İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

DISTRIBUTION OF OBJECTS ON MULTICORE PROCESSORS

M.Sc. Thesis by
Tolga KAYAR

Department : Computer Engineering

Programme : Computer Engineering

OCTOBER 2009

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF SCIENCE AND TECHNOLOGY

DISTRIBUTION OF OBJECTS ON MULTICORE PROCESSORS

M.Sc. Thesis by
Tolga KAYAR
(504071532)

Date of submission : 29 September 2009
Date of defence examination: 02 October 2009

Supervisor (Chairman) : Assis. Prof. Dr. Feza BUZLUCA (ITU)
Members of the Examining Committee : Prof. Dr. Nadia Erdoğan (ITU)
Prof. Dr. Coşkun Sönmez (YTU)

OCTOBER 2009

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

ÇOK ÇEKİRDEKLİ İŞLEMCİLERDE NESNELERİN DAĞITILMASI

YÜKSEK LİSANS TEZİ

Tolga KAYAR
(504071532)

Tezin Enstitüye Verildiği Tarih : 29 Eylül 2009
Tezin Savunulduğu Tarih : 02 Ekim 2009

Tez Danışmanı : Yrd. Doç. Dr. Feza BUZLUCA (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. Nadia Erdoğan (ITU)
Prof. Dr. Coşkun Sönmez (YTU)

EKİM 2009

FOREWORD

I would like to express my deep appreciation and thanks for my advisor. This work is supported by ITU Institute of Science and Technology. In addition I would like to express my deep appreciation and thanks for TUBITAK for their support.

October 2009

Tolga Kayar

Computer Engineer

TABLE OF CONTENTS

Page

ABBREVIATIONS ..... **ix**
LIST OF TABLES ..... **xi**
LIST OF FIGURES ..... **xiii**
SUMMARY ..... **xv**
ÖZET ..... **XVII**
1. INTRODUCTION **1**
1.1 Purpose of the Thesis 2
1.2 Background 2
1.3 Hypothesis 7
2. DEFINITIONS **9**
2.1 Multiprocessor 9
2.2 Multicore 10
2.3 Cache 11
2.4 Scheduler 12
3. INVESTIGATING PERFORMANCE OF PARALLEL OBJECTS ON MULTICORE SYSTEMS **13**
3.1 Test Environment 13
3.2 Test Cases and Algorithm 14
3.3 Experiments 16
3.3.1 Investigating method call count effect 16
3.3.1.1 Experiment1: 500 method call 16
3.3.1.2 Experiment2: 1000 method call 17
3.3.1.3 Experiment3: 2000 method call 17
3.3.1.4 Experiment4: 4000 method call 18
3.3.1.5 Experiment5: 8000 method call 18
3.3.1.6 Experiment6: 16000 method call 19
3.3.1.7 Conclusion 19
3.3.2 Investigating attribute size effect 21
3.3.2.1 Experiment1: 500 array size 21
3.3.2.2 Experiment2: 1000 array size 22
3.3.2.3 Experiment3: 2000 array size 22
3.3.2.4 Experiment4: 4000 array size 23
3.3.2.5 Experiment5: 8000 array size 24
3.3.2.6 Experiment6: 16000 array size 24
3.3.2.7 Conclusion 25
4. DISTRIBUTION OF OBJECTS ON MULTICORE SYSTEMS **27**
4.1 Proposed Algorithm 27
4.2 Experiments 30
4.2.1 Investigating method call count effect on performance for 15 objects 31
4.2.1.1 Experiment 1: 3000 method call 31
4.2.1.2 Experiment 2: 5000 method call 31

4.2.1.3 Experiment 3: 8000 method call 32
4.2.2 Investigating method call count effect on performance for 30 objects 32
4.2.2.1 Experiment 1: 3000 method call 33
4.2.2.2 Experiment 2: 5000 method call 33
4.2.2.3 Experiment 3: 8000 method call 34
4.2.3 Investigating effect of size of the object on performance for 15 objects .. 34
4.2.3.1 Experiment 1: 500 array size 35
4.2.3.2 Experiment 2: 1000 array size 35
4.2.3.3 Experiment 3: 2000 array size 36
4.2.4 Investigating effect of size of the object on performance for 30 objects .. 36
4.2.4.1 Experiment 1: 500 array size 36
4.2.4.2 Experiment 2: 1000 array size 37
4.2.4.3 Experiment 2: 2000 array size 38
4.3 Conclusion 38
5. CONCLUSION AND RECOMMENDATIONS 43
REFERENCES 45
CURRICULUM VITA 47

ABBREVIATIONS

CPU **:** Central Processing Unit
MHz **:** Megahertz
MP **:** Multiprocessing

LIST OF TABLES

Page

Table 3.1: Duration of the program with 500 method call, 1000 array size……..16
Table 3.2: Duration of the program with 1000 method call, 1000 array size……17
Table 3.3: Duration of the program with 2000 method call, 1000 array size……17
Table 3.4: Duration of the program with 4000 method call, 1000 array size……18
Table 3.5: Duration of the program with 8000 method call, 1000 array size……18
Table 3.6: Duration of the program with 16000 method call, 1000 array size……..19
Table 3.7: Duration of the program with 500 array size, 1000 method call……..21
Table 3.8: Duration of the program with 1000 array size, 1000 method call……22
Table 3.9: Duration of the program with 2000 array size, 1000 method call……22
Table 3.10: Duration of the program with 4000 array size, 1000 method call……..23
Table 3.11: Duration of the program with 8000 array size, 1000 method call……..24
Table 3.12: Duration of the program with 16000 array size, 1000 method call……24
Table 4.1: Duration of the program with 15 objects and 3000 method calls…….31
Table 4.2: Duration of the program with 15 objects and 5000 method calls…….31
Table 4.3: Duration of the program with 15 objects and 8000 method calls…….32
Table 4.4: Duration of the program with 30 objects and 3000 method calls…….33
Table 4.5: Duration of the program with 30 objects and 5000 method calls…….33
Table 4.6: Duration of the program with 30 objects and 8000 method calls…….34
Table 4.7: Duration of the program with 15 objects and array size 500……35
Table 4.8: Duration of the program with 15 objects and array size 1000……..35
Table 4.9: Duration of the program with 15 objects and array size 2000……..36
Table 4.10: Duration of the program with 30 objects and array size 500……..37
Table 4.11: Duration of the program with 30 objects and array size 1000……....37
Table 4.12: Duration of the program with 30 objects and array size 2000……38

LIST OF FIGURES

Page

Figure 1.1 : The dualcore architecture. 1
Figure 1.2 : The level 2 cache dualcore architecture. 2
Figure 1.3 : The polyhedral model. 3
Figure 1.4 : Unfair cache sharing. 4
Figure 1.5 : CPU latency comparison. 4
Figure 1.6 : Cache-fair algorithm structure. 5
Figure 1.7 : A two-processor Pfair schedule of a set of five tasks. 6
Figure 1.8 : Peak Performance Policy. 6
Figure 1.9 : Power Saving Policy. 7
Figure 2.1 : Multiprocessor Architecture 9
Figure 2.2 : Homogeneous multicore and heterogeneous multicore CPU 10
Figure 2.3 : L1 and L2 cache of a dual core processor 11
Figure 3.1 : Test computer CPU structure 13
Figure 3.2 : Test program flow chart 14
Figure 3.3 : Test Cases a: same core, b: same processor,different core, c: different processor and different core 15
Figure 3.4 : Effect of method call count on the performance 20
Figure 3.5 : Effect of method call count on the performance first part detailed 21
Figure 3.6 : Effect of size of the object on the performance 25
Figure 3.7 : Effect of size of the object on the performance first part detailed 26
Figure 4.1 : Main fuction flowchart 28
Figure 4.2 : Assign core algorithm flowchart 29
Figure 4.3 : Effect of method call count on the performance for 15 objects 39
Figure 4.4 : Effect of method call count on the performance for 30 objects 40
Figure 4.5 : Effect of size of the object on the performance for 15 objects 41
Figure 4.6 : Effect of size of the object on the performance for 30 objects 42

DISTRIBUTION OF OBJECTS ON MULTICORE PROCESSORS

SUMMARY

In the last two decades multicore multiprocessors have become more popular. Multicore multiprocessors are not only used in servers or high capacity computers, they are also used in personal computers[13]. According to Moore's Law, the number of transistors will double every two years. However increase in the number of transistors slows down, because of power needed[4]. Thus multicore era begins.

Software development on multicore processors is an interesting research issue. There are lots of researches about multicore programming. Development on multicore is more complex and time consuming issue then single core. There are differences between single core multiprocessors and multicore multiprocessors on development styles. Current softwares, that are not designed to work on multicore CPUs, do not work efficiently on multicore CPUs. This problem reveals an adaptation process that is not easy. In this process the most significant responsibility is on software developers. There are books and lots of other documents about multicore programming pointing this issue.

Object oriented programming is popular and prefered programming method. Software development on multicore with object oriented programming is another popular issue nowadays. There are tools like OpenMP and Cilk++ to develop parallel object oriented software. They provide parallelism with multithreading. However programmer is expected to write his/her code parallel with commands provided by these tools. The other method for parallelism is automatic (implicit) parallelism. In this method programmer writes his/her code without thinking parallelism, then on compile level program is adapted to run parallel on runtime. However automatic parallelism is not made on object level, it is made on function level. Since object oriented programming consists of objects not functions, function level parallelism is not suitable.

In this thesis we propose an algorithm for the distribution of objects to cores on compile level to provide better performance and parallelism. Distribution of objects to cores means creating an object on specific core and calling its methods on the core which the object is created on. We show that our algorithm is more efficient and meaningful than random distribution or distribution to the first empty core methods. This is because of level one and two caches of processors. For each object staying in the same processor, even in the same core results better use of caches and increases performance due to the attributes and shared data of objects among cores of the processors.

ÇOK ÇEKİRDEKLİ İŞLEMCİLERDE NESNELERİN DAĞITILMASI

ÖZET

Son yıllarda çok çekirdekli işlemciler daha popüler hale geldi. Çok çekirdekli işlemciler sadece sunucularda ve yüksek kapasiteli bilgisayarlarda değil evlerdeki kişisel bilgisayarlarda da kullanılmaya başlandı[13]. Moore Yasası'na göre tranzistör sayısı her iki yılda iki katına çıkacak. Fakat tranzistör sayısındaki bu artış güç ihtiyacı sebebiyle hız kesmiş görünüyor[4]. Böylece çok çekirdekli işlemcilerin çağı başlamış oldu.

Çok çekirdekli işlemcilerde yazılım geliştirme son zamanlarda üzerinde durulan konulardan bir tanesidir. Çok çekirdekli işlemcilerde programlama üzerine yapılmış birçok araştırma bulunmaktadır. Bu işlemcilerde yazılım geliştirmek hem zaman alan hem de karmaşık bir süreç. Eski tip işlemcilerle çok çekirdekli işlemciler arasında yazılım geliştirme açısından farklılıklar bulunmaktadır. Şu anda kullanımda olan ve çok çekirdekli işlemcilere uygun bir biçimde geliştirilmemiş yazılımlar bu işlemciler üzerinde verimli çalışmamaktadır. Bu sorun hiçte kolay olmayan bir adaptasyon süreci ortaya çıkarmaktadır. Bu süreçte en büyük iş programcılara düşmektedir. Bu sorunu işaret eden birçok kitap ve yayın bulunmaktadır.

Yazılım geliştirme sürecinde şu anda en çok tercih edilen yöntem nesneye dayalı programlamadır. Günümüzde üzerinde durulan bir diğer önemli ve popüler konu çok çekirdekli işlemcilerde nesneye dayalı yazılım geliştirmedir. Paralel nesneye dayalı yazılım geliştirmek için Cilk++ ve OpenMP gibi araçlar bulunmaktadır. Bu araçlar çok iplikli yapı ile paralelliği sağlamaktadır. Bu paralelleştirmeden faydalanabilmek için programcının bu araçların sağladığı özel komutları kullanması gerekmektedir. Paralelleştirme için kullanılan bir diğer yöntem otomatik paralelleştirmedir. Bu metoda göre programcı kodunu yazarken paralelliği düşünmeden yazar, daha sonra derleyici düzeyinde program çalışma zamanında paralel çalışması için adapte edilir. Fakat otomatik paralelleştirme nesneler düzeyinde değil fonksiyonlar düzeyinde yapılmaktadır. Nesneye dayalı programlama foksiyonlardan değil nesnelerden oluştuğundan dolayı fonksiyonlar düzeyinde paralelleştirme tam olarak uygun olmamaktadır.

Bu tezde nesnelerin çekirdeklere derleme seviyesinde dağıtılmasına ilişkin önerilen algoritmayla daha iyi performans ve paralellik sağlanmıştır. Nesneleri çekirdeklere dağıtmak, nesneyi önceden belirli bir çekirdekte yaratıp daha sonra yapılan tüm yordam çağrılarını nesnenin yaratıldığı çekirdeğe yapmaktır. Önerilen algoritmanın nesneleri rastgele dağıtmaktan veya ilk boş bulunan çekirdeğe atmaktan daha iyi ve anlamlı olduğu gösterilmiştir. Bu sonuç birinci ve ikinci seviye önbellekten kaynaklanmaktadır. Nesnelerin niteliklerinden ve paylaşılan verilerinden dolayı nesnelerin aynı işlemcide hatta aynı çekirdekte kalması performansı arttırmış ve önbelleklerin daha iyi kullanılmasını sağlamıştır.

1. INTRODUCTION

After the invention of computer, a new era began. Computers are used for every purpose and now they are almost at every house[16]. Multiprocessors are brains of computers and they are very significant because of this reason. Since Intel 4004 (first CPU with no cache, 2250 transistor count, 0.74MHz internal clock invented in 1971 [13]) development of CPU never stopped. In 1965 Intel co-founder Gordon E. Moore said that the number of the transistors will double every two years, and he is almost right about that. However for the last decade producers chose developing multicore CPUs instead of traditional single core CPUs, because increasing the number of the transistors on a single chip become harder and ineffective. This behaviour changed everything. It is clear that the future of CPUs is multicore.

