

VISVESVARAYA TECHNOLOGICAL UNIVERSITY
“JNANA SANGAMA”, BELGAUM – 590014



A Project Report on
Performance Impact of NUMA Architecture on Redis
Submitted in partial fulfillment of the requirements for the award of degree of

Bachelor of Engineering

in

Information Science & Engineering

Submitted by:

Prabhanjan S K

1PI14IS036

Prajwal M R

1PI14IS037

Varsha Ramesh

1PI14IS081

Under the guidance of

Internal Guide

Dr. K V Subramaniam

January 2018 – May 2018



DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

PES INSTITUTE OF TECHNOLOGY

100 Feet Ring Road, BSK 3rd Stage, Bengaluru – 560085



**PES INSTITUTE OF TECHNOLOGY 100 Feet Ring Road, B S K 3rd Stage,
Bengaluru-560085**

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the project work entitled “**Performance Impact of NUMA architecture on Redis**” carried out by

Prabhanjan S K USN 1PI14IS036

Prajwal M R USN 1PII4IS037

Varsha Ramesh USN1PI14IS081

in partial fulfillment for the award of degree of **BACHELOR OF ENGINEERING IN INFORMATION SCIENCE & ENGINEERING** of **Visvesvaraya Technological University, Belgaum** during the year **January – May 2018**. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the above said degree.

Dr. K V Subramaniam

Professor

Dr. Shylaja S S

Professor and Head

Dr. K. S. Sridhar

Principal

External Viva

Name of the Examiners

Signature with Date

1. _____

1. _____

2. _____

2. _____

DECLARATION

We hereby declare that the project entitled “**Performance Impact of NUMA Architecture on Redis**” has been carried out by us and submitted in partial fulfillment of the course requirements for the award of degree of **Bachelor of Engineering in Information Science and Engineering** of **Visvesvaraya Technological University, Belagavi** during the academic semester January – May 2018. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

USN	Name	Signature
1PI14IS036	Prabhanjan S K	
1PI14IS037	Prajwal M R	
1PI14IS081	Varsha Ramesh	

ACKNOWLEDGEMENT

We are grateful to PES Institute Of Technology for providing us a wonderful platform, necessary infrastructure and opportunity to work on this project. We are also thankful to all our faculty for providing us with guidance and suggestions during the course of this project.

We are thankful to Dr K.V Subramaniam , Professor , Computer science and Engineering department for providing continuous advice and expert guidance during this project.

We also want to thank Kanishka Lahiri from AMD for his expert guidance about NUMA architecture of AMD processors.

ABSTRACT

The aim of this project is to analyze the effect of NUMA architecture on the performance of REDIS (a real-time in-memory database).

NUMA stands for Non-Uniform Memory Access. It is an architecture which enables a particular core in the processor to gain faster access to a particular part of memory and hence enabling better performance termed as local and remote accesses. In a real time, in-memory database like redis which reads a lot of data over a long period of time, making more local accesses than remote accesses should impact performance.

In this project we perform various experiments to study the behavior of redis (as a specimen to real-time memory intensive applications) and deduce the effect of making redis a NUMA aware system by using the linux flags such as numactl which enables forcing local and remote accesses.

We perform all analysis on a multi socket machine and perform experiments forcing local accesses and forcing remote accesses to study performance and the reason underlying the same.

We produce the results of the experiments in the form of graphs which gives a clear picture of the experiment results.

LIST OF FIGURES

Figure number	Title	Page number
6.1	Uniform memory architecture	16
6.2	Non-uniform memory architecture	17
6.3	Numa model : Intel perspective(a)	18
6.4	Numa model : Intel perspective(b)	19
6.5	Caching model	20
6.6	AMD Numa model	22
6.7	Example of hierarchy in linux scheduling domains	26
7.1	Sample output from pcm numa	28
7.2	pcm.x output showing the machine configuration	31
7.3	Hardware abstraction numactl level	31
7.4	pcm.x-memory output showing memory configuration	32
7.5	Detailed output of pcm.x	33
7.6	Benchmark results snapshot	60
8.1	Number of local access performed by experiment 5 with number of keys inserted as 10^6	61
8.2	Number of local access performed by experiment 5 with number of keys inserted as 10^7	62

8.3	Number of remote accesses performed by experiment 5 with number of keys inserted as 10^7	63
8.4	Number of remote accesses performed by experiment 6 with set size of 10^6	63
8.5	Cycles per instruction vs time performed by experiment 5 with number of keys inserted as 10^6	64
8.6	Cycles per instruction vs time performed by experiment 5 with number of keys inserted as 10^7	64
8.7	IPC vs time elapsed for experiment 7 on sorted set size of 10^6	65
8.8	IPC vs time elapsed for experiment 7 on sorted set size of $2 \cdot 10^6$	65
8.9	Comparison of benchmark results	67

LIST OF TABLES

Table number	Title	Page number
7.1	Wall clock time observations for experiment 1	36
7.2	Wall clock time observations for experiment 2	39
7.3	Wall clock time observations for experiment 3	44
7.4	Wall clock time observations for experiment 5	52
7.5	Wall clock time observations for experiment 6	58

TABLE OF CONTENTS

ACKNOWLEDGEMENT

ABSTRACT

LIST OF TABLES

LIST OF FIGURES

1. Introduction
2. Problem Definition
 - 2.1. Detailed problem definition
 - 2.2. Generic proposed solution
3. Literature Survey
4. Project Requirement Specification
 - 4.1. Timeline
 - 4.2. Gantt chart
5. System Requirements Specification
 - 5.1. Functional Requirement
 - 5.2. Non-Functional Requirement
 - 5.3. Hardware Requirement
 - 5.4. Software Requirement
6. System Design
 - 6.1. High level NUMA vs Traditional architecture
 - 6.2. NUMA Architecture
 - 6.2.1. NUMA Model : INTEL Perspective
 - 6.2.2. NUMA Model : AMD Perspective
 - 6.3. Generic method to compute NUMA distances
 - 6.4. Linux Scheduler
7. Implementation
 - 7.1. Implementation Details
 - 7.2. Intel Performance Counter Monitor
 - 7.3. Experiments on Intel Broadwell
 - 7.3.1. Experiment 1
 - 7.3.2. Experiment 2
 - 7.3.3. Experiment 3
 - 7.3.4. Experiment 4
 - 7.3.5. Experiment 5
 - 7.3.6. Experiment 6
 - 7.3.7. Experiment 7
8. Results and Discussions
9. Conclusion
10. Further Work
11. References

CHAPTER 01

INTRODUCTION

In the recent years there has been drastic improvements in hardware which has enabled to extend innovations in the fields of Big data, Machine learning, AI, Computer vision etc. Hence advancements in high performance computing has become a very important aspect of innovation in general. With the advancements of hardware the increasing need to support and utilize the same becomes necessary and has benefits over the native or traditional approach and gives significant improvements and better results.

