In experiment-9 we have discussed the speed-up for an example using static multithreading. And our experimental speed-up follows the Amdahl’s law. All of our experiments, above discussions and answers of all research sub questions address the main research question.

6.2 Conclusion

The parallel computing is a vast topic for present as well as next generation computation in the world. There are so many complex computational problems in front of us. So this is a great challenge for us to overcome the problem of parallel computing making efficient parallel applications in a multiprocessor environment and solve those worlds complex problems in a very efficient way.
This thesis paper has been focused mainly on the different terminology for understanding parallel programming (Chapter 2), finding the different design patterns of parallel application in .NET Framework 4 (Chapter 3), finding performance factors and evaluate the performance of parallel application over sequential approach for different parallel application patterns (Chapter 4,5).

Understanding the methodology of parallelism and design patterns of parallel programming will help one to make an application with proper parallelism for solving any complex problem. The discussion of patterns of this paper helps one for understanding and gathering enough idea about parallel application. Finally uses of the new and different parallelization support.NET Framework 4 which helps to build high featured parallel applications. And solve problems in a cost effective way.

Performance is a main issue of parallel applications. That performance depends on choosing the pattern and proper way of implementation. The knowledge of proper partitioning, avoiding oversubscription, proper workload balancing, the proper way of memory sharing etc. makes parallel application's performance higher. Also applications on an operating system are not able to 100% use of the CPU resources. The operating system is responsible for fair scheduling of different current threads in a system. When program creates some threads for execution there might have some other threads in a queue of operating system. So it is quiet impossible to engage the CPU resource 100%.

In our thesis we have tried to measure the performance of parallel applications with different parallelism criteria using different important parallel constructs and APIs of .NET Framework 4.0. We have tried to complete our experiments with in limited LAB resources. On the basis of experimental result we have discussed some important performance issues as well. The discussions addressed the all research questions. So we think that our thesis will be helpful for the academia as well as professionals for further research or making complex parallel applications in a practical field. This thesis will help as a foundation for further research. Using this foundation the researcher can find more depth knowledge about patterns. Professionals can find a more efficient way of parallel programming for solving the upcoming complex problems in the world.

References

1. Carriero, N., and Gelernter, D. (1988), How to Write Parallel Programs. A Guide to the Perplexed. Yale University, Department of Computer Science, New Heaven, Connecticut.
2. Chandy, K. M., and Taylor, S. (1992), An Introduction to Parallel Programming. Jones and Bartlett Publishers, Inc., Boston.
3. Darlington, J. and To, H. W. (1993), Building Parallel Applications without Programming. Department of Computing, Imperial College. United Kingdom. In Abstract Machine Models, Leeds.
4. SP Parallel Programming Workshop parallel programming introduction, http://www.mhpcc.edu/training/workshop/parallel_intro/MAIN.html, access 22-11-2010
5. Blaise Barney, “introduction to parallel computing”, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/, access 20-11-2010
6. Paul E. McKenney, Linux Technology Center IBM Beaverton,” Is Parallel Programming Hard, And, If So, What Can You Do About It?”, March 23, 2010.
7. Grama, A. Gupta, G. Karypis, and V. Kumar, “Introduction to Parallel Computing . (2nd edition);”, Addison-Wesley (2003).
8. Maurice Herlihy, Sergio Rajsbaum, Mark R. Tuttle, “Unifying Synchronous and Asynchronous Message-Passing Models”, poDc 98 Puerto Vallarta Mexico, ACM 1998 O-89791-977-7/98/6.
9. William Carlson-IDA CCS, Robert Numrich- Cray, Inc., “Parallel programming using a distributed shared memory model”, 2001.
10. Azali Bin Saudi,” PARALLEL COMPUTING”, Universiti Malaysia Sabah,April 2008.
11. Joseph Albahari; Ben Albahari, “C# 4.0 in a Nutshell”, Fourth Edition, Chapters 21 and 22, Publisher: O'Reilly Media, Inc. January 26, 2010.
12. Threads per Processor, http://stackoverflow.com/questions/215236/threads-per-processor, access 28-11-2010
13. Andrew Troelsen, “Pro C# 2010 and the .net 4 Platform Fifth Edition (chapter 19)”, Copyright © 2010 by Andrew Troelsen, Apress, USA
14. Stephen Toub, “PATTERNS OF PARALLEL PROGRAMMING”, July 1, 2010, ©2010 Microsoft Corporation.
15. Y.-K. Chen, X. Tian, S. Ge, and M. Girkar. Towards efficient multi-level threading of H.264 encoder on Intel hyperthreading architectures. Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, April 2004.
16. D. Lea. Concurrent Programming in Java: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.
17. S. MacDonald. “From Patterns to Frameworks to Parallel Programs.”, PhD thesis, Department of Computing Science, University of Alberta, November 2001. Available at www.cs.ualberta.ca/∼systems.
18. “ConcurrentQueue<T> Class”, MSDN, http://msdn.microsoft.com/en-us/library/dd267265.aspx, access 24-dec-2010.
19. “Partitioning in PLINQ”, http://blogs.msdn.com/b/pfxteam/archive/2009/05/28/9648672.aspx, access 18-dec-2010
20. T. Mattson, B. Sanders, and B. Massingill. Patterns for parallel programming. Addison-Wesley Professional, 2004.
21. E. G. Richardson. H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia. John Wiley and Sons, 2003.