Multicore CPUs provide multithreading. These CPUs support parallelism more than single core CPUs. Each core can execute threads independently that is the reason that multicore CPUs provide support for multithreading. The multicore architecture can be seen on Figure 2.1[12]. This architecture shows dual core processor with level one cache. Level one cache is the cache that belongs to each core. The number of level one cache is equal to the number of cores[10].

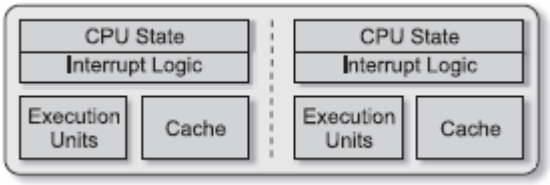


Figure 1.1 : The dualcore architecture.

Multicore CPUs can also have level two cache that is shared among cores[12]. This cache level can be seen in Figure 2.2. In this thesis level one cache and level two cache advantages and disadvantages will be shown among the experiments made[14].

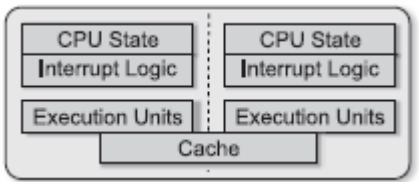


Figure 1.2 : The level 2 cache dualcore architecture.

Multicore CPUs are suitable for parallelism, however software developped by programmers should be convenient, because multicore programming is different from serial programming. It requires multithreading support to run efficiently on multicore CPUs. Anyway it is not meaningful to use these processors. This issue is very popular nowadays. Lots of papers, books and other types of documents published and researches made about this issue. However these publications are not about distribution of objects, they are about distribution of functions.

1.1 Purpose of the Thesis

The main objective of this study is to show that distribution of objects to cores of multiprocessors by an algorithm that enforces object to stay in the processor even in the core which it is created on provides better performance and parallelism than random distribution or distribution to the first empty core methods. This is because of level one and two caches of processors. The other objective of this study is to show that the number of attributes and method calls made have an effect on performance.

1.2 Background

Software developers should be aware of multithreading to use multicore CPUs efficiently. Programming tools help developers about this issue such as Cilk++,OpenMP etc., however to use these tools programmer should use their syntax and special commands. Furthermore programmer should have sense about where to use parallelism. This is difficult and a time consuming matter.

The other issue about multithreading is automatic parallelism[5]. This means compile level parallelism. Programmer develops software without thinking parallel and then compiler makes decisions about which parts of code will run parallel[15]. First automatic parallelization model sample was Illiac IV. Then some commercial versions of this model invented. For instance, Intel's Production Compiler. However there is not an efficient method for automatic parallelization for multicore processors.

Alternative methods proposed about automatic parallelization issue. One of them is polyhedral model. According to this model loop nests are modelled and integer value is attached to each statement in loops. In this model dependency test made, after functional decomposition and transformation compile of code completed. This process can be seen in Figure 1.3[8]. These automatic parallelization methods use function decompositions, and optimizations on loops etc.

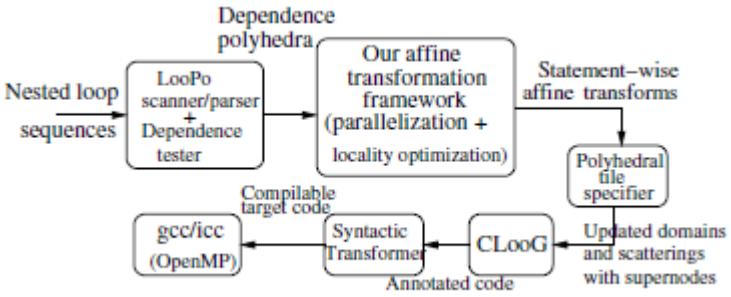


Figure 1.3 : The polyhedral model.

There are proposed scheduling algorithms for multicore processors to reach better performance and parallelism. Old scheduling algorithms for single core processors are not suitable for multicore systems. CPU usage and priority mechanism for threads are main problems for old fashion algorithms and they have to be adapted to work on multicore processors. Lots of researches made for this issue. One of them is "Cache-Fair Thread Scheduling for Multicore Processors". In this research it is emphasized that cache sharing depends on the cache needs of the co-runners, and this results in unfair cache sharing. This can be seen in Figure 1.4. Co-runners mean threads working at the same time. It is indicated that co-runners effect cache miss rate.

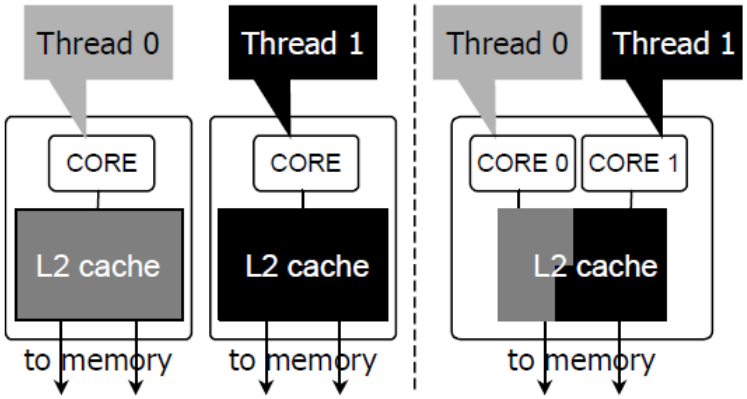


Figure 1.4 : Unfair cache sharing.

In this algorithm L2 (level 2) cache allocation must be considered. It is more complicated to determine cache allocation compared to shared memory allocation. Runtime statistics and analytical models are used to determine thread's performance and then decision made to arrange execution times of threads.

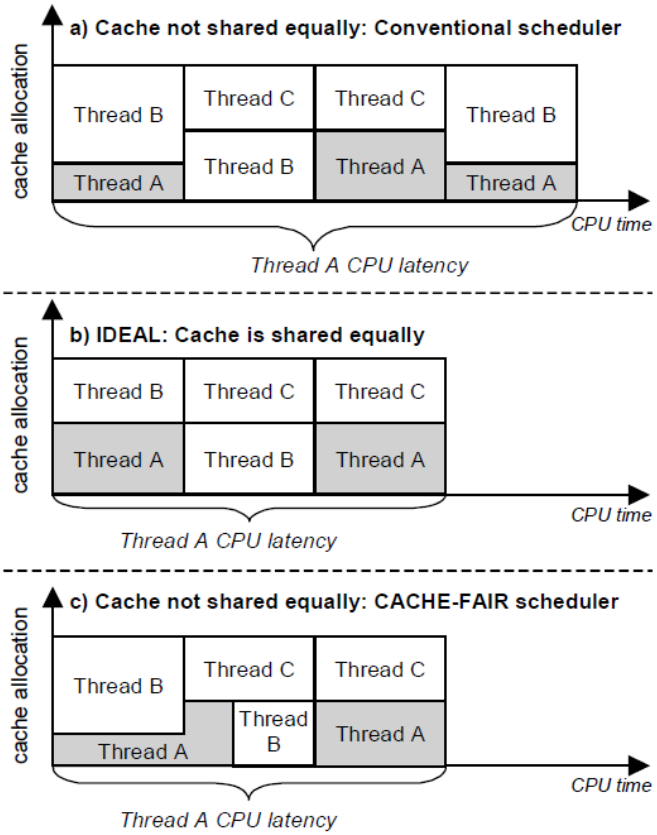


Figure 1.5 : CPU latency comparison.

In Figure 1.5, first graphic shows conventional scheduler cache allocation and CPU latency times, second graphic shows ideal cache allocation and the last graphic shows cache-fair scheduler cache allocation. The structure of the cache-fair scheduler algorithm can be seen in Figure 1.6[2].

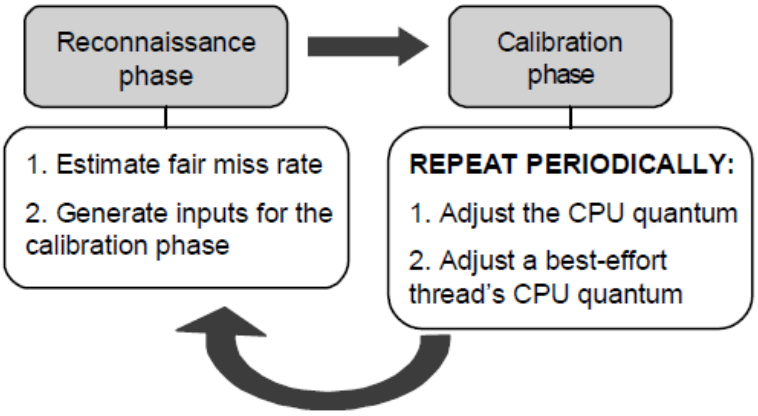


Figure 1.6 : Cache-fair algorithm structure.

One other research about multicore system schedulers is "Parallel Task Scheduling on Multicore Platforms". In this research it is indicated that shared caches cause low performance, this is not convenient for multicore systems. L1 cache misses or pipeline conflicts are not significant compared to L2 cache misses. In this research a new scheduling algorithm is proposed. According to this scheduling algorithm, pfair scheduling algorithm and the global earliest-deadline-first (EDF) algorithm are used with a single run queue. Pfair scheduling algorithm is to divide tasks to subtasks that each subtask has its own window (execution time interval) and complete these subtasks with earliest dead line first method. Furthermore, these subtasks can be released early before its window. A factor that describes distance between two tasks execution quantum is called spread.

In Figure 1.7 a two-processor Pfair schedule of a set of five tasks is shown. Each task has its own weight that determines execution time in its window. In this figure ¼ weighted tasks are assumed to be in the same task group. In inset (a) early releasing is not used, spread is calculated 3 in this phase. In inset (b) all tasks' windows are shifted one quantum back and all tasks early released by one quantum, and then spread is calculated 3 again. In inset (c) selective early release is applied and spread decreased to 2. Last inset shows no early shift applied, however tasks can miss deadlines by one quantum[6].

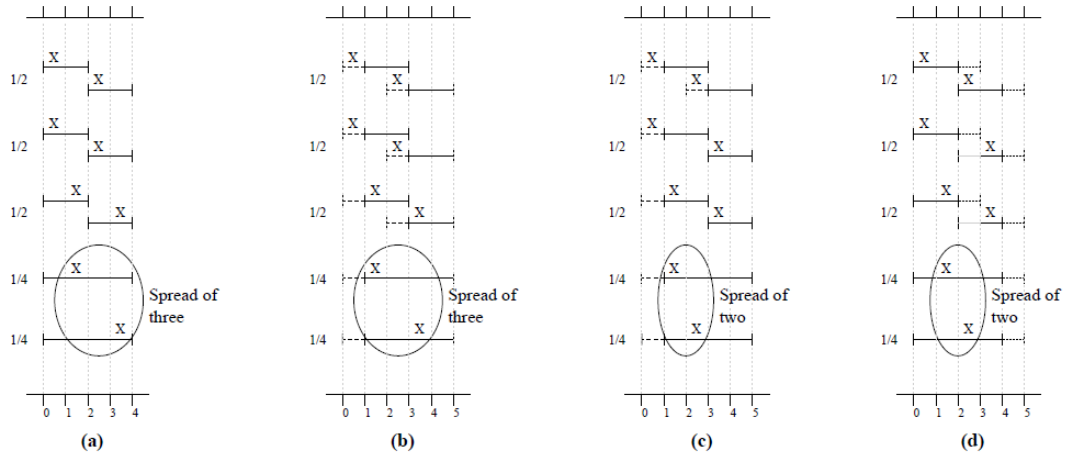


Figure 1.7 : A two-processor Pfair schedule of a set of five tasks.

Operating systems are also released that conventional schedulers are insufficient and unsuitable for multicore systems. Linux is one of them and a popular operating system. Linux kernel 2.6 is adapted to provide better performance on multicore processors. Two policies are used to determine load balancing and scheduler structure. They are power saving policy and peak performance policy. Peak performance policy is added in Linux 2.6.17. Peak performance policy is about equal load balancing. An example of peak performance policy can be seen in Figure 1.8[11].

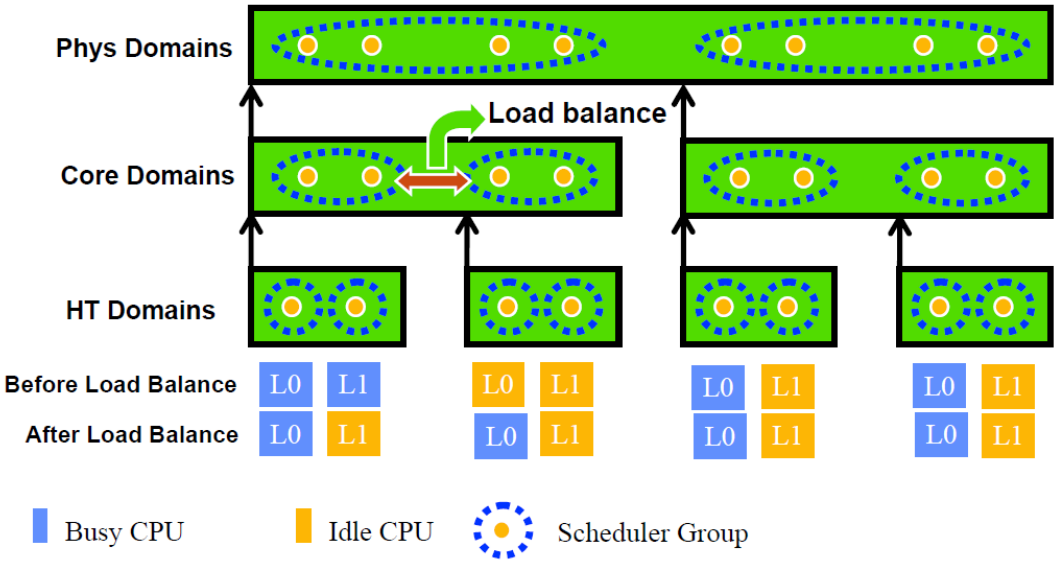


Figure 1.8 : Peak Performance Policy.