Traditional hardware systems can provide only limited performance with single core or multiple logical cores that can run threads in parallel. They also do not scale well, It does not have high levels of concurrency and parallelism and a single socket machine connected to a memory through the DDR channels and accessing any part of memory takes the same time and therefore optimisations cannot be thought about. Therefore NUMA architecture was introduced making the system more powerful and software systems leveraging the underlying architecture and make programs perform better.

Effects of NUMA architecture is not used by default and is a different concept compared to the traditional approach, as said earlier all memory accesses are the same and therefore optimizations cannot be done in traditional systems but in NUMA based systems which have much higher levels of concurrency and parallelism and has multiple physical cores and multi sockets connected to its own memory through it own DDR channel which introduces local and remote memory access and not surprisingly these two does not take the same time and that is where the performance impact and optimizations taking into account that accessing a local memory attached to node faster than accessing the remote memory attached to another node. So the operating system abstracts the processes and the utilisation of the underlying architecture and therefore Linux Operating System released newer versions of kernel with a rebuilt Scheduler

Performance Impact of NUMA architecture on Redis

called the Completely Fair Scheduler (CFS) but the Linux operating system by default does not take these into account and all of these options are provided as flags which can be used by the

software developers to gain performance improvements and one such thing is called the numactl(stands for numa control), using which a process can be forced to use local or remote memory. The performance benefit got by doing local accesses than remote accesses depend on the architecture of NUMA model, computed by something called the numa distance between nodes.

Redis is a real-time in-memory database. It is written in ANSI C and works in most systems like linux , Ubuntu , BSD , OS X etc without any additional dependencies.

It is used in applications of various nature and scale. In the industry Redis can be used in applications storing few gigabytes to hundreds of terabytes of data depending on the nature of the application. For example , Many websites use Redis to hold login tokens to validate web sessions. Twitter uses redis to store its trending tweets.

Redis can store data structures such as lists , strings , sets , sorted sets , hashes and other unconventional data structures such as hyperlogs , geospatial data etc. Redis has built-in replication on different levels of on-disk persistence. It also provides high availability and automatic partitioning of data.

Redis can be used for use cases where atomic operations are required such as appending to a string , incrementing a value in hash , pushing an element into a list , computing set intersection , union , difference ..etc.

Redis works with in-memory dataset to get high performance. It can be used to dump data into disk in order to persist data or it can be used as a real-time in-memory cache which can only store data in memory for time being.

Performance Impact of NUMA architecture on Redis

It supports features such as transactions , publish/subscribe , Lua scripting , keys with limited time to live , LRU eviction of keys , automatic failover etc.

Hence the scale can vary to a great extent. Hence Redis is a perfect specimen to perform experiments to analyze of NUMA architecture on different scales of applications

Making a system NUMA aware involves studying the underneath architecture of the machine, computing the number of available NUMA Nodes , Figuring out the NUMA model of the processor , modifying the execution of processes , making changes to the execution environment and making sure that more local accesses are made.

Analyzing memory access patterns in different scenarios will give us a clear picture of the impact that NUMA can have. We perform various such experiments with various numactl flags and various data structures available in redis and collect data to analyze different memory access patterns and hence study the.

CHAPTER 02

PROBLEM DEFINITION

2.1 Detailed Problem Definition

In a typical uniform memory architecture, it is frequently observed that there is a large gap between the rate at which the processor processes data and for memory accesses to actually fetch the data from the random access memory. Consequently, the processor is often left hungry for data and a number of cycles are wasted in the processor just waiting for memory access operations to return.

To overcome this drawback, non uniform memory access design was introduced wherein each processor has its own fast memory that is referred to as the “local memory”. The cores on the socket that share this fast memory are collectively termed as a NUMA node. Also, the cores of a NUMA node can also access the memory of other NUMA nodes, which are called “remote memory”. Unfortunately, an access to the remote memory of a node has higher latency than that of a local memory access. As a result, the application running on a node should use the remote memory of the node as minimally as possible.

To analyze the performance impact of NUMA architecture on Redis. Real time applications such as Redis can have huge variety of memory access patterns. Our experiments aim to analyze the impact of having a NUMA architecture for Redis. Making an application NUMA aware can vary the performance differently in different scenarios.

We perform experiments with various scenarios and memory access patterns to analyze the behavior of the application on NUMA architecture and Traditional architecture and make a comparison between them to arrive at a conclusion.

Performance Impact of NUMA architecture on Redis

We make use of various tools to study and collect data about various performance measures in order to analyze the impact of NUMA architecture under different circumstances.

2.2 Generic Proposed Solution

The proposed approach to analyze the performance impact is to run various memory intensive experiments on a NUMA machine and capture various parameters with local and remote access and making a comparison analysis between the two sets of data to deduce the impact of NUMA architecture. By using performance and benchmark tools available with redis like the redis-benchmark or the ycsb(yahoo cloud serving benchmark) and study the impact of forcing local and remote access and understand the difference in the performance and conclude as to the fact that making redis NUMA aware indeed impacts performance and also analyzing the scale of improvement.

CHAPTER 03

LITERATURE SURVEY

3.1 Literature survey

The prerequisite to our study of the performance of NUMA architecture on real time applications was twofold - to understand the non uniform memory access architecture in a great depth i.e with respect to the operating system as well as at the processor level, and to study optimisations adopted by other real time applications to make them aware of the underlying architecture. Our literature survey was directed towards the same.

To get a better understanding of the way the operating system in general, we need to get an idea of how the operating system perceives the memory of a NUMA model. [1] compares and contrasts linux's view of the memory architecture in uniform memory access architecture and non uniform memory access architecture. The key differences that they point out are with respect to the memory model.

In any operating system, the memory manager is responsible for two key operations - memory allocation and memory reclaim to the processes. In case of a uniform memory access model, the memory management is done by the page allocator. Every process requests the page allocator for memory, which it hands out to the process in chunks of page size. The pages thus allocated can be broadly categorized into two categories - anonymous pages, which are essentially temporary pages used by the process for storing variables, heap area etc... These pages do not have a corresponding mapping to a page on the disk; and we have page caches which have a corresponding page on the disk. These can possibly contain the executable instructions of the process.

On the other hand, in a NUMA architecture, the memory model is slightly different. As opposed to a uniform model, we need a page allocator for every node on the system because every node

Performance Impact of NUMA architecture on Redis

has its own local memory which needs to be managed. The page allocator that is present on every NUMA node is referred to as the swapper. Hence, the swapper thread is responsible for the allocation and reclaim of memory in a NUMA system.