22. Rodriguez, A. Gonzalez, and M. P. Malumbres. Hierarchical parallelization of an h.264/avc video encoder. In PARELEC ’06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering, pages 363–368, Washington, DC, USA, 2006. IEEE Computer Society.
23. S. Siu, M. D. Simone, D. Goswami, and A. Singh. Design patterns for parallel programming, 1996.
24. M. Snir, S. Otto, S. Hess-Lederman, D. Walker, and J. Dongarra. MPI: The Complete Reference. MIT Press, 1996.
25. Michael J. Flynn. “Some Computer Organizations and their Effectiveness.” IEEE Transactions on Computers, 21(9):948–960, 1972.
26. M. J. Flynn. Very high-speed computing systems. pages 519–527, 2000.
27. Smith, C. U. and Williams, L. G. (1993), Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives. IEEE Transactions on Software Engineering, Vol. 19, No. 7.
28. ManualResetEvent Class, URL: http://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(v=vs.71).aspx, Access 14-12-2010
29. Stephen Toub, ”Aggregating Exceptions”, http://msdn.microsoft.com/sv-se/magazine/ee321571(en-us).aspx, access 11-11-2010.
30. Pancake, C. M. (1996), Is Parallelism for You? Oregon State University. Originally published in Computational Science and Engineering, Vol. 3, No. 2.
31. Pancake, C. M., and Bergmark, D. (1990), Do Parallel Languages Respond to the Needs of Scientific Programmers? Computer Magazine, IEEE Computer Society.
32. Data Parallelism (Task Parallel Library) .NET Framework 4, http://msdn.microsoft.com/en-us/library/dd537608.aspx, access 25-11-2010
33. Parallel Programming in the .NET Framework, http://msdn.microsoft.com/en-us/library/dd460693.aspx, access 20-11-2010
34. Introduction to PLINQ, http://msdn.microsoft.com/en-us/library/dd997425.aspx, access 29-11-2010.
35. Peter Bromberg, “Task Parallelism in C# 4.0 with System.Threading.Tasks“, http://www.eggheadcafe.com/print.aspx, access 23-11-2010
36. Task.Factory Property, .NET Framework 4, http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.factory.aspx, access 16-12-2010
37. System.Collections.Concurrent Namespace, .NET Framework 4, http://msdn.microsoft.com/en-us/library/dd287108.aspx, access 16-12-2010
38. Vivek Sarkar, ”Introduction to Parallel Computing”, Department of Computer Science Rice University, 2008.
39. Hahn Kim, Julia Mullen and Jeremy Kepner, ”Introduction to Parallel Programming and pMatlab v2.0”, MIT Lincoln Laboratory, Lexington, MA 02420
40. “Introducing ConcurrentStack < T >”, http://blogs.msdn.com/b/pfxteam/archive/2008/06/18/8614596.aspx, access 21-dec-2010
41. Eric Aubanel, “Resource-Aware Load Balancing of Parallel Applications”, Faculty of Computer Science, University of New Brunswick Fredericton, NB Canada, February 2008
42. Hans Wolfgang Loidl, “Granularity in large scale parallel functional programming”, University of Glasgow, March 1998.
43. Suresh Paldia, “Parallel.Invoke in .Net Framework 4”, C# Corner, Nov-15-2010, http://www.c-sharpcorner.com/UploadFile/61b832/4215/, access 08-dec-2010.
44. JunSeong Kim and David J. Lilja,” Characterization of Communication Patterns in Message-Passing Parallel Scientific Application Programs”, october 1997.