Power saving policy is added in Linux 2.6.18-rc1. In this policy physical packages/cpu-cores will be minimized, in this way some of the processors can be inactive, thus provides power saving. An example of power saving policy can be seen in Figure 1.9.

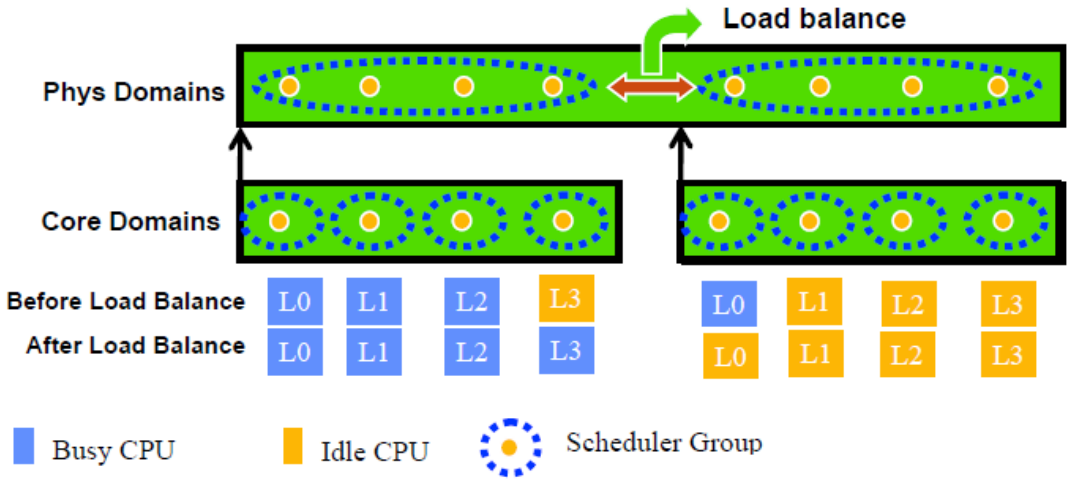


Figure 1.9 : Power Saving Policy.

1.3 Hypothesis

This study provides a different point of view to multithreading parallelization on multicore systems. Distribution of objects to cores of multiprocessors by an algorithm that enforces object to stay in the processor even in the core which it is created on provides better performance and parallelism than random distribution or distribution to the first empty core methods. Furthermore this study reveals that number of attributes and method calls made have an effect on performance and execution time.

2. DEFINITIONS

2.1 Multiprocessor

Multiprocessor is a structure that consists of more than one processors which share all memory and I/O[17]. All processors communicates via memory, in order to provide load balancing, decrease extra work. Since all processors share memory, read/write conflicts occur. Cache coherency unit, that exists in every processor, is used to resolve these conflicts[18]. Multicore architecture can be seen on Figure 2.1.

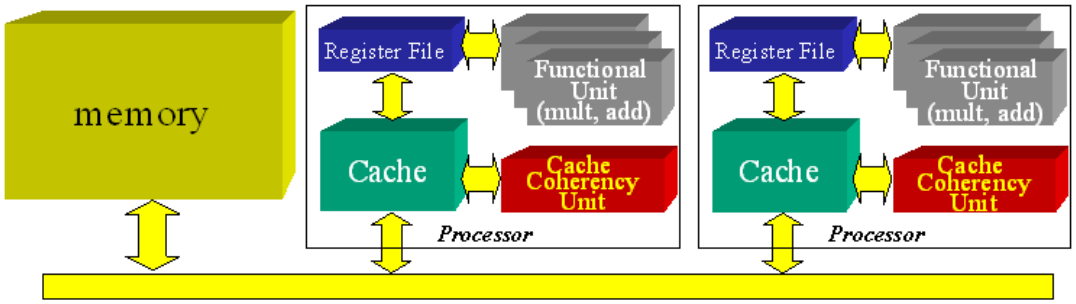


Figure 2.1 : Multiprocessor Architecture

There are several categories of multiprocessing. They are shared nothing MP, shared disks MP, shared memory cluster and shared memory MP. In shared nothing MP processors have their own memory, cache and disk. They don't share anything. Pure cluster is another name for this multiprocessing type. Processors interact with message-passing. In shared disk MP processors have their own memory and cache. They only share disk. Processors interact with message-passing. Processors are loosely coupled. In shared memory cluster processors have their own main memory, cache and disk. However they have a shared memory, and they communicate via this shared memory and all processors are tightly coupled. In shared memory MP processors share disk, main memory and I/O devices. Processors are tightly coupled[19].

2.2 Multicore

Multicore architecture is a new term that a single chip which consists of multiple processor cores. Chip multiprocessor (CMP) is another name for multicore. Multiple threads can run concurrently in multicore processors. Multicore technology is invented because of clock frequency handicap, improving parallelism and multithreading support and heat problems. Lower power consumption and performance boost are another advantages of multicore architectures[7].

Operating systems see every core as a seperate processor. Every task assigned to a core. Every operating system develops new schedulers for adaptation process. Multicore processors requires new technologies to work efficiently[1].

Multicore architecture can be categorized into two types: homogeneous multicore processor and heterogeneous multicore processors. In homogeneous multicore processors all core types are same. In heterogeneous multicore processors different types of cores can be inside the processor. These two processor types can be seen on Figure 2.2[20].

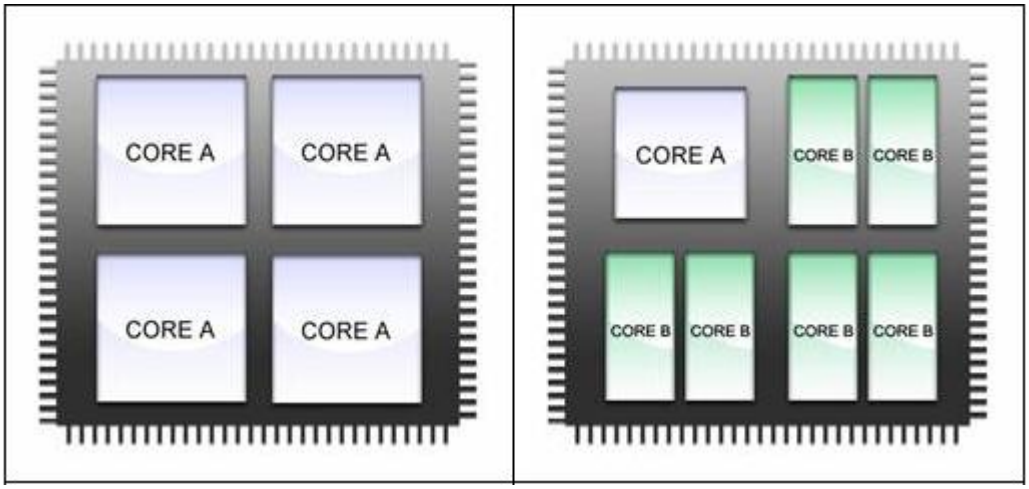


Figure 2.2 : Homogeneous multicore and heterogeneous multicore CPU

Dual core or quad core processors that are used in personal computers are examples of homogeneous multicore processors. IBM Cell is the example of heterogeneous multicore processors.

2.3 Cache

Every central processing unit (CPU) has its own cache. It reduces average time to access memory and increases performance. Cache has an important role on computers. There is a tradeoff between speed increase and cost. Cache is more expensive and faster than memory due to its architecture. That is why cache is very small on processors. If data is accessed frequently, then it is stored in cache. Processors first access their own cache before accessing memory. If data is found then there is no need to access memory. If data is not accessed for a while, then it will be removed from cache and new data will be stored instead of it. There are two terms that are used for cache access. First of them is cache hit that means processor found data which it looked for. Second term is cache miss that means processors didn't found data which it looked for[21]. Cache miss rate is very important for performance analysis and scheduling algorithms.

In multicore processors all cores have its own cache. It is called level 1 cache (L1). Most of multiocore processors also have level two cache (L2) that shared among cores. Level 2 cache is bigger than level one cache. Each core firstly accesses to its own cache (L1 cache), if it misses, then it accesses L2 cache, if it misses again, it accesses to main memory. This is the data access process of cores. L1 and L2 cache can be seen on Figure 2.3[22].

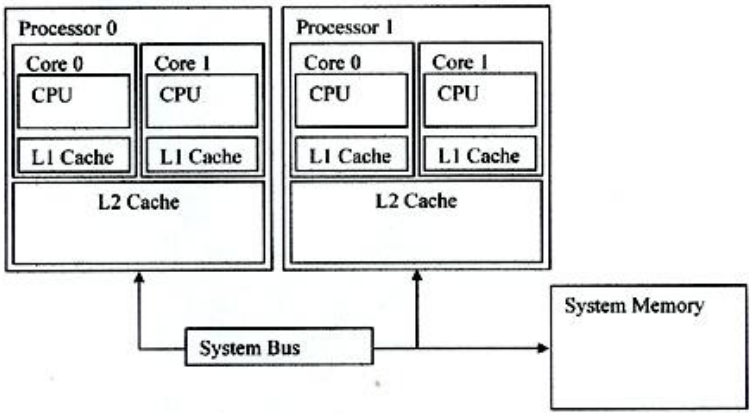


Figure 2.3 : L1 and L2 cache of a dual core processor

Cache coherence problem is very important on L1 cache. This problem is that if a value on one cache of core updated, then its other copies on cores are wrong. There are protocols to solve this problem. One of them is invalidate protocol that if a core writes to a data, then other cores are sent invalid signal of data.

2.4 Scheduler

Scheduling is a task assignment problem. In this problem there are constraints to obey. Resource capacities, deadlines, presedences, priorities are examples of constraints[23].

Operating systems use schedulers to assign CPU time (quanta) to processes. Schedulers determine start time and end time of processes subject to a scheduling algorithm and constraints. There are three types of this schedulers. They are: short-term, long-term and mid-term schedulers[3]. When a new process is created, long-term scheduling is done. Admition decision is given by this scheduler. Mid-term scheduler determines memory management, it decides which process to be in memory. Short-term scheduler decides which process to execute next.

There are two scheduling policies: preemptive or non-preemptive scheduling (also known blocking or nonblocking scheduling). In preemptive scheduling process execution can be cut by an interrupt or system call. Round-Robin (RR) and priority are examples of preemptive scheduling algorithms. In non-preemptive scheduling process is executed until it finishes. first come first served (FCFS) and shortest job first (SJF) are examples of non-preemptive scheduling algorithms[9].

Different scheduling policies and algorithms are needed for multicore systems. Conventional schedulers are insufficient for multicore systems. It is not enough to increase the number of the run queue for adaptation process. Since each core has its own level one cache, coherence problems occur. Second problem is multicore systems can be heterogeneous. In this case it is not possible to threat all cores as they are all same. Load balancing and power saving policies are another factors that affect scheduling algorithm[25].

Linux kernel 2.6 is adapted to provide better performance on multicore processors[24]. Two policies are used to determine load balancing and scheduler structure. They are power saving policy and peak performance policy. Peak performance policy is about equal load balancing. In power saving policy physical packages/cpu-cores will be minimized, in this way some of the processors can be inactive, thus provides power saving[11].

3. INVESTIGATING PERFORMANCE OF PARALLEL OBJECTS ON MULTICORE SYSTEMS

On multicore systems processor based and even core based context switch is expensive. Frequently used objects staying in the same core have performance advantages against switching core that is in the same processor or different processor. Furthermore the number and the size of the attributes and method call counts have an effect on performance and response time. In this section experiments are done to investigate performance of core switch and effect of attributes and method call counts on multicore systems. Test environment and test algorithm will be introduced firstly.

3.1 Test Environment

For this study test environment is a computer with Fedora 10 Linux 64 bit operating system, 4 Intel(R) Xeon(TM) CPU 2.60GHz, 4096KB level 2 cache size processors. Each processor has 12K instruction cache and 16K data cache as level 1 cache. Every processor has two cores, therefore total 8 cores are used for test environment. Machine processor structure can be seen on Figure 3.1. C++ is used as programming language, and g++ is used as compiler. Processors have hyperthreading support, however this technology is not used for experiments.

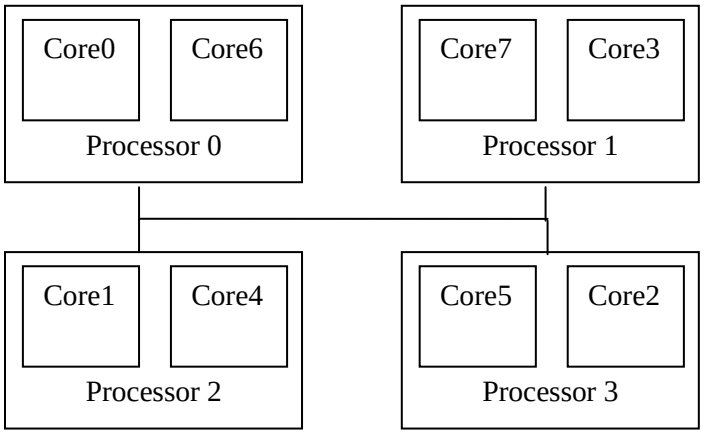


Figure 3.1 : Test computer CPU structure

3.2 Test Cases and Algorithm

In this study a test program is developed for experiments. The point is to create an object on certain core and then call its methods on the same core that object created and another test case is calling its methods on other cores. Flow chart of the test algorithm can be seen on Figure 3.2.