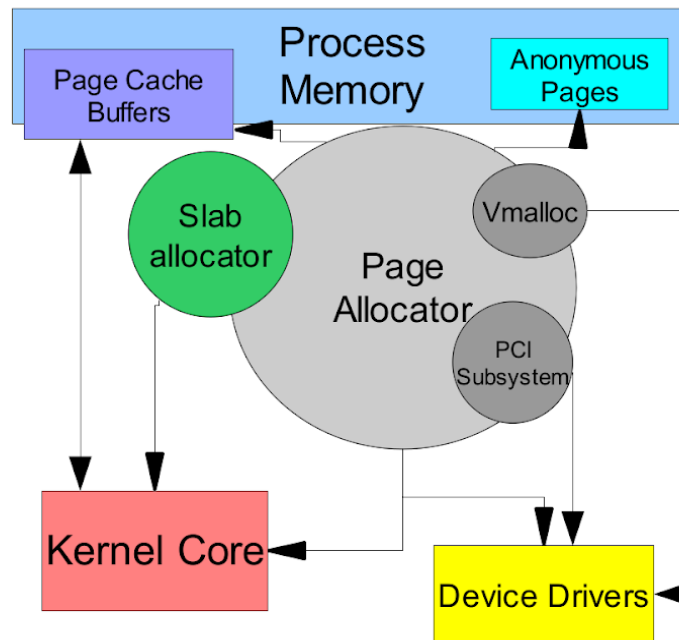


Figure 3.1 Linux memory subsystems

Getting an insight into the operating system's perspective of the NUMA model is necessary for our future experiments.

Going further, we need to understand the optimisations adopted by other real time applications so that we can potentially leverage the same to make Redis recognize the underlying architecture and take appropriate action for the same.

Gmail and Google search are large scale applications that handle a large number of requests on a daily basis. [2] explores the impact of NUMA architecture on these server. The authors perform

Performance Impact of NUMA architecture on Redis

an analysis for optimising warehouse scale computers, that are Gmail backend servers and the web search front end servers. These servers are handling real time workloads and hence it becomes an even more challenging task to study the performance impact of the underlying architecture. This is owing to the many factors that are likely to influence the performance such as the configuration of the machines, the workload on each cluster etc... However, a load test is designed keeping in mind the various factors and the experiments are conducted on a few selected clusters of servers. They also conduct load test to study the tradeoff between optimising NUMA performance and cache contention on applications such as BigTable, cluster-docs and Web-search frontend reader. They propose the computation of a locality score and study its correlation with performance metrics such CPI, CPU utilisation etc... NUMA optimisations have a significant effect at the production level servers. With respect to Gmail servers, they yielded a performance benefit of almost 15% while in case of the web search front end, they yielded a performance benefit of 20%. These are significant numbers to be seen in the production scenarios. Nevertheless, in some cases (such as BigTable), the application yields better performance when all the memory accesses are remote. This is due to the cache contention. Thus, improvement in performance resulting from NUMA optimisations are constrained by the design of the application itself.

The results obtained by [2] give us reason to believe that we can expect similar yet scaled down benefits with Redis as well, if it is made NUMA aware. Afterall, Redis is an extensive application that is widely used in the industry as well, handling data in the order of terabytes. The design and architecture of the application of Redis need to be slightly tweaked to leverage this potential improvement in performance.

To get some ideas on the we studied the optimisations adopted by certain applications. The most common of them being the node local strategy i.e to schedule the process on a node which contains the data that the process is working on. Though this works with most applications, this needs to be done with caution as it can lead to issues such as contention for shared resources as highlighted in [3].

Cache contention is a common issue that occurs when processes are scheduled on cores of the same node, wherein the cores essentially share a cache, typically the last level cache. Each of the processes contend for the cache. If the process relies heavily on the use of caches, it can prove to be very expensive when only half the cache is effectively available to the process. With the node local strategy, cache contention is like to show up. However, the tradeoff between the localisation of data and cache contention is worth exploring. The proposed 'maximum-local' algorithm studies the effect of the same.

The paper also proposes a refined N-MASS scheduling algorithm that takes into account cache contention by computing a NUMA penalty factor. A combination of memory intensive and CPU intensive workloads are used for experimental purposes and the performance of the scheduled processes are measured.

Overall, if the memory allocation in the system is balanced, then maximum-local scheduling provides large performance benefits as the cache contention is not that significant. In other cases, it is necessary to refine the local mapping (as with N-MASS) to avoid degradation in performance. However the work is based on a number of assumptions such as all the cores of a processor share an LLC. They have also ignored the influence of additional factors such as garbage collectors, other processes etc.. on the results.

This is relevant to our work as it gives us insight about the factors that we need to consider prior to introducing optimisations in redis.

Applications are generally written with high levels of abstraction wherein a number of sub tasks are performed by existing external libraries. For instance, most applications rely on runtime libraries such as OpenMP for the creation of the threads to perform certain tasks. In these cases, the application itself is not aware of how the threads are being created by the operating system and how they are being scheduled by the thread scheduler. If the underlying architecture is NUMA, the threads of an application need to be scheduled close to one another i.e on the same

node, so that the data that is shared among those threads are all present in the local memory of the node on which the threads are running. Thus, the thread-data affinity needs to be

communicated to the scheduler beforehand so that the application does not suffer any penalty in terms of performance owing to the system architecture whether it is NUMA or not. But these cannot be handled or communicated that easily to the scheduler as it is mostly available only during runtime. The above effects are studied in [4].

They also introduce a NUMA-aware memory manager that is used by a scheduler to solve thread and memory issues. The next-touch policy introduced in this paper is a generalisation of the first touch policy. In this approach, the application asks the system to migrate the page to memory associated with a core on which a thread that will next access it is running. However, this is challenging to implement as a number of ambiguous situations arise. For instance, the case where two threads on different cores accessing the same zone will give rise to a conflict. We learn that thread/memory affinity does matter mainly because of congestion issues in modern NUMA architectures. The multi-level thread scheduler improves performance considerably. However, the next-touch policy might not always give optimal performance as it is not aware of the memory load of the target node.

Since redis is also a multi-threaded application, these factors are relevant points to be guarded while considering scheduling threads of redis-server onto the NUMA node.

Understanding memory access patterns of a process is the key to performing well with respect to memory. Redis supports a number of data structures such as strings, lists, sets, sorted sets, hash maps etc... These structures have different use cases in real time applications. Thus, we come across a number of different access patterns, with varying granularity of the data that is accessed from the respective data structure.

Performance Impact of NUMA architecture on Redis

For example, the granularity of memory access in a Java application is that of an object as opposed to pages. [5] exploits this fact to propose a NUMA aware Java heap layout. This paper studies the effect of dynamic page migration of Java objects on Java applications running on cc-NUMA architectures. Since objects tend to span across pages and also have different access

patterns, this optimisation technique is not quite promising. Hence, they propose two NUMA aware Java heap layouts - NUMA aware young generation and NUMA aware young and old generation. These layouts are based on the static-optimal, prior-knowledge and object migration techniques. The proposed Java NUMA-aware heap layouts always reduced the total number of non-local object accesses compared to the original Java heap layout. Using both the NUMA aware young and old generations combined with dynamic object migration is better than that of only the young generation. However, no object migration is performed on the eden space because of high infant mortality rate.