45. David S. Wise and Joshua Walgenbach, "Static and Dynamic Partitioning of Pointers as Links and Threads Technical Report 437", Computer Science Department, Indiana University Bloomington, Indiana 47405-4101 USA Copyright c 1996 by the Association for Computing Machinery, Inc.
46. Herbert Schildt, "C# 4. 0 the Complete Reference (Chapter 23,24)", Copyright © 2010 by The McGraw-Hill Companies.
47. Wikipedia, "Speedup", http://en.wikipedia.org/wiki/Speedup, access 18-12-2010
48. David E. Culler, Jaswinder Pal Shingh with Anoop Gupta, "Parallel Computer Architecture: hardware/Software Approach", Copyright © 1999 Morgan Kaufmann Publishers, Inc.
49. Mark N. K. Saunders, Adrian Thornhill, (2003) "Organisational justice, trust and the management of change: An exploration", Personnel Review, Vol. 32 Iss: 3, pp.360 – 375.
50. Jennifer Marsman, "Parallel Programming in .NET 4.0: PLINQ", http://blogs.msdn.com/b/jennifer/archive/2010/06/21/parallel-programming-in-net-4-0-plinq.aspx, access 19-12-2010
51. Jidong Zhai, Wenguang Chen, Weimin Zheng "PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node", PPoPP'10, January 9–14, 2010, Bangalore, India, Copyright © 2010 ACM 978-1-60558-708-0/10/01
52. "How to: Use Parallel.Invoke to Execute Parallel Operations", http://msdn.microsoft.com/en-us/library/dd460705.aspx , access 22-dec-2010.
53. Indranil Chatterjee, "LINQ executed in Parallel (PLINQ)", http://www.eggheadcafe.com/tutorials/aspnet/042c0b06-95f2-4944-9b52-46be6eeb3e7d/linq-executed-in-parallel-plinq.aspx, access 20-dec-2010.
54. "Introduction to PLINQ", http://msdn.microsoft.com/en-us/library/dd997425.aspx, access 22-dec-2010.
55. JasonShort, "Speed up blocking functions with PLINQ", http://www.codeproject.com/Articles/153694/Speed-up-blocking-functions-with-PLINQ.aspx, access 22-dec-2010.
56. "Potential Pitfalls in Data and Task Parallelism", http://msdn.microsoft.com/en-us/library/dd997392.aspx, access 15-12-2010.
57. Norman Matloff, "Programming on Parallel Machines", University of California, Davis1, http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf, access 25-dec-2010.
58. Wikipedia, "Thread (computer science)", http://en.wikipedia.org/wiki/ Thread_ (computer_science) , access 05-dec-2010
59. Endika Bengoetxea, "On processes and threads: synchronization and communication in parallel programs", PhD Thesis, 2002.
60. "How to: Use a Thread Pool (C# Programming Guide)", http://msdn.microsoft.com/en-us/library/3dasc8as(v=vs.80).aspx, access 20-dec-2010.
61. "TaskClass",http://msdn.microsoft.com/en-us/library/system.threading.tasks.task.aspx, access 23-12-2010
62. "How to: Write a Parallel.For Loop That Has Thread-Local Variables", http://msdn.microsoft.com/en-us/library/dd460703.aspx, access 25-Nov-2010
63. A.Sethi, "Parallel Programming Framework 4.0 Part 2 – Parallel.ForEach", Towards Next, http://towardsnext.wordpress.com/2010/08/17/, access 06-dec-2010.
64. Kourosh Gharachorloo, "MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS ", Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, 1995.