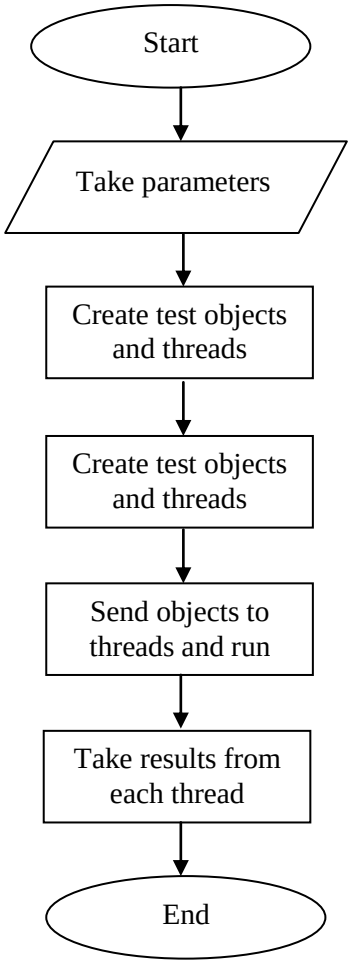


Figure 3.2 : Test program flow chart

In test program there is a test class for measurements. It consists of a double array attribute and two methods. One of them reads array and second one writes on array elements. The size of the array is taken as a start parameter of the program. Two test objects and threads created at first and these objects send to the threads as parameters. Then in each thread method of the objects is called repeatedly. This method reads the array in a loop one time to increase cache usage. After every four method call, one other method is called that makes write operation on array elements. Write operation is done to mess the caches. Otherwise whole array will be in every

caches of processors and even cores. Thus it will be pointless to measure performance and compare test cases. Method call count is taken as a start parameter of the program, too. Before each method call a decision made to find the core which the method call is done on. Core assignment function is called every time before method call, even it stays on same core to provide fairness. Core assignment policy depends on test case. Type of the test case is taken as a start parameter of the program, too. After all method calls finish, time is measured to see the whole execution time.

There are four test cases in these experiments. One of them is same core case. In this case two objects are in the same processor, but different core. All method calls made in the same core, without core and processor switch. However core assign function is called every time before method call to provide fair measurement. Second case is same processor, but different core case. In this case two objects are in the same processor, but different core again. However at every method call objects swap cores. They stay in the same processor, but they swap cores. In third case two objects are in the same processor, but different core again. Then at every method call objects switch processor. All 4 processors and 8 cores are used in this case. At each call another core is used. After each 8 core switch, method call is done on the same core again. In the last case assign core function is not used. It is left to the operating system to determine core assignment. These test cases can be seen on Figure 3.3.

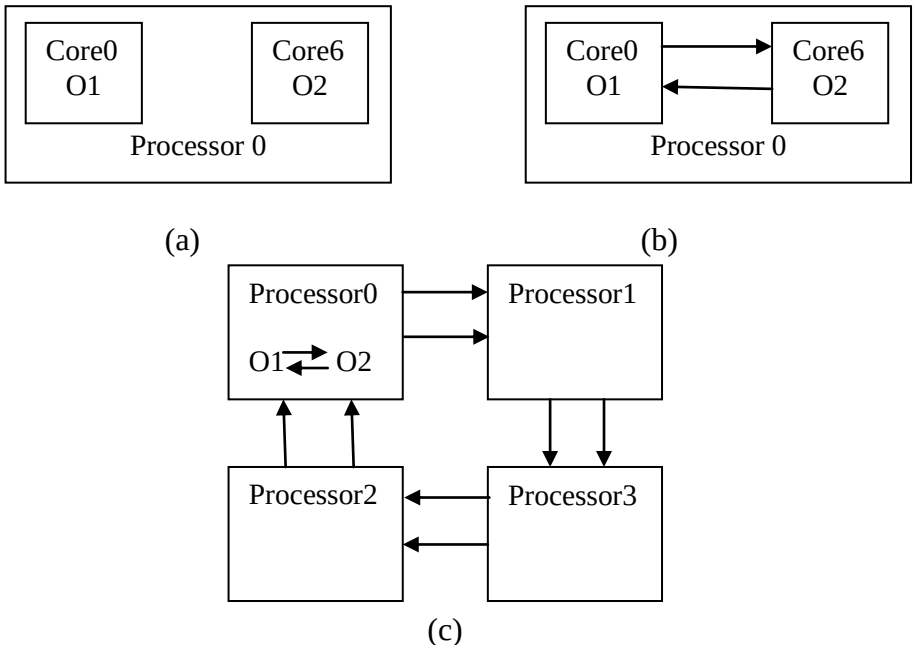


Figure 3.3 : Test Cases a: same core, b: same processor,different core, c: different processor and different core (for all 2 objects)

3.3 Experiments

Experiments to investigate performance on multicore systems are done according to test cases that explained in Section 3.2. Time measured at the end of the last method call, at the end of the method. Measurement unit is the number of clock ticks elapsed since the program start. This is the output of the "clock" function of time library of c. For easy notation and understanding last three zero deleted. For each experiment minimum 10 trials made to take accurate results, and additionally average, maximum and minimum values are recorded. Experiments are done seperately for every method call count and array size, and for each method call count and array size trials are done for four test cases about minimum 10 times each. For example for 500 method call size and 1000 array size 40 trials are done (4 test case * 10 trials each). Experiments are divided into two parts to show the effects of method call count and attribute size.

3.3.1 Investigating method call count effect

In these experiments array size of test object is 1000 double numbers. This value stays constant while these experiments. Method call count effect will be investigated through these experiments for all four test cases. Average values are calculated after minimum 10 trials and extreme results aren't used.

3.3.1.1 Experiment1: 500 method call

Results of this experiment can be seen on Table 3.1.

Table 3.1: Duration of the program with 500 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	10	10	50	0
Max	20	40	60	30
Avg	13	30	54	10

This experiment results on Table 3.1 shows that same core and leaving to operating system gives better results. Operating system has its own scheduler and it doesn't use core swicth so that its result is close to same core. It is difficult to compare with operating system, because experiments and core switch operations made in user level. However same core case gives better results compared to different core and different processor cases. This is because the effect of level one cache. Most used

variables are stored in the caches of the cores and it provides an increase on performance and decrease on execution time. At this point the attributes of the objects are stored to cache of the core and then access to these attributes is easier than accessing from memory. Different core case results are better than different processor case due to the level 2 cache. Level two cache takes place between the cores of the same processor. Thus level two cache has a performance increase among switching processor and staying in the same processor.

3.3.1.2 Experiment2: 1000 method call

Results of this experiment can be seen on Table 3.2.

Table 3.2: Duration of the program with 1000 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	30	60	100	30
Max	40	80	130	40
Avg	34	70	116	31

Inreasing method call count don't make any important changes compared to previous experiment. All values nearly doubled. It is an expected result. Same core case is better than different core and different processor case and again different core case (staying in the same processor, but switching core) is better than different processor case. If it is left to the operating system it gives results close to same core case. (Note that in all cases except operating system case core assign function run, and that makes unfairness. Furthermore operating system arranges threads and tasks better than user level with system calls. That is why it is not realistic to compare operating system cases with other cases. It is shown only to draw a conclusion and give hints.)

3.3.1.3 Experiment3: 2000 method call

Results of this experiment can be seen on Table 3.3.

Table 3.3: Duration of the program with 2000 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	70	130	220	70
Max	80	170	260	70
Avg	73	142	237	70

Inreasing method call count don't make any important changes compared to previous experiment. All values nearly doubled. It is an expected result. Same core case is

better than different core and different processor case and again different core case (staying in the same processor, but switching core) is better than different processor case.Same core case is still nearly same with operating system case. Results increase linearly compared to previous experiments, this is because write operation done once in four method call, and this operation makes caches dirty. Since data changed, it will result in cache miss.

3.3.1.4 Experiment4: 4000 method call

Results of this experiment can be seen on Table 3.4.

Table 3.4: Duration of the program with 4000 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	150	260	470	130
Max	180	310	530	160
Avg	160	284	488	140

Inreasing method call count don't make any important changes compared to previous experiment. All values nearly doubled. Same core case is better than different core and different processor case and again different core case (staying in the same processor, but switching core) is better than different processor case. Same core case is still nearly same with operating system case. It is also important that maximum value of same core case is still better than minimum value of different core case.

3.3.1.5 Experiment5: 8000 method call

Results of this experiment can be seen on Table 3.5.

Table 3.5: Duration of the program with 8000 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	310	550	904	260
Max	330	610	1050	290
Avg	317	575	994	274

According to results on Table 3.5 same core case is better than different core and different processor case and again different core case (staying in the same processor, but switching core) is better than different processor case. It can be seen that processor switch is more expensive than core switch (but in same processor). Level two cache provides this performance boost. Level one cache benefit is between same core and different core case, because if object stays in the same processor and in the

same core, it uses level one cache, and it results cache hit. So that there is no need to look level two cache or main memory. Although write operations and data change, it is still better. Difference between operating system case and same core case is more explicit. Since whole execution time increased, operating system took advantage of it. Operating system intervenes and does its own jobs, schedules in lower level (not in user level). That is why operating system case results are better.

3.3.1.6 Experiment6: 16000 method call

Results of this experiment can be seen on Table 3.6.

Table 3.6: Duration of the program with 16000 method call, 1000 array size

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	620	1100	1940	550
Max	650	1160	2060	580
Avg	632	1128	2018	560

According to results on Table 3.5 same core case is better than different core and different processor case and again different core case (staying in the same processor, but switching core) is better than different processor case. Gap between results of test cases is very long. It is shown that core switch in same processor is worse than staying in the same core. As method call count increases, context change cost is increase, too. In same processor level one cache and level two cache affect performance very much. Operating system difference is more clear when method call count increases and whole execution time increases.

3.3.1.7 Conclusion

All experiment results are shown in Figure 3.4 as a graphic. On x axis method call count is shown and on y axis average time is shown. As a conclusion same core case gives better results compared to different core and different processor cases. Level one cache is the key factor of this result. Most used variables are stored in the caches of the cores and it provides an increase on performance. The attributes of the objects (in these experiments a double array that consist of 1000 double elements) are stored to cache of the core and then access to these attributes is easier than accessing from memory. Level one cache is very small amount, however it is important to store data and access them via cache. Different core case results are better than different processor case due to the level 2 cache. In same processor cores first access to their own level one caches. If there is a cache miss, then core accesses to level two cache

that is shared between cores of the same processor. That is why staying in the same processor is better than processor switch.

Operating system results are shown only to draw a conclusion and give hints. It is unrealistic and not meaningful to compare other cases with operating system case. Operating system intervenes and does its own jobs, schedules in lower level (not in user level). Furthermore assign core method is not called in every method call in operating system case, and it is not fair for measurements.

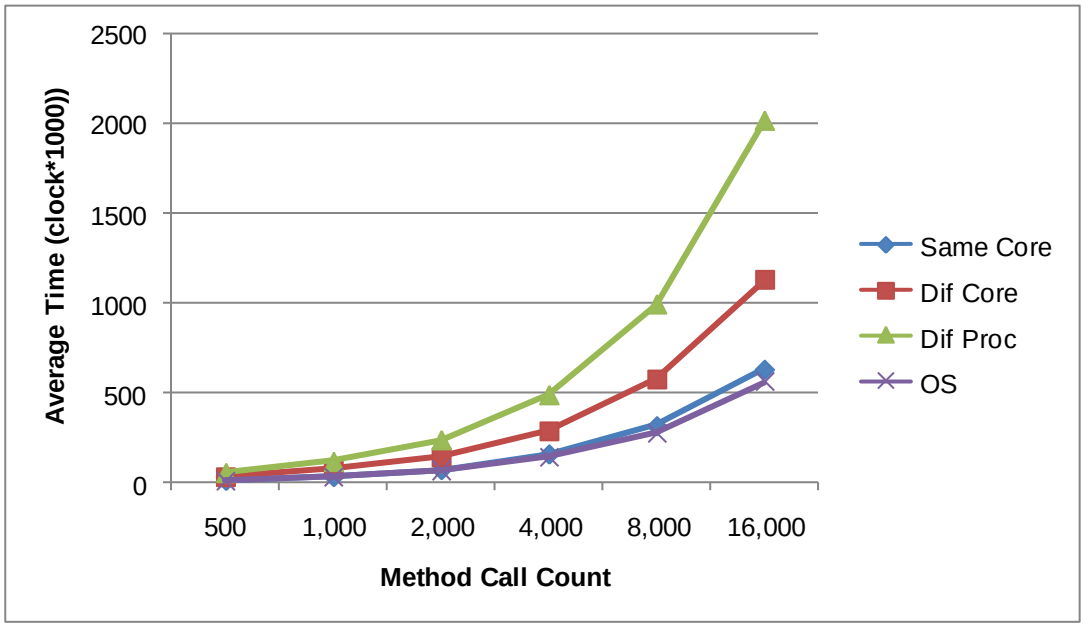


Figure 3.4 : Effect of method call count on the performance

In the graphic on Figure 3.4 0-2000 interval on x axis isn't clear enough to draw a conclusion. Thus this part of the graphic is redrawn in more detail again. This graphic is on Figure 3.5. In this graphic it can be seen that same core case is similar to the operating system case. Same core case is better than different core case and different core case is better than different processor case. These values are expected, because of caches.

As a conclusion object based distribution based on same core principle gives better results than different core and different processor distributions. As method call count increases, distance between same core case, different core case and different processor case increases,too.