While considering redis data structures as well, we need to take into consideration the granularity of the data being accessed as well.

There are also other factors that influence the effect of well-known optimisations such as asymmetric interconnect as emphasized in [6]. In certain multi-processor systems, the interconnect between nodes are of varying bandwidth. The authors design a thread and memory placement algorithm that delivers better performance. The algorithm works in 3 stages - measurement, decision and migration. Migration of pages is implemented by a their own system call that gets rid of the overhead introduced due to locking in traditional linux migration call. The performance of NUMA systems is heavily impacted by the bandwidth of the interconnect between the nodes more than the distance. Optimizations that targeted at placements with optimal bandwidth interconnects performed better than those that simply counted the number of hops.

Processor architecture plays a pivotal role in determining the way NUMA workloads behave. We have focussed on the Intel Broadwell and AMD Ryzen architectures to get a NUMA picture.

CHAPTER 04

SCHEDULE

4.1 Timeline

Jan 2 - Jan 10	Literature survey
Jan 11 - Jan 20	Understanding processor architecture of AMD and INTEL
Jan 21 - Jan 30	Understanding NUMA models of AMD and Intel processors
Feb 1 - Feb 20	Understanding the linux source code and libnuma library source code
Feb 21 - Feb 28	Acquiring and setting up machine
March 1 - March 15	Experiments and analysis with c programs with local and remote memory access
March 16 - March 28	Experiments and analysis with Redis programs with local and remote memory access
March 29 - March 30	AMD sessions
March 31 - April 10	Experiments and analysis with YCSB
April 11	Redis workshop

Performance Impact of NUMA architecture on Redis

April 12 - April 20	Experiments with Intel PCM tool
April 20 - April 30	Report writing

4.2 Gantt Chart

Gantt Chart

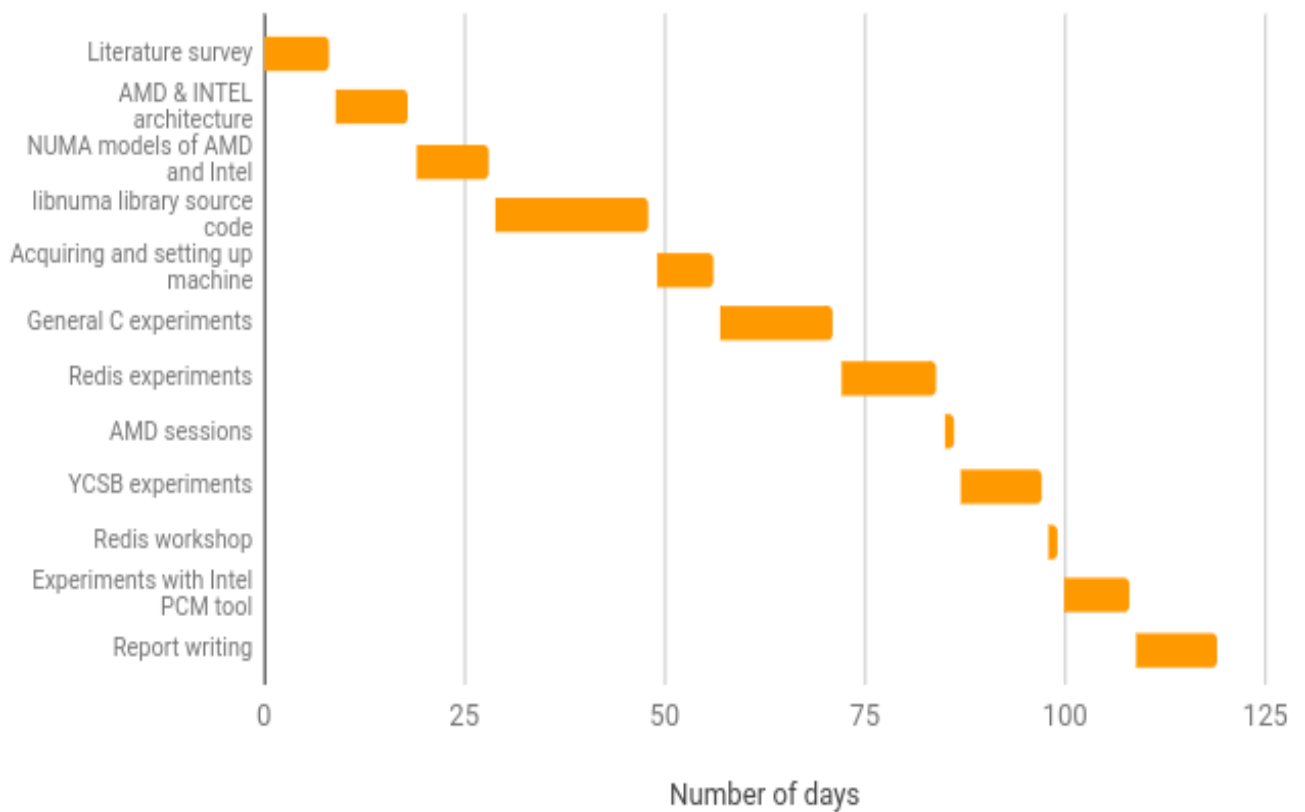


Figure 4.1 Gantt Chart

CHAPTER 05

SYSTEM REQUIREMENTS SPECIFICATIONS

5.1 Functional Requirements

- Understanding the NUMA based architecture of both Intel and AMD machines.
- Understanding the working of Linux Scheduler in general and also specific to NUMA architecture and how the scheduler takes the underlying NUMA architecture into account.
- Understanding Redis as an application, write C/C++ code to interact with redis and run redis commands.
- Compile and build Linux Kernel from its source code on a VM.
- Make changes in the Linux Kernel to incorporate these NUMA aware scheduling and study performance impacts on real time database applications like Redis.
- Make changes in the Redis source code to make it NUMA aware and study the impact of performance.
- Understand how to tie a process to a particular core and also analyze the memory being used the process.
- Understand how to tie a process to use the memory either local memory or remote memory and also run tests to make sure that a process can be forced to access local memory or remote memory explicitly.
- Run sample programs like sort random numbers, random access to an array with these explicit forcing of the program to use local memory and remote memory and analyze the results using tools in linux.
- Run the sample programs for different array sizes and different number of accesses and time the program for each case of local memory bound and remote memory bound processes.
- Run sample programs of redis using the C library hiredis for a few data structures like key-value pairs,lists,sets and use data such as numbers,random length strings stored in

Performance Impact of NUMA architecture on Redis

them and accessing them to modify it and study the impact of the same with forcing the local and remote memory access controls bound to the redis server.

- Run the sample redis programs for different sizes of data structures and different kinds of data and time the program for each case of redis server bound to use local memory and redis server bound to use remote memory.
- Understand the usage of pcm tool by Intel and analyze the results of the pcm-numa and pcm-memory tool by running the above sample redis code and redis server tied to use local memory and remote memory in separate cases by isolating the core under study and plotting the graphs and understanding the outcomes of the same.

5.2 Non Functional Requirements

- Studying and improving performance of real time database applications like Redis.
- Exploring various different techniques and optimizations to utilize the NUMA based architecture to improve performance.

5.3 Hardware requirements

- Multicore generic VM on AWS.
- Intel Broadwell Multicore processor system.
- AMD Ryzen Multicore processor system.

5.4 Software requirements

- PCM tools.
- Redis library
- Hiredis Library.
- Linux source code.

CHAPTER 06

SYSTEM DESIGN

6.1 High level NUMA vs Traditional architecture

Traditional Architecture

Traditional architecture involves multiple processors sharing a single BUS to the memory. This becomes a bottleneck and does not allow optimal utilization of processing power of the processors. Each processor will waste multiple cycles in waiting for memory access through the bus.

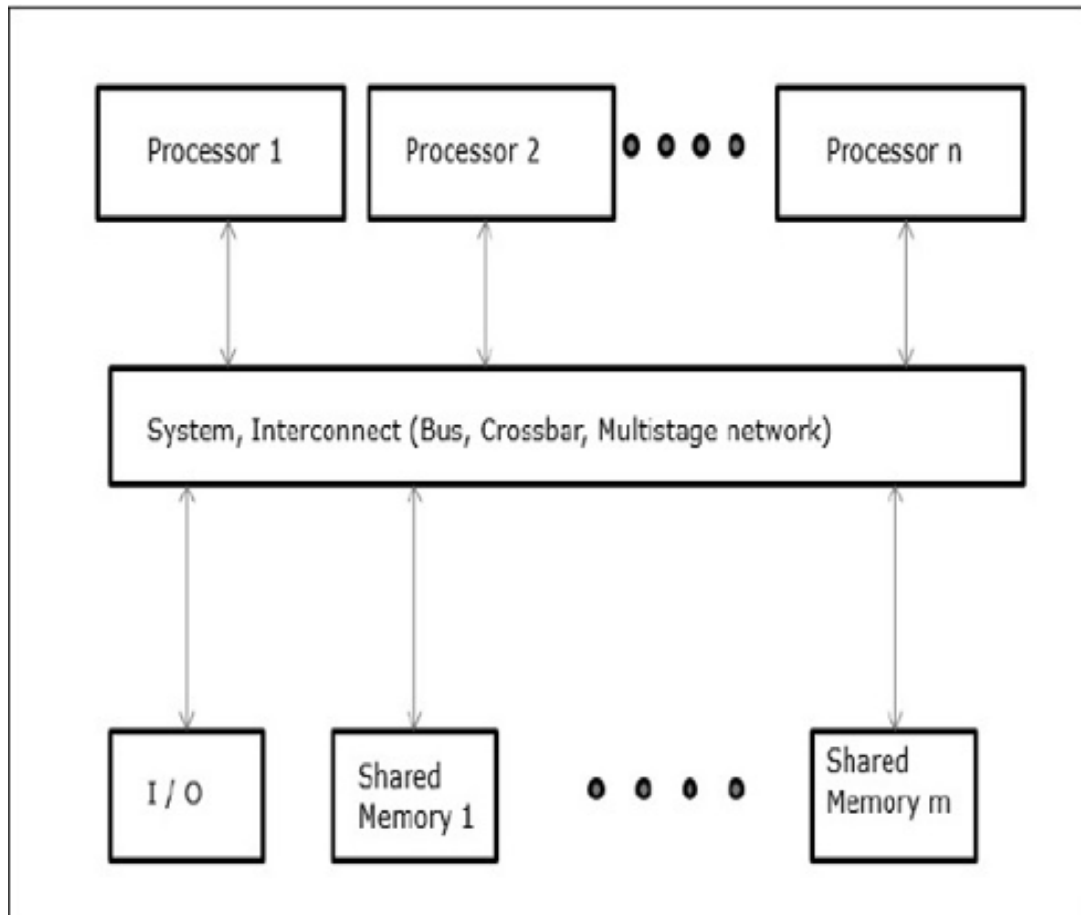


Figure 6.1 Uniform memory architecture

6.2 NUMA Architecture

NUMA architecture has different parts of memory directly accessible to different processors. This means that local memory access is fast and optimal for the particular core. But the access to other parts of memory (remote access) has to go through the bus, hence will be slower. Thus NUMA allows to leverage the locality of data to make processing faster.