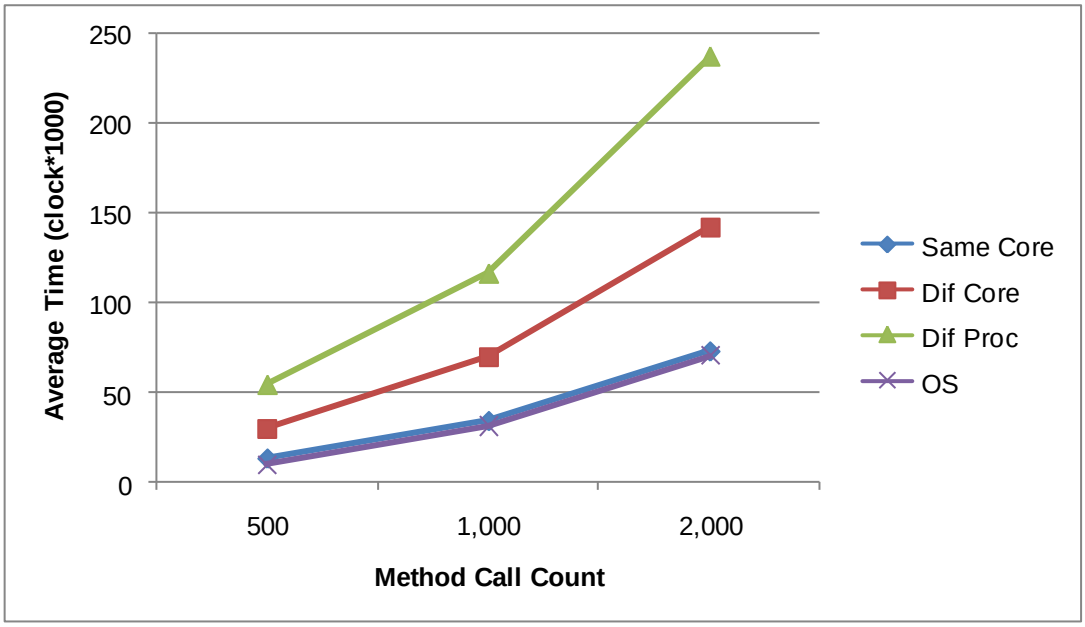


Figure 3.5 : Effect of method call count on the performance first part detailed

3.3.2 Investigating attribute size effect

In these experiments method call count is 1000 double numbers. This value stays constant while these experiments. Object's attribute size effect will be investigated through these experiments for all four test cases. Average values are calculated after minimum 10 trials and extreme results aren't used. Attribute size of object is important because it affects cache usage rate.

3.3.2.1 Experiment1: 500 array size

Results of this experiment can be seen on Table 3.7.

Table 3.7: Duration of the program with 500 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	10	40	50	0
Max	30	50	70	10
Avg	18	49	58	5

On Table 3.7 it is seen that operating system case has the best results. However it isn't fair and meaningful to compare this case with others, because in operating system case core assign function isn't called on every method call. Operating system results are shown to give opinion. Level one cache length is greater than 500 double elements, so it isn't problem to store this array in level one cache of the core.

Although write operations made on array, it is more efficient to use level one cache as staying on same core of the processor. Hence same core case results are better than different core (but same processor) and different processor case results. If an object stays on same core, its attributes are stored in level one cache of the core. That is the reason that same core case is better. Different processor case is worse than different core case as expected due to the level two cache. If object stays on the same processor, its attributes are stored in level two cache of the processor.

3.3.2.2 Experiment2: 1000 array size

Results of this experiment can be seen on Table 3.8.

Table 3.8: Duration of the program with 1000 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	30	60	100	30
Max	40	80	130	40
Avg	34	70	116	31

Inreasing array size don't make any important changes compared to previous experiment. Since double has a size of 8 byte, array size is 1000*8=8000 byte, smaller than 8 KB. This is still under the size of level one cache (Level one cache data part is 16 KB). There is no overflow event. Therefore staying in the same core is again has better results compared to different core and different processor case results. Accessing to level one cache is quicker than level two cache and in the same way accessing level two cache is quicker than shared memory. This experiment is also confirms and verifies this statement. Besides operating system case results are very close to same core case results despite some disadvantages like calling assign core function on every method call and its own scheduler.

3.3.2.3 Experiment3: 2000 array size

Results of this experiment can be seen on Table 3.9.

Table 3.9: Duration of the program with 2000 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	70	100	200	70
Max	70	120	230	70
Avg	70	109	210	70

In this experiment due to the results on Table 3.9, there isn't any unexpected value. Since double has a size of 8 byte, array size is 2000*8=16000 byte, smaller than 16 KB. This value is close to the size of level one cache (Level one cache data part is 16 KB). Therefore it is an expected result that same core case results are better than different core and different processor cases. Same core case results and operating system case results are same. This is an interesting result. Minimum value of the different core case result is still greater than maximum value of the same core case results. Similarly minimum value of the different processor case result is still greater than maximum value of the different core case results.

3.3.2.4 Experiment4: 4000 array size

Results of this experiment can be seen on Table 3.10.

Table 3.10: Duration of the program with 4000 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	140	190	390	130
Max	150	210	430	160
Avg	148	195	400	140

In this experiment the level one data cache exceeded. Since double has a size of 8 byte, array size is 4000*8=32000 byte, close to 32 KB. Level one cache data part is 16 KB. Therefore data size is greater than data part of the level one cache and all arrray can't be stored in the level one cache. However same core case results are better than different core and different processor case results again. The reason of that is even all array can't be stored, some parts of it is stored in the level one cache. Moreover address of the first element is stored in the level one cache and access to the array is quicker anyway. Otherwise level one cache would be useless. It can also seen that same core case results are close to different core case results. This is because in same core case if data isn't found in level one cache, then data is found in level two cache most likely. However the usage of the level one cache provide a benefit. According to the results on Table 3.10 different core case results are better than different processor case results. It is an expected result, because level two cache is big enough to hold the whole array (4 MB). That is why the the ratio between different core case and different processor case isn't changed compared to previous experiments. Operating system case results are close to, but better than same core case results, and this is also expected, because the reasons explained before.

3.3.2.5 Experiment5: 8000 array size

Results of this experiment can be seen on Table 3.11.

Table 3.11: Duration of the program with 8000 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	280	350	760	270
Max	340	370	840	330
Avg	290	360	790	280

In this experiment the level one data cache exceeded, too. Since double has a size of 8 byte, array size is 8000*8=64000 byte, approximately 64 KB. Level one cache data part is 16 KB. Therefore data size is greater than data part of the level one cache and all arrray can't be stored in the level one cache again. Same core case results are again better than different core case results, however the difference between these results are close. In same core case mostly level one cache is used, but if data can't be found in level one cache, then level two cache is used to find data. In different core case mostly level two cache is used, because method calls made on different cores at each time and write operation on array is made. That is why level one cache of cores aren't sufficient to access data. In different processor case each method call made on different processor each time and write operation on array is made once in every 4 method calls. Therefore level two caches of processors are useless. Thus different processor case results are worse than different core case results. Operating system case results are similar to same core case results, however it isn't meaningful to compare them. This values are just shown to give opinion.

3.3.2.6 Experiment6: 16000 array size

Results of this experiment can be seen on Table 3.12.

Table 3.12: Duration of the program with 16000 array size, 1000 method call

Results (clock ticks x 1000)	Same Core	Different Core	Different Processor	Operating System
Min	550	690	1520	550
Max	570	720	1570	570
Avg	560	705	1547	560

Array size of 16000 is big enough to exceed level one cache. Since double has a size of 8 byte, array size is 16000*8=128000 byte, approximately 128 KB. Level one

cache data part is 16 KB. Therefore data size is greater than data part of the level one cache and all arrray can't be stored in the level one cache again. Same core case results are similar to operating system case results. This is very interesting, because it is expected that operating system has advantages in comparison with same core. Besides same core case results are better than different core and different processor case results again. Minimum value of the different core case result is still greater than maximum value of the same core case results. Similarly minimum value of the different processor case result is still greater than maximum value of the different core case results.

3.3.2.7 Conclusion

All experiment results are shown in Figure 3.6 as a graphic. On x axis array length is shown and on y axis average time is shown. As a conclusion same core case gives better results compared to different core and different processor cases. In the graphic on Figure 3.6 the difference between operating system case and same core case can't be seen clearly. The results are very close. The different core case is very separate. It means that processor switch is very costly and expensive. It slows down the performance. Since the data is already changed with writing operations each time, it is not possible to use caches for accessing data. In this case data is accessed from shared memory. This event increases whole exection time. That is why different processor case is worst case.

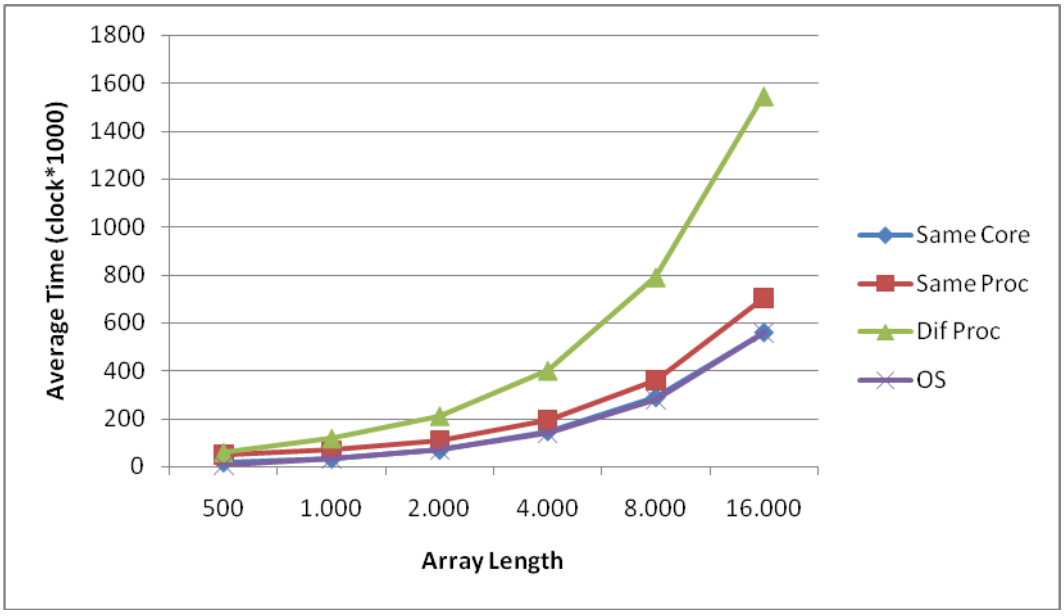


Figure 3.6 : Effect of size of the object on the performance

In same core case data is accessed from level one cache as much as possible. There is a limit because level one cache is small compared to level two cache and shared memory. Under the limit that data isn't exceed level one cache, same core is much better than same processor case, however above the limit that data exceeds level one cache, then gap between same core and same processor (different core) case is close. However same core case is better because level one cache is used mostly, although all data can't be found. When level one cache miss happens then level two cache is accessed to find data.

Operating system case results are very close to the results of same core case results. Operating system results are shown only to draw a conclusion and give opinion. It is unrealistic and not meaningful to compare other cases with operating system case. Operating system intervenes and does its own jobs, schedules in lower level (not in user level). Furthermore assign core method is not called in every method call in operating system case, and it is not fair for measurements. However at some points same core case results are same with the operating system case results.

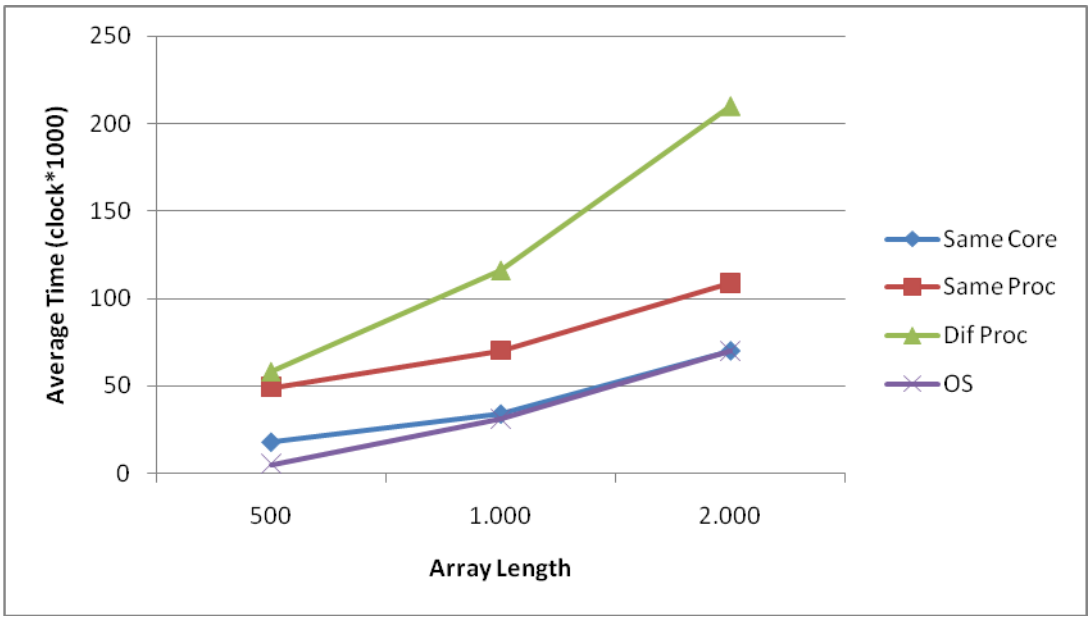


Figure 3.7 : Effect of size of the object on the performance first part detailed

First three points of the graphic is not clear enough to draw a conclusion. Therefore this part of the graphic is redrawn again. It can be seen on Figure 3.7. This graphic is significant, because it is shown that same processor (different core) case and same core case aren't same. Data size is under the level one cache in these array length values. Hence ratio between same core case and same processor results is nearly two.

4. DISTRIBUTION OF OBJECTS ON MULTICORE SYSTEMS

In section 3 performance of parallel objects is investigated on multicore systems and problems introduced related with cache and shared memory access. It is a scheduling problem. It is shown that for an object staying in the same core is better than switching core or processor and staying in the same processor has better performance compared to switching processor. In this chapter new object distribuion algorithm will be introduced. This proposed algorithm the main purpose is to keep object in the same core. If this core isn't empty and the other core of the same processor is empty, then object is assigned to this core. If this other core isn't empty,too, then object is assigned to the queue of the previous core that object is on. According to the algorithm main purpose is to let object to stay in the processor and not allow to leave it.

4.1 Proposed Algorithm

The fundamental of the algorithm is based on the results of the experiments in third section. These experiments show that object should stay in the same core for every method call to increase performance at access times. The basis of the proposed algorithm is to keep object in the same core and in the same processor.