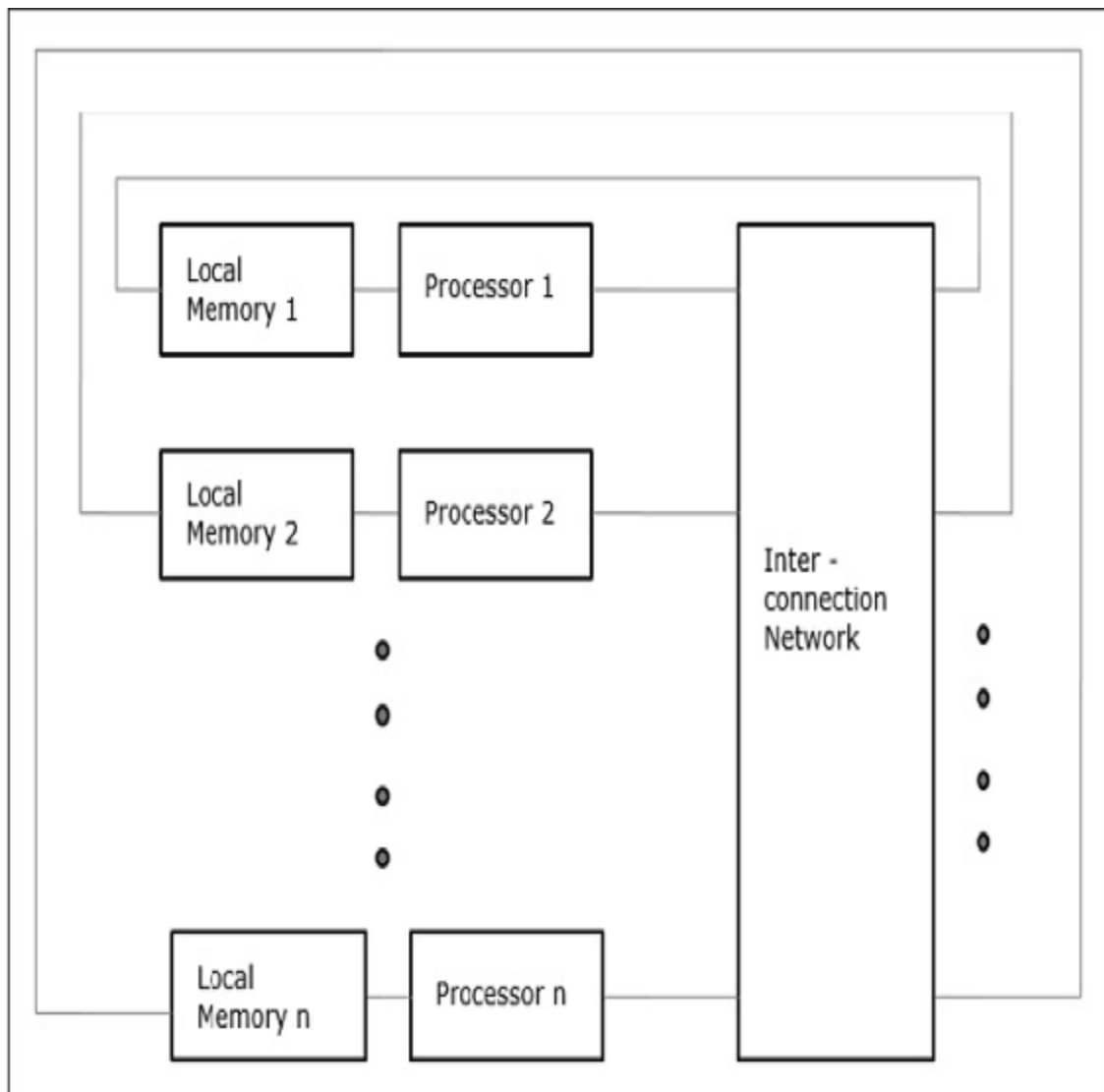


Figure 6.2 Non-Uniform memory architecture

6.2.1 NUMA Model INTEL Perspective

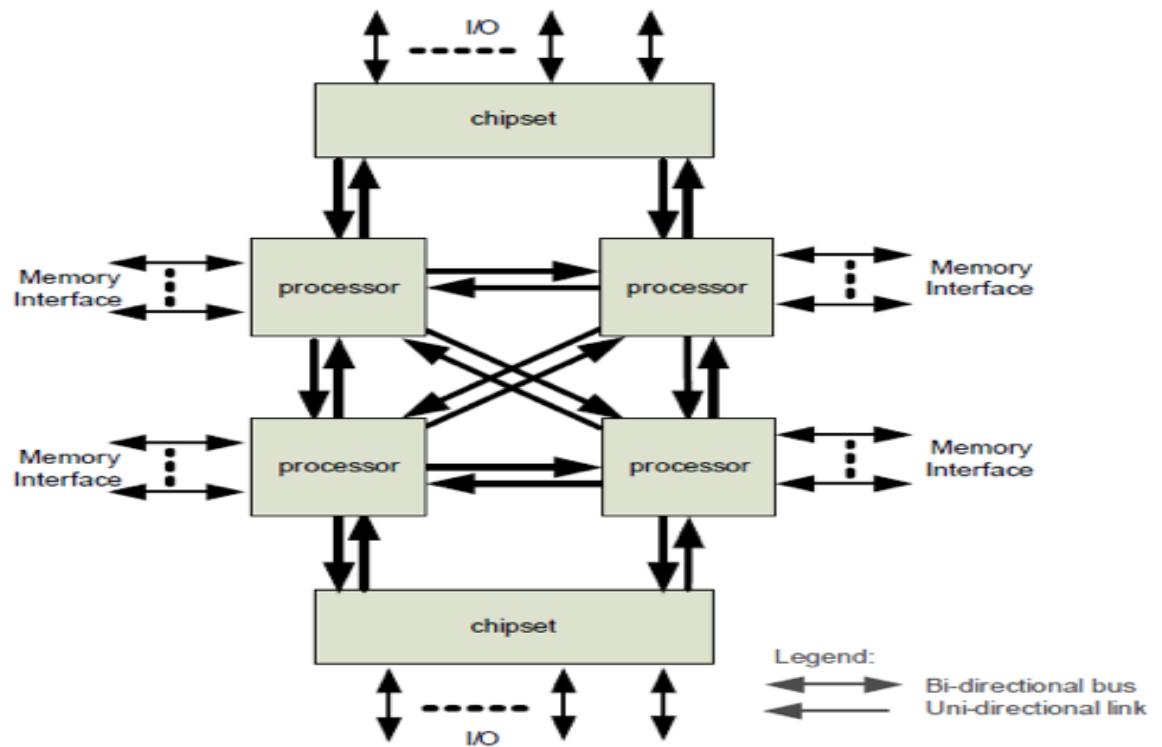


Figure 6.3 Numa model Intel perspective(a)

As far as the NUMA architecture of Intel processors is concerned, it essentially consists of a number of cores that have independent DRAM links and also interconnected through the Quick Path Interconnect (QPI). QPI is Intel's point-to-point interprocessor connect technology that is supposed to provide fast communication between the cores on a die. The exchange of data is duplex and follows a protocol specified by a five layered architecture.

In our project, we have focused on the Intel Broadwell architecture which is essentially Intel's 14 nanometer die. It is regarded as a "tick" of the existing Haswell microarchitecture in the Intel tick-tock jargon.

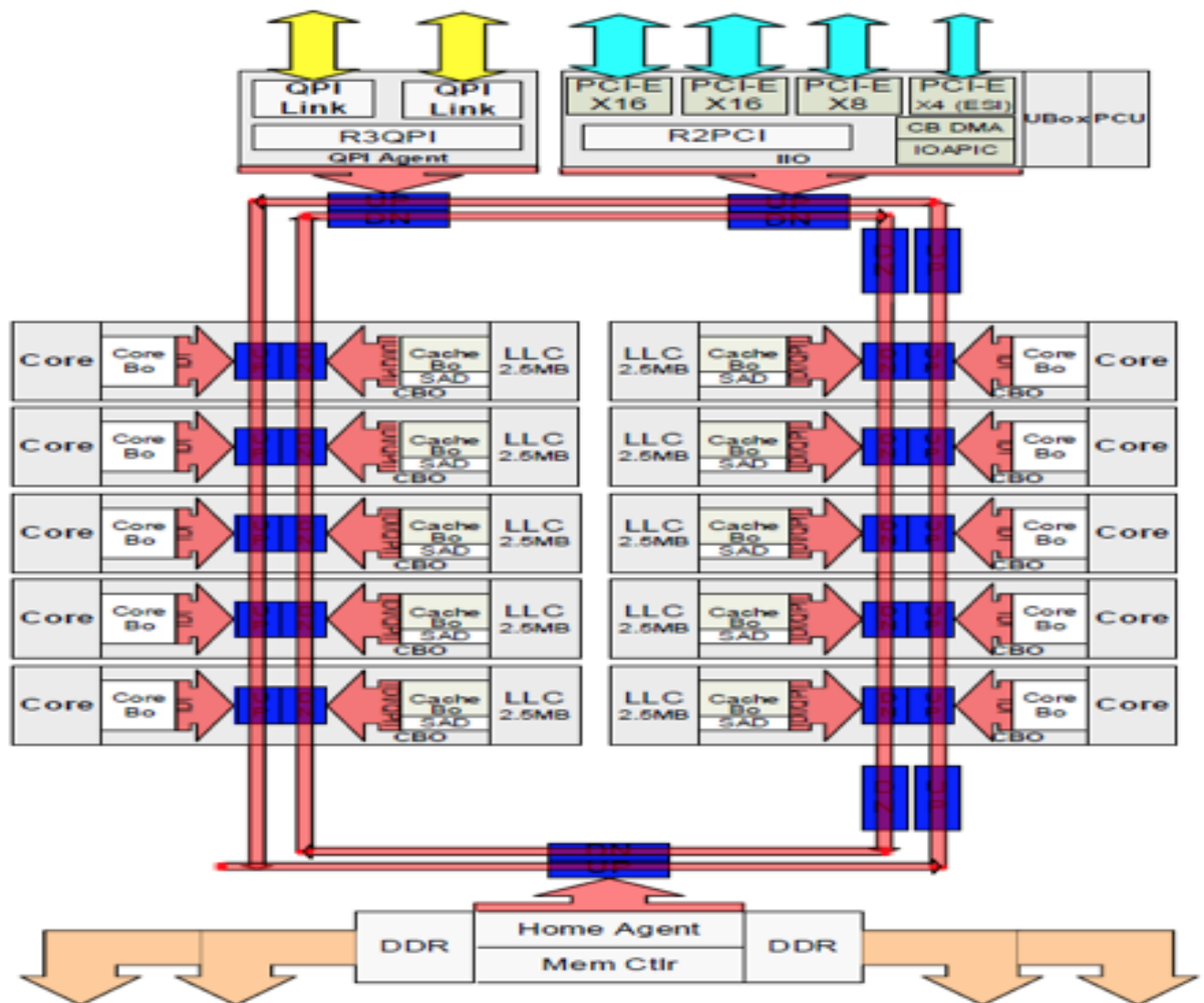


Figure 6.4 Numa model Intel perspective(b)

There are three main types of dies that we can categorize Intel processors into - Low Core count (LCC) consisting of upto 10 cores, Medium Core Count(MCC) consisting of 12 to 14 cores and High Core Count(HCC) consisting of more than 16 cores. The area of these dies increases with the increase in core count as well.

As far as the memory of the broadwell microarchitecture is concerned, it mostly consists of an 8 way associative L1 cache. This cache is local to every core. The L1 cache has two instances - one that serves as a data cache and the other as an instruction cache. Followed by the L1 cache,

is the 8 way associative cache that is local to every core as well. Lastly, we have the Last Level Cache (LLC). The LLC is shared by the cores on a die.

Each of these caches is connected to a caching agent that takes part in cache coherence protocols which ensures that the data shared by the caches is the same. Snooping modes are for cache coherence.

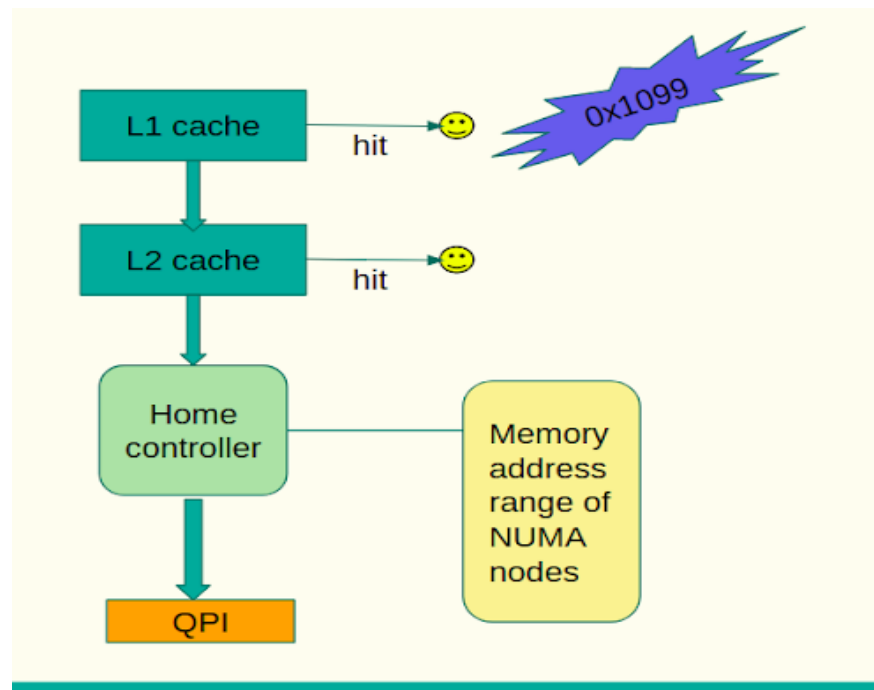


Figure 6.5 Caching model

When a particular memory address data is to be fetched from memory, then the data is first checked for in L1 cache of the core in which the memory address is required. If the data is absent in the L1 cache, then the data is looked for in the L2 cache. If found, the same data is returned to core requesting the memory access. If the L2 cache also fails, then the request is passed to the home controller in the processor for further processing.

In a NUMA architecture, the home controller maintains the address range of the existing NUMA nodes. Hence, the home controller will resolve the memory address corresponding to the respective node. The data is then fetched from the actual NUMA node over the Quick Path Interconnect link.

Though there is an QPI link between every pair of nodes in the NUMA architecture, the latency for a remote memory access may not be uniform. This is because the nodes may be physically located at different distances from the node requesting the memory access. This is one of the factors that affects the remote memory access latency. This varying remote memory access latency is computed as the NUMA distance of every node. The NUMA distance of a node from itself is considered to be ten, which is the lowest distance possible. Thus, the NUMA distance of a node from another gives us an idea about how much more expensive a remote memory access is, relative to a local memory access. For example, if the NUMA distance between a the home node and the node on which the memory address resides is 21, it means that a remote memory access is likely to be 2.1 times slower than a local memory access. However, note that these numbers are just approximations computed as a combination of a number of factor. Hence, in reality, these numbers are just probable, not exact.