In this algorithm there are two map data structures, and a queue data structure. First data structure is object map. This data structure consists of objects that are in the system. Additionally in this data structure object's last assigned core number and last run time are hold. This data structure is used when new method call arrives and during core assignment alogorithm to determine core to be assigned by data hold in this data structure. Second data structure is used to keep load of the cores. Queue data structure is used to keep objects that are assigned to run on a specified core. Each core has its own queue. These queues are hold in a map data structure that are accessed by core numbers (core number is key of the map data structure). After a method execution, object removed from queue. In this algorithm one object can be in one queue at the same time.

First of all when the program starts 8 threads (because in the test server there are 8 cores) are created and each thread is assigned to execute on a unique core. Each thread runs in a loop and waits until its own queue has a new object to run. When an object is found in queue, it is taken from queue and its method executed and then it is removed from queue. Then thread starts to wait new object via its queue. Flowchart of the algorithm can be seen on Figure 4.1.

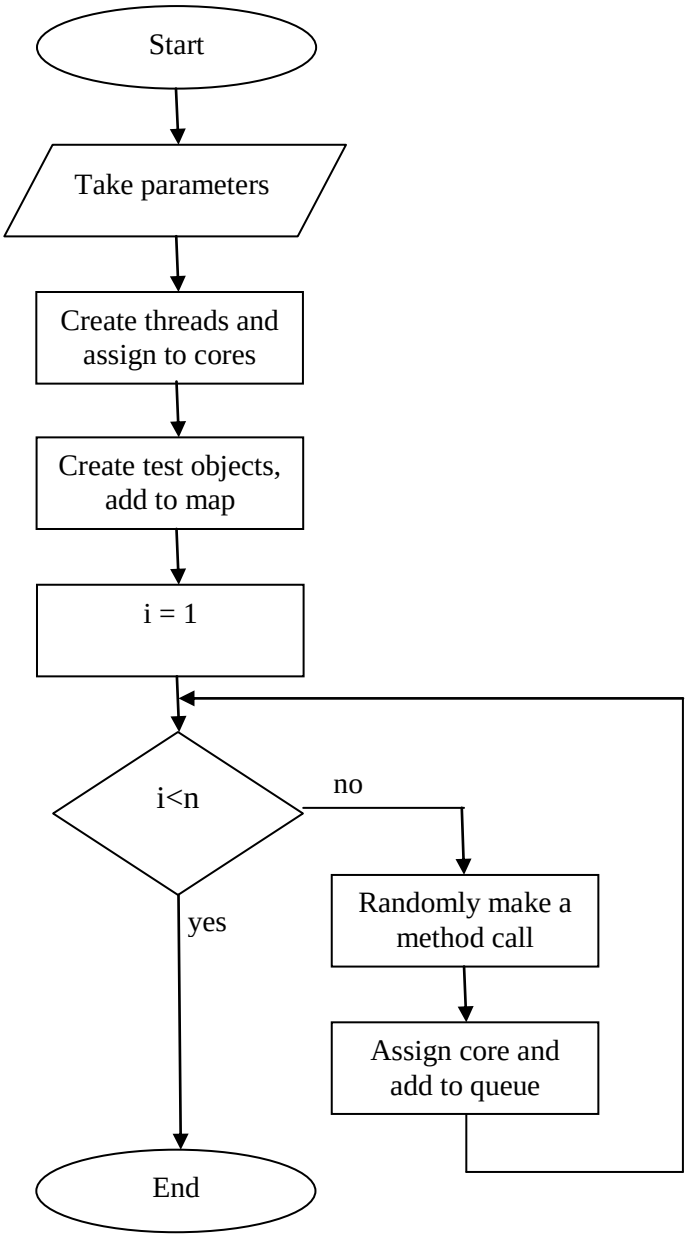


Figure 4.1. Main fuction flowchart

When an object's method is called, a function is called to determine the core number to assign the object. After the core is determined, object is added to the core's queue. The main part of the algorithm is this assign decision function. In this function object

is the parameter. At first it is checked that if object is assigned to a core before or not. In this stage object map data structure is used. If object is not assigned to a core before, then core queues are scanned to find an empty core or to find least filled queue. In this stage core map data structure and queues are used. If object is assigned to a core before, then our new approach is utilized. According to this approach firstly queue of the previous assigned core is checked, if it is empty then object will be added to the queue of this core. If previous assigned core isn't empty then the other core in the same processor is checked, if it is empty then object will be added to the queue of this core. If it isn't empty, too, then object will be added to the queue of the previously assigned core, despite it is not empty. Assign core part can be seen on Figure 4.2 as a flowchart.

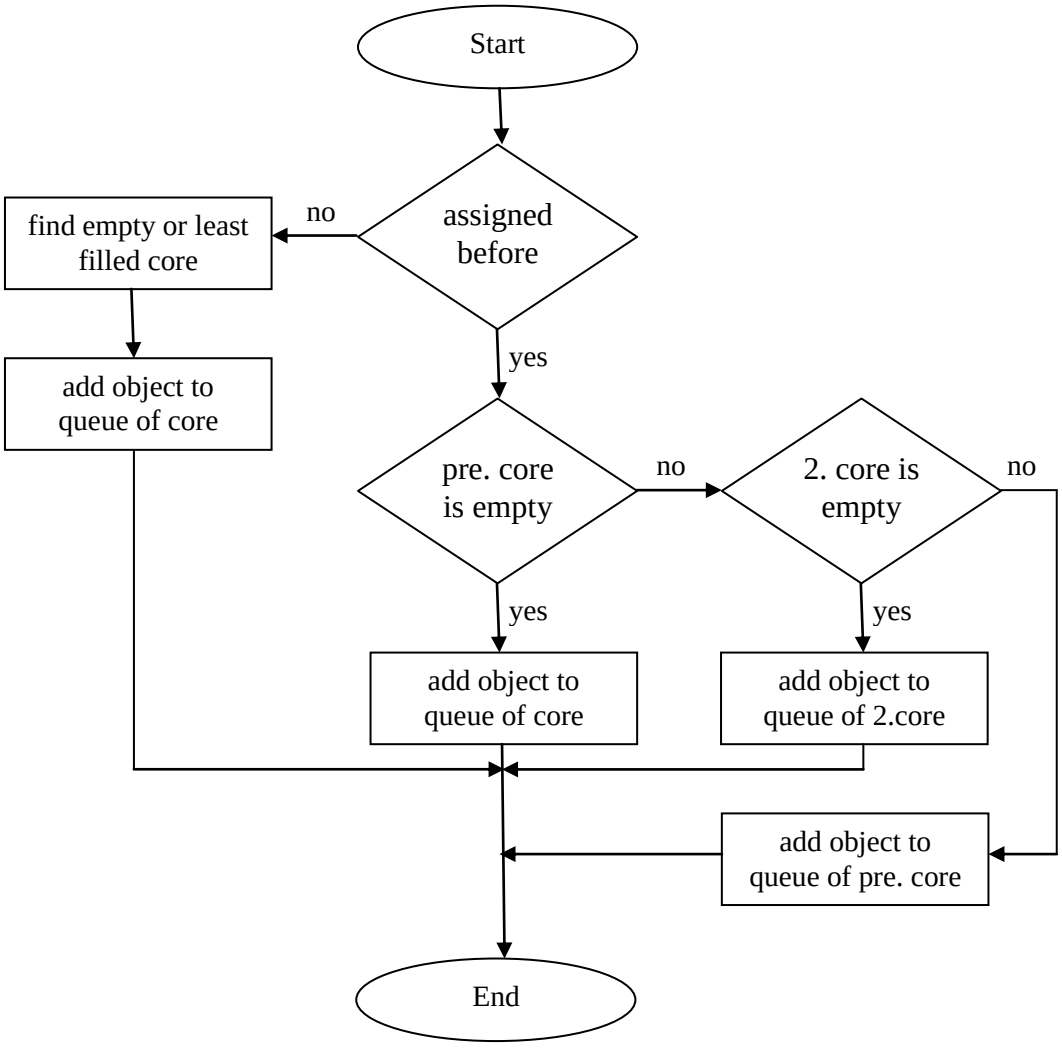


Figure 4.2. Assign core algorithm flowchart

The main purpose of this algorithm is to use benefits of level one and level two cache to increase performance. The object is kept in the same core to utilize level one cache, and if it isn't empty and the other core of the same processor is empty, then object is kept in same processor, but different core. In this case since object is in the same processor, level two cache is suitable to use. Access time is important and this time is shortened by usage of level two cache and especially level one cache. Level one cache access is quicker than level two cache access and level two cache access is quicker than shared memory access. Therefore increase in cache usage and cache hits affect performance directly.

4.2 Experiments

Experiments are done to investigate performance of proposed algorithm on multicore systems. Three algorithms are compared as test cases. All algorithms use data structures that are developed. They are object map, core map and queues data structures. First algorithm is proposed distribution algorithm. Second algorithm is least filled core algorithm. According to this algorithm core assignment is done like that: searching all queues of cores that are stored in the map data structure developed, finding empty or least filled core queue and assigning object to this core. The last algorithm is an algorithm that is based on assigning objects to cores randomly.

A test program is developed for experiments. In this program object count, total method call count and the array size are parameters. At the beginnig objects are created and added to the object map. Then method calls are made randomly to this objects. Tests are consists of different object counts and different method call counts. The important point is that object count is bigger than core count. Performance of the algorithms will be investigated through these experiments for multicore systems. Measurement unit is the number of clock ticks elapsed since the program start. This is the output of the "clock" function of time library of c. For easy notation and understanding last three zero deleted. For each experiment minimum 10 trials made to take accurate results, and additionally average, maximum and minimum values are recorded to draw a conclusion.

4.2.1 Investigating method call count effect on performance for 15 objects

In these experiments 15 test objects are used and array size of each test object is 1000 double numbers. This value stays constant while these experiments. Method call count effect will be investigated through these experiments for all three algorithms. Average values are calculated after minimum 10 trials.

4.2.1.1 Experiment 1: 3000 method call

Results of this experiment can be seen on Table 4.1.

Table 4.1.: Duration of the program with 15 objects and 3000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	70	170	150
Max	110	220	160
Avg	90	180	158

According to the results on Table 4.1 new proposed distribution algorithm gives best results, because this algorithm uses level one and level two cache more than others. If object stays on same core then level one cache hit rate increases, cache miss rate decreases. If object stays on same processor then level two cache hit rate increases, cache miss rate decreases. In proposed object distribution algorithm object stays in the same processor in the worst scenario. In the second algorithm the important point is to execute method in the quickest way by finding empty core or least filled core. In this case object doesn't stay in the same core and its method is executed in the core that isn't connected with its previous core. They are the reasons that this algorithm can't use caches efficiently and worse than proposed distribution algorithm. In the last case random assignment algorithm is tested.It doesn't calculate anything, and quickly makes its decision randomly. In this experiment random assignment is better than least filled algorithm. In this case core and processor switch rate increases and that decreases performance and whole execution time.

4.2.1.2 Experiment 2: 5000 method call

Table 4.2.: Duration of the program with 15 objects and 5000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	150	290	230
Max	180	340	250
Avg	160	320	240

Results of this experiment can be seen on Table 4.2. According to the results on Table 4.1 new proposed distribution algorithm gives best results, too. Increasing method call count doesn't make any change in result orders. Random assignment algorithm results are worse than proposed algorithm results as expected, because its assignment isn't based on core queues or cache usage. Increasing cache hit rate affects performance directly. At this stage least filled algorithm that isn't based on cache usage, is based on least filled or empty core to execute method quickly. It is a kind of load balancing algorithm, however it doesn't take into account data of objects. Therefore load balancing isn't enough to determine core assignment. That is why least filled algorithm is worse than new proposed distribution algorithm.

4.2.1.3 Experiment 3: 8000 method call

Results of this experiment can be seen on Table 4.3.

Table 4.3.: Duration of the program with 15 objects and 8000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	260	520	390
Max	290	600	410
Avg	270	550	405

Results on Table 4.3 shows that proposed algorithm gives better results compared to least filled algorithm and random assignment algorithm. Since total core count is 8, and total object count is 15, almost every core has two objects. This count increases the level one cache miss rate. Thus the main benefit is gained by level two cache for proposed object distribution algorithm. This advantage is the reason that proposed algorithm is better than least filled algorithm and random assignment algorithm. Least filled algorithm is worse than proposed algorithm, despite the scheduling according to the load balance of the core queues. It is also important that minimum value of least filled algorithm results and random assignment algorithm is worse than maximum value of proposed object distribution algorithm results.

4.2.2 Investigating method call count effect on performance for 30 objects

In these experiments 30 test objects are used and array size of each test object is 1000 double numbers. This value stays constant while these experiments. Method call count effect will be investigated through these experiments for all three algorithms. Average values are calculated after minimum 10 trials.

4.2.2.1 Experiment 1: 3000 method call

Results of this experiment can be seen on Table 4.4.

Table 4.4.: Duration of the program with 30 objects and 3000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	70	190	150
Max	110	220	170
Avg	90	210	160

Increasing total object count is important to investigate performance of the algorithms much better, because distribution of objects has an effect on multicore systems. According to the results on Table 4.4 new proposed object distribution algorithm results are better than least filled algorithm and random assignment algorithm. Furthermore maximum value of proposed object distribution algorithm results are better than minimum value of least filled algorithm results and random assignment algorithm results. Cache hits on proposed object distribution algorithm creates the difference between other algorithms. Least filled algorithm results are worse than random assignment results again. That is why in random assignment neither queues of cores nor data of object aren't taken into account on assignment algorithm, however makes quick decision.

4.2.2.2 Experiment 2: 5000 method call

Results of this experiment can be seen on Table 4.5.

Table 4.5.: Duration of the program with 30 objects and 5000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	160	240	240
Max	200	360	250
Avg	185	320	248

According to the results on Table 4.5 new proposed object distribution algorithm results are again better than least filled algorithm and random assignment algorithm. The result of proposed algorithm nearly doubled compared to previous experiment that is done by 30 objects and 3000 method calls, and close but bigger than an experiment that is done by 15 objects and 5000 method calls. Increasing object count increases all results of algorithms compared among same method call counts. That is an expected result, because cache miss rate increases. Level one cache miss rate increases with the number of the objects. Level two cache amount is larger than level

one cache, so size of 30 objects doesn't exceed level two cache. Therefore proposed object distribution algorithm gives better results compared to other algorithms. Least filled algorithm results are again worse than random assignment algorithm, however it isn't meaningful to compare random assignment algorithm with least filled algorithm, because core queues aren't in the operating system side. Therefore it results in time loss.

4.2.2.3 Experiment 3: 8000 method call

Results of this experiment can be seen on Table 4.6.

Table 4.6.: Duration of the program with 30 objects and 8000 method calls

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	210	450	380
Max	300	550	440
Avg	270	500	410

According to the results on Table 4.6 new proposed object distribution algorithm results are again better than least filled algorithm and random assignment algorithm. As method call count increases, size of queues increases, too. Furthermore it is more possible that more than one objects are assigned to each core. As a result level one cache miss rate will increase, because level one cache is small and it isn't possible store so many objects. Object count that is in the same core and the size of the array affect level one cache miss rate. Level two cache is still able to store objects. Proposed object distribution algorithm takes advantage of level two caches mostly. In least filled algorithm and random assignment algorithm object size and data isn't important at the assignment decision stage. In least filled algorithm queues of cores are checked and least filled or an empty core is found and object is added to queue of this core. That is why proposed algorithm has the best results.

4.2.3 Investigating effect of size of the object on performance for 15 objects

In these experiments 15 test objects are used and total method call count is 5000. This value stays constant while these experiments. Effect of size of the object will be investigated on performance through these experiments for all three algorithms. Average values are calculated after minimum 10 trials.

4.2.3.1 Experiment 1: 500 array size

Results of this experiment can be seen on Table 4.7.

Table 4.7.: Duration of the program with 15 objects and array size 500

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	130	200	160
Max	160	230	180
Avg	140	220	163

According to the results on Table 4.7 new proposed object distribution algorithm results are better than least filled algorithm and random assignment algorithm. Since double has a size of 8 byte, array size is 500*8=4000 byte, close to 4 KB. This is still under the size of level one cache (Level one cache data part is 16 KB). There is no overflow event. Therefore staying in the same core as in proposed object distribution algorithm is better than switching core or processor. Least filled algorithm and random assignment algorithm don't consider previous core that object is on, and that is why they aren't catch the results of proposed algorithm.

4.2.3.2 Experiment 2: 1000 array size

Results of this experiment can be seen on Table 4.8.

Table 4.8.: Duration of the program with 15 objects and array size 1000

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	150	290	230
Max	180	340	250
Avg	160	320	240

According to the results on Table 4.8 new proposed distribution algorithm gives best results, too. Since double has a size of 8 byte, array size is 1000*8=8000 byte, close to 8 KB. This is still under the size of level one cache (Level one cache data part is 16 KB). Random assignment algorithm results are worse than proposed algorithm results as expected, because its assignment isn't based on core queues or cache usage. Increasing cache hit rate affects performance directly. At this stage least filled algorithm that isn't based on cache usage, is based on least filled or empty core to execute method quickly. It is a kind of load balancing algorithm, however it doesn't take into account data of objects. Therefore load balancing isn't enough to determine core assignment. That is why least filled algorithm is worse than new proposed distribution algorithm.

4.2.3.3 Experiment 3: 2000 array size

Results of this experiment can be seen on Table 4.9.

Table 4.9.: Duration of the program with 15 objects and array size 2000

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	270	320	470
Max	300	420	490
Avg	273	400	480

According to the results on Table 4.9 new proposed object distribution algorithm results are better than least filled algorithm and random assignment algorithm. Since double has a size of 8 byte, array size is 2000*8=16000 byte, close to 16 KB. Since level one cache data part is 16 KB, level one cache miss rate increases. In this case level two cache is important. Object size is still smaller than level two cache size. In proposed object distribution algorithm object stays in same core or other core of the same processor. Thus cache hit rate increases, because object attributes are stored in caches. One other important point is that in this experiment least filled algorithm results are better than random assignment algorithm results. Increase in the method execution time due to the object size can be the reason that least filled algorithm is better than random assignment algorithm. In small object size and small method call count least filled algorithm seems worse than random assignment algorithm, because core queues are in user level, in the scheduling algorithms that are used for this purpose use operating system values to make a decission.

4.2.4 Investigating effect of size of the object on performance for 30 objects

In these experiments 30 test objects are used and total method call count is 5000. This value stays constant while these experiments. Effect of size of the object will be investigated on performance through these experiments for all three algorithms. Average values are calculated after minimum 10 trials.

4.2.4.1 Experiment 1: 500 array size

Results of this experiment can be seen on Table 4.10.

Table 4.10.: Duration of the program with 30 objects and array size 500

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	130	210	160
Max	160	230	170
Avg	140	220	165

According to the results on Table 4.10 proposed object distribution algorithm results are better than least filled algorithm and random assignment algorithm. Since double has a size of 8 byte, array size is 500*8=4000 byte, close to 4 KB. This is still under the size of level one cache (Level one cache data part is 16 KB). However increasing the object count increases the possibility of the object count of each core. Therefore level one cache miss rate increases, because attributes of each object in same core are stored in level one and level two caches. Despite level one cache is insufficient for large object size and lots of objects, level two cache has the capacity to store much more data compared two level one cache. Random assignment algorithm results are better than least filled algorithm results, because in this scheduling algorithm core queues aren't used in operating system level, and it is time consuming work to search a map with queues and then make a decision compared to random assignment. Core cache isn't considered in both algorithms, so this gain is ignored.

4.2.4.2 Experiment 2: 1000 array size

Results of this experiment can be seen on Table 4.11.

Table 4.11.: Duration of the program with 30 objects and array size 1000

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	160	240	240
Max	200	360	250
Avg	185	320	248

According to the results on Table 4.11 proposed object distribution algorithm results are again better than least filled algorithm and random assignment algorithm results. Since double has a size of 8 byte, array size is 1000*8=8000 byte, close to 8 KB. This is still under the size of level one cache (Level one cache data part is 16 KB). Level one cache miss rate increases with the number of the objects. Level two cache amount is larger than level one cache, so size of 30 objects doesn't exceed level two cache. Therefore proposed object distribution algorithm gives better results

compared to other algorithms. Least filled algorithm results are again worse than random assignment algorithm, however it isn't meaningful to compare random assignment algorithm with least filled algorithm, because core queues aren't in the operating system side that makes a time loss. It is also important that maximum value of proposed algorithm results is better than minimum value of other algorithm results.

4.2.4.3 Experiment 2: 2000 array size

Results of this experiment can be seen on Table 4.12.

Table 4.12.: Duration of the program with 30 objects and array size 2000

Results (clock ticks x 1000)	Proposed Alg.	Least Filled Alg.	Random Assign
Min	260	330	470
Max	300	420	490
Avg	285	405	480

According to the results on Table 4.12 proposed object distribution algorithm results are better than least filled algorithm and random assignment algorithm. Since double has a size of 8 byte, array size is 2000*8=16000 byte, close to 16 KB. Since level one cache data part is 16 KB, level one cache miss rate increases. In this case level two cache is important. In proposed object distribution algorithm object stays in same core or other core of the same processor. Thus cache hit rate increases, because object attributes are stored in caches. One other important point is that in this experiment least filled algorithm results are better than random assignment algorithm results. Increase in the method execution time due to the object size can be the reason that least filled algorithm is better than random assignment algorithm. In small object size and small method call count least filled algorithm seems worse than random assignment algorithm, because core queues are in user level, in the scheduling algorithms that are used for this purpose use operating system values to make a decission.

4.3 Conclusion

Investigation effect of method call count on performance with 15 objects and array size 1000 double numbers experiment results are shown in Figure 4.3 as a graphic. On x axis method call count is shown and on y axis average time is shown as clock

ticks. According to the results on Figure 4.3 proposed object distribution algorithm is better than least filled algorithm and random assignment algorithm results. Gap between least filled algorithm results and random assignment algorithm results increases as method call count increases. Random assignment isn't suitable to compare with other algorithms, because it doesn't contain any methodologies and estimations. Randomly a core is selected and object is added to its queue. According to the results on Figure 4.3 proposed algorithm results and least filled algorithm results don't intersect at any point. Similarly least filled algorithm results don't intersect with the random assignment algorithm results at any point. Another important point is that core queues are generated in user level for test, if operating system data can be used for this purpose, it is expected least filled algorithm to give much better results. This results a time loss and that is why it isn't meaningful to compare random assignment algorithm with other algorithms in this case.

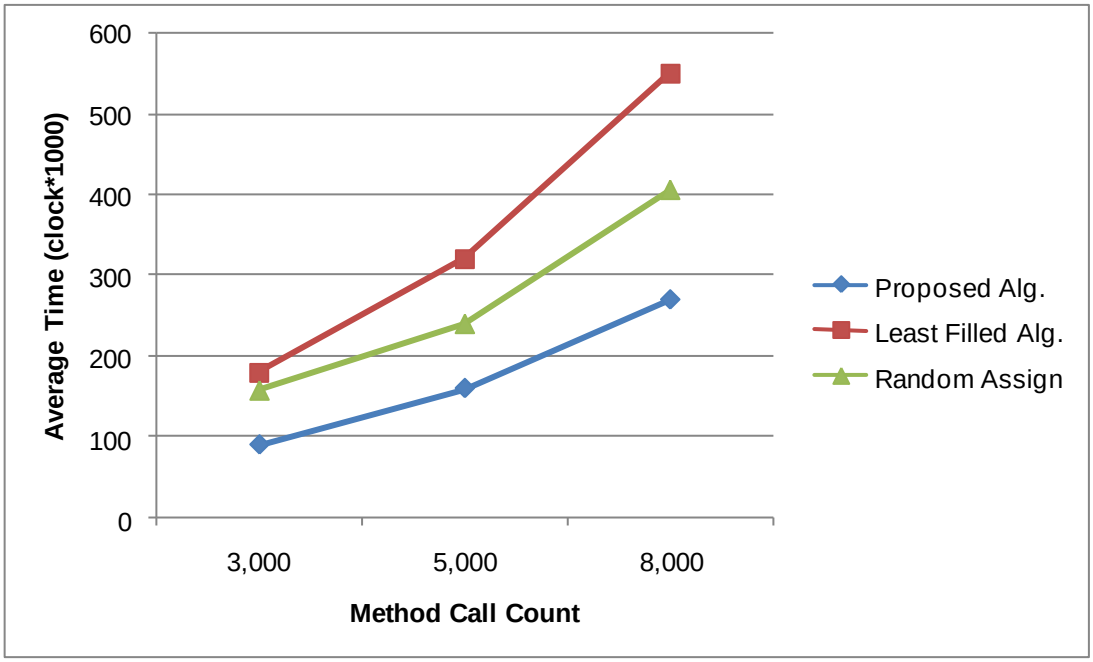


Figure 4.3. Effect of method call count on the performance for 15 objects

In proposed object distribution algorithm it is important to keep object in the same core and same processor. In this case if object stays in the same core, level one cache of core affects performance as high cache hit rate. If object changes its core, however stays in the same processor, then instead of level one cache, level two cache is used mostly to access data. Cache hit rate increases in this way. In other algorithms this point isn't taken into account. Increasing the cache hit rate results in good

performance as shown in Figure 4.3. Gap between proposed object distribution algorithm results and other algorithm results increases as method call count increases. This is an expected result, because the core and processor switch counts (context change) increase and this decreases performance, increases cost.

Least filled algorithm is based on load balancing and scheduling. Whenever an object arrives to the system, it searches for an empty or least filled core queue to assign object to core. Queue size is the important part of decision. The most important distinction between proposed distribution algorithm and least filled algorithm is that in least filled algorithm data isn't considered, only queue sizes are considered to make an assignment decision. However cache hit and miss rates aren't considered as proposed distribution algorithm. In proposed distribution algorithm instead of queue size of core, object's previous assigned core and processor is important during assignment process. That is why this algorithm gives better results.

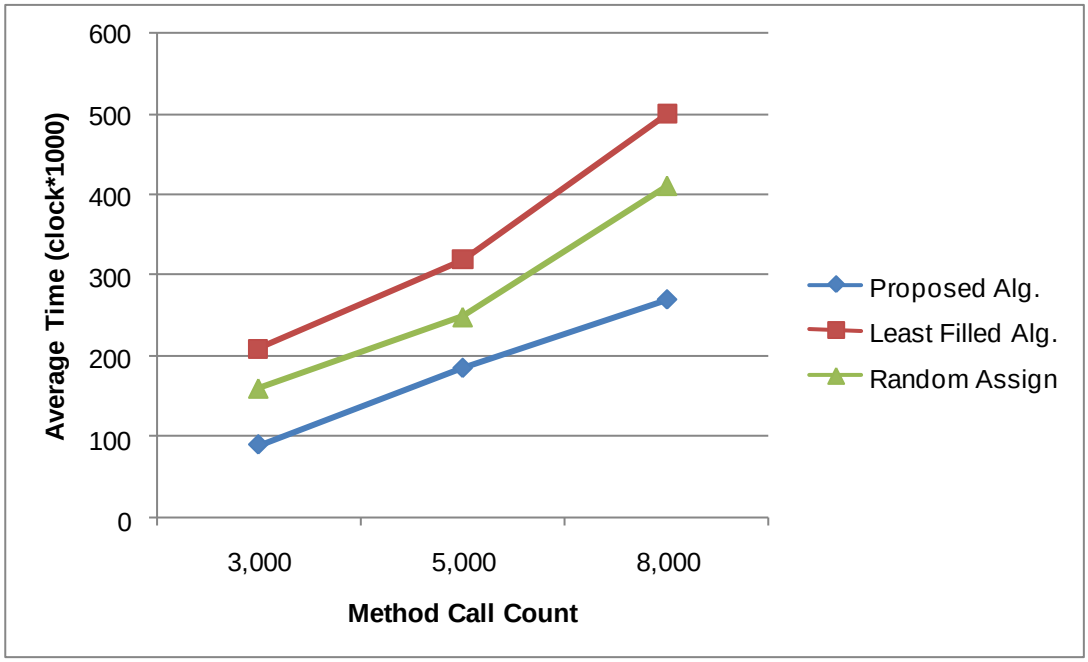


Figure 4.4. Effect of method call count on the performance for 30 objects

Investigation effect of method call count on performance with 30 objects and array size 1000 double numbers experiment results are shown in Figure 4.4 as a graphic. On x axis method call count is shown and on y axis average time is shown as clock ticks. Average values of results are taken as average time. According to the results on Figure 4.3 proposed object distribution algorithm is better than least filled algorithm

and random assignment algorithm results. Least filled algorithm and random assignment algorithm results aren't linear. These results show that performance of these algorithms is bad, if the method call count increases. However, new proposed object distribution algorithm results show nearly a linear move, that is important to be scalable for large amount of objects and many method calls.