6.2.2 NUMA Model : AMD Perspective

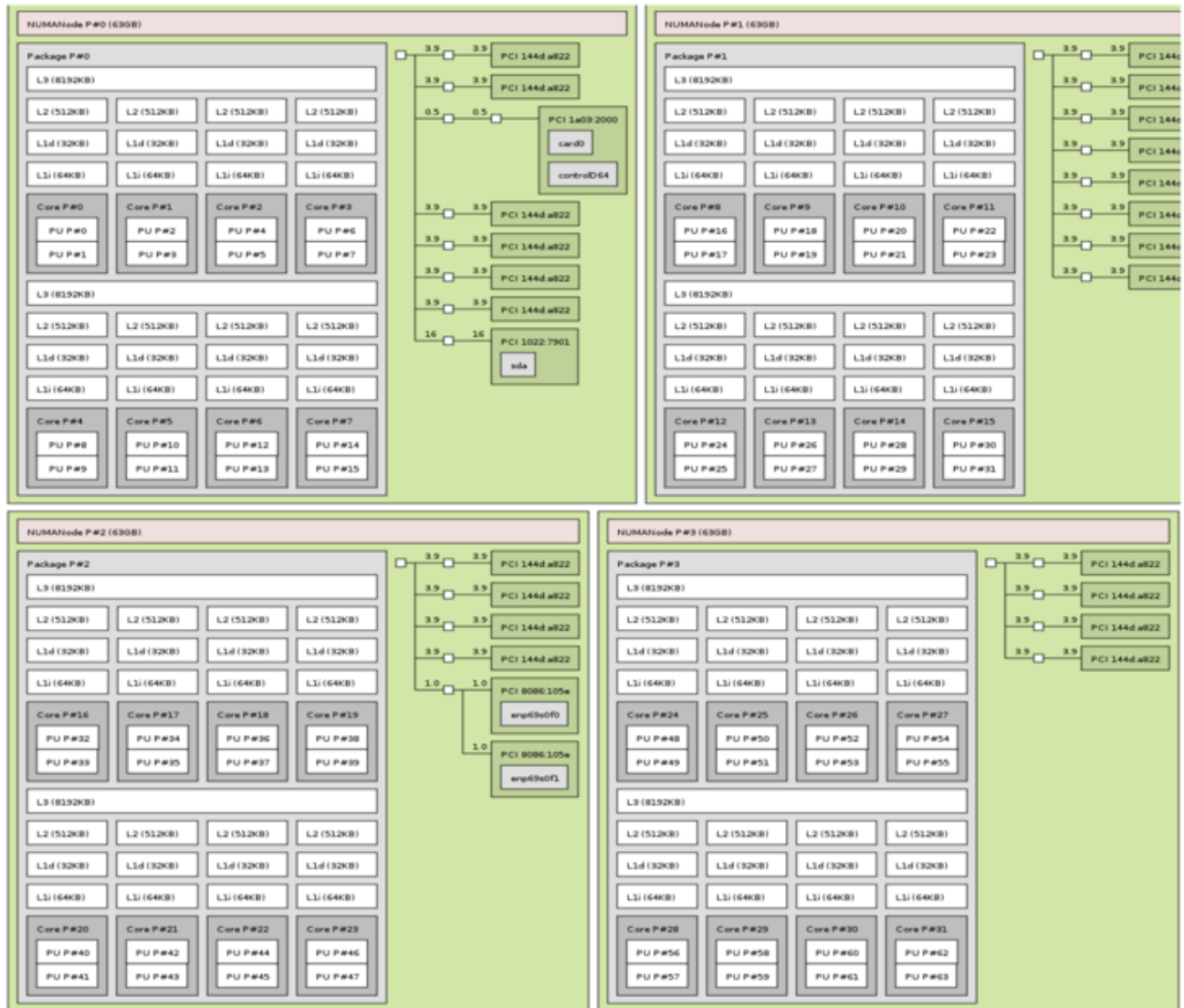


Figure 6.6 AMD Numa model

AMD processor is divided into 4 dies. Each die has 2 core computing complexes(CCX). Each core computing complex has 4 “zen” cores each. Each of those cores share a L3 cache and has its own L1 and L2 caches. Hence It has 8 physical cores per die. Hence the processor has 32 cores in total. Linux considers each of these dies as a numa node.

Inter-die communications happens via a low latency infinity fabric. Each die internally connects with multiple SATA , NVMe and PCIe device.

Although it may seem that the distance between two nodes is very large in a system with multiple NUMA nodes, the architecture is so designed any remote memory accesses does not take more than two hops. This is achieved by connecting every pair of core complexes itself so that every remote access is directed to the appropriate core complex in the first hop to the appropriate node in the core complex in the subsequent hop. AMD has ensured that no remote memory access takes more than two hops.

6.3 Generic method to compute NUMA distances

NUMA distance represents how far another node is located with respect to a node. NUMA distance determines the access speed between two nodes. NUMA distance is hardcoded in the system locality information table (SLIT) by the manufacturer of the processor , which is then picked up by linux for process scheduling purposes.

NUMA distance for each node can be seen under the path :

`/sys/devices/system/node/node<index>/distance`

the node index starts from 0 and goes up to the number of nodes present in the machine.

Inside the distance file , we have a vector representing distance to each other node , including to itself. NUMA distance is 10 for itself which is the minimum distance and it is calculated in the multiples of 10 to each adjacent node.

For example, inside the Node0/distance file we have a vector [10 , 21] which says the distance to itself is 10 and distance to Node1(adjacent) is 21. Inside the Node1/distance file we have an array [21 , 10] which says distance from Node1 to Node0 is 21 and to itself is 10.

Performance Impact of NUMA architecture on Redis

Numa distance is calculated using the formula :

distance_table[a * distance_numnodes + b]

where:

- **distance_table** is a matrix populated by the linux numa library using the above mentioned distance files.
- **“a”** is the ordinal of the first node
- **“b”** is the ordinal of the second node
- **“distance_numnodes”** is the highest number of numa nodes available in the system.

6.4 Linux Scheduler

Scheduler is the part of kernel that selects the processes to run on the underlying architecture and utilize the hardware resources like CPU and give the process a chance to complete its desired work. There has been several versions of the linux scheduler. It is the basis of multitasking and it is responsible for best utilizing the hardware system and also in the context of NUMA architecture this becomes important because of the fact that the scheduler is the one which would put a process on a particular core and therefore the further effects of what all memory accesses the process needs would be either remote or local based on the core to which the process was scheduled by the scheduler. The linux scheduler can be found under the kernel/sched.c. The newer and improved features of the linux scheduler included the following things

- Implement fully O(1) scheduler which means that the scheduler is constant time and therefore gains a huge improvement in scheduling and also puts more concern if the underlying architecture is NUMA and the scheduler as it is constant time maybe doesn't make the resources to figure out if the process is put on which core will it make more local accesses then remote access and benefit from doing the same.
- Implement improved SMP(Symmetric MultiProcessing) affinity which means that certain interrupts or processes are assigned to certain processors and has an affinity towards that.
- Runqueues - This is the basic data structure used in the scheduler and the concept of runqueue is that it houses the list of runnable processes on a given processor and it is

Performance Impact of NUMA architecture on Redis

specific per processor. Also each runnable process is on exactly