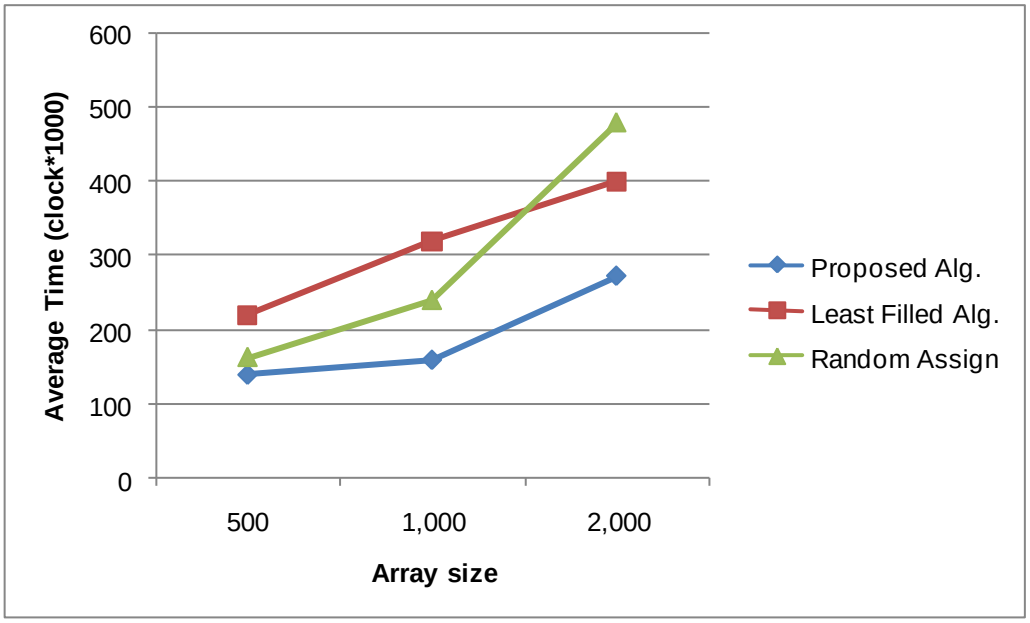


Figure 4.5. Effect of size of the object on the performance for 15 objects

Investigation effect of object size on performance with 15 objects and 5000 method calls experiment results are shown in Figure 4.5 as a graphic. On x axis object's array size is shown and on y axis average time is shown as clock ticks. Average values of results are taken as average time. According to the results proposed object distribution algorithm gives the best results compared to least filled algorithm and random assignment algorithm. Random assignment algorithm results get worse when object size increases. In random assignment object is assigned to a core randomly. Since core queues aren't considered and execution time of each method increases as array size increases. Object size is important for the proposed object distribution algorithm, because in this algorithm cache usage is taken into accunt. Increasing the cache hit rate is the main advantage of this algorithm. While object size increases, level one cache miss rate will increase. However level two cache is bigger than level one cache and stores much more data than level one cache. Staying in the same core or the other core in the same processor provides object attributes to be stored in level two cache. This decreases cache miss rate. In least filled algorithm object size or

cache miss rate isn't considered during assignment. Least filled or empty core is searched in core queues and then the assignment is done. In this case core and processor switches are done, and this event is costly and expensive. That is why proposed object distribution algorithm is better than this algorithm.

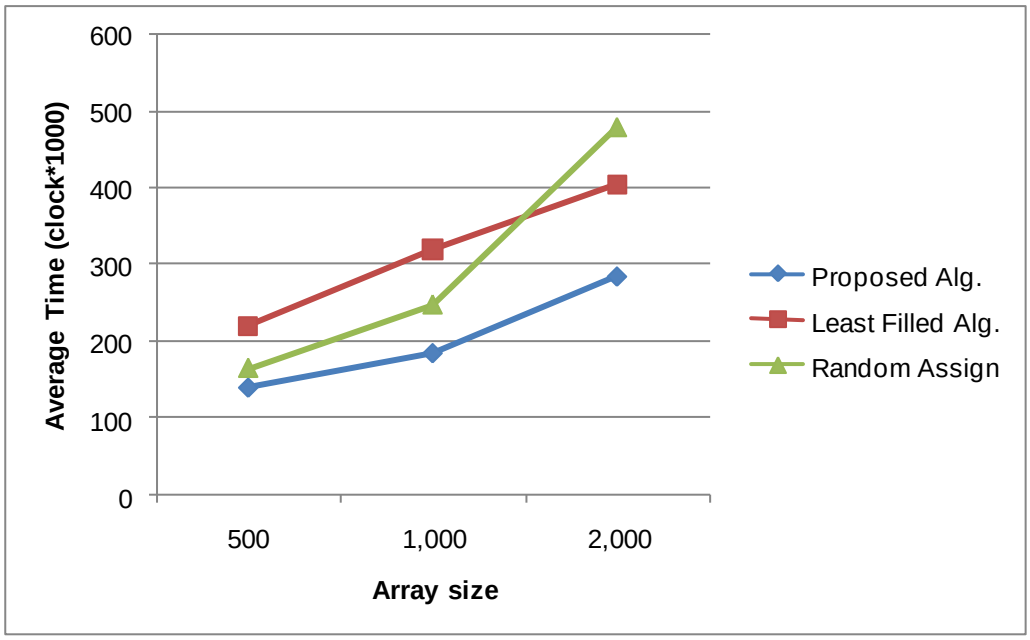


Figure 4.6. Effect of size of the object on the performance for 30 objects

Investigation effect of object size on performance with 30 objects and 5000 method calls experiment results are shown in Figure 4.6 as a graphic. On x axis object's array size is shown and on y axis average time is shown as clock ticks. Average values of results are taken as average time. According to the results proposed object distribution algorithm results are again better than least filled algorithm and random assignment algorithm results. Increasing the total object count doesn't change the results compared to previous experiments that are done with 15 objects. Least filled algorithm results are better than random assignment algorithm results, when object size is big. Since core queues aren't considered and execution time of each method increases as array size increases.

5. CONCLUSION AND RECOMMENDATIONS

The major purpose of this research was to achieve the hypothesis mentioned in the first section and propose a new object distribution algorithm. In this thesis, it is shown that distribution of objects on multicore processors is meaningful to increase cache hit rate. Distribution of objects is to create object on one of the cores of processor and then make method calls on the core which object is created. Since object oriented programming is the mostly used programming model, it isn't suitable to think function based. In multithreading on multicore processors it is more realistic and meaningful to use distribution of objects instead of distribution of functions.

Core cache has an important role on multicore processors. Generally multiprocessors have two levels of cache. Each core has its own level one cache. Level two cache is shared between the cores of processor. These caches have an effect on performance and execution times of softwares. Mostly used variables are stored in these caches. At this point mostly used attributes of objects are stored in caches of cores, too. For each method call that is made on the core that object is on, cache hit rate increases. Otherwise making method calls frequently on different cores slows down the execution time and decreases the performance.

Object size has an effect on cache usage and cache miss rate. If there are lots of mostly used attributes or attribute sizes are too large, then it is not possible to be stored in caches. It is also about object's cohesion. It is shown in the experiments that object size is important to show processor and core change is costly and very expensive. Therefore if object stays in the same core, then it is possible to access object attributes on level one cache. Similarly if the objects stays in the same processor (but it can change cores on condition that staying in the same processor), it is possible to access object attributes on level two cache that is shared between the cores that is in the same processor.

One other important factor on performance of object distribution is the number of the method calls. In this study all experiments show that high number of method calls on objects increases usage of cache more efficiently and decreases cache miss rate,

because mostly accessed data is stored on caches (level one and two), and then on method calls it is more possible to find data from caches.

In this thesis it is also shown that making method calls in same processor gives better results than making method calls in different processors. It is because of level two cache that is between the cores that is in same processor. Cores that are in same processor share this cache and this cache stores shared variables that are mostly used. If an object changes its core within the same processor, then it is more possible to access data from level two cache. If object is kept in the same core, then level one cache hit rate increases. That difference explains the performance distinction between staying in the same processor and staying in the same core. This performance boost and time benefit are shown in experiments done.

According to the experiments a new object distribution algorithm is proposed. This algorithm is about keeping object on the same core and in the same processor. Purpose of this algorithm is to decrease cache miss rate and make a smart decision on core assignment. Distribution of object is important, because object attributes are accessed almost every time whenever its method is called. Then it is suitable to keep this attributes on a cache of core and then make object's method call on this core, too. Level one cache is very small and it isn't possible to store much data on it. For lots of objects and large object sizes it isn't possible to use level one cache for performance increase. However level two cache is larger than level one cache and is able to store much more data. Thus level two cache has an important role on object distribution. Therefore keeping object in the same processor is very important.

For this subject, there are lots of future works. It is an open issue for researches. This new object distribution approach is open for developing. First of all core count, processor count and cache sizes can be found by programmatically. Core assignment mechanism can be more complex and smart by using more parameters about objects and core conditions. Moreover operating system data can be used instead of a map data structure that is used to keep queue sizes of cores. Load balancing can be added to the algorithm. Furthermore data shared by different objects can be inspected and these objects can be assigned to same processor cores.

As a conclusion the distribution of objects to cores of multiprocessors is more realistic and meaningful. The purpose of this thesis is to taking advantage of cache of cores for object oriented programming and it is supported by the experiments.

REFERENCES

[1] **Alexander Vitkalovengrc** November 21, 2005, Heterogeneous Multicore Processors, Cornell University

[2] **Alexandra Fedorova, Margo Seltzer, Michael D. Smith** 2006, Cache-Fair Thread Scheduling for Multicore Processors, *Harvard University, Sun Microsystems*

[3] **B.Ramamurthy** 2002, Process Scheduling

[4] **Charles E. Leiserson Ilya B. Mirman** 2008: How to Survive the Multicore Software Revolution, *Cilk Arts,Inc. ,* 1-3

[5] **Christoph von Praun, Luis Ceze, Calin Cas¸caval** 2008: Implicit Parallelism with Ordered Transactions, Department of Computer Science University of Illinois at Urbana-Champaign

[6] **James H. Anderson and John M. Calandrino** 2006, Parallel Task Scheduling on Multicore Platforms, Department of Computer Science, The University of North Carolina at Chapel Hill

[7] **Jernej Barbic** May 4, 2006, Multi-core architectures

[8] **Maurice Herlihy, Nir Shavit** 2008: The Art of Multiprocessor Programming, *Elsevier,* 1-5

[9] **M. Rasit Eskicioglu, T. Anthony Marsland** 1998, Scheduling

[10] **Shameem Akhter, Jason Roberts** 2006: Muticore Programming, *Intel Corporation,* 9-12

[11] **Suresh Siddha**, Multi-core and Linux* Kernel, Intel Open Source Technology Center

[12] **Uday Bondhugula, Muthu Baskaran, Albert Hartono, Sriram Krishnamoorthy1, J. Ramanujam, Atanas Rountev, P.** Sadayappan 2008: Towards Effective Automatic Parallelization for Multicore Systems, The Ohio State University.

[13] **Url-1** < http://www.redhill.net.au/c/c-1.html >, accessed at 29.02.2009.

[14] **Url-2** < http://www.embedded.com/columns/technicalinsights/183702075?_requestid=4682 >, accessed at 19.03.2009.

[15] **Url-3** < http://developer.amd.com/documentation/articles/pages/NewApproachParallelII.aspx >, accessed at 12.04.2009.

[16] **Url-4** < http://java.sys-con.com/node/419716>, accessed at 20.04.2009.

[17] **Url-5** < http://sc.tamu.edu/help/power/powerlearn/html/Origin_nw/sld004.htm >, accessed at 15.06.2009.

[18] **Url-6** < http://www.cs.berkeley.edu/~culler/cs258-s99/slides/lec05/sld035.htm >, accessed at 17.06.2009.

[19] **Url-7** < http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com .ibm.aix.p rft ungd/doc/prftungd/types_multiproc.htm>, accessed at 18.06.2009.

[20] **Url-8** < http://moss.csc.ncsu.edu/~mueller/cluster/ps3/doc/CellProgramming Tutorial/BasicsOfCellArchitecture.html >, accessed at 19.06.2009.

[21] **Url-9** < http://www.pantherproducts.co.uk/Articles/CPU/CPU%20Cache.shtml >, accessed at 20.06.2009.

[22] **Url-10** < http://www.embedded.com/columns/technicalinsights/183702075? _requestid=33391 >, accessed at 20.06.2009.

[23] **Url-11** < http://cirl.uoregon.edu/research/scheduling.html >, accessed at 22.06.2009.

[24] **Url-12** < http://www.scribd.com/doc/4838281/Operating-System-Scheduling-on-multicore-architectures >, accessed at 22.06.2009.

[25] **Url-13** < http://www.usenix.org/event/hotos07/tech/full_papers/rajagopalan/ rajagopalan_html/ >, accessed at 22.06.2009.

CURRICULUM VITA



Candidate's full name:	**Tolga Kayar**
Place and date of birth:	**Beyoğlu 23.10.1984**
Permanent Address:	**Sinemköy Değirmen Sok. No: 90/5 Kurtuluş / İSTANBUL**
Universities and Colleges attended:	**Istanbul Technical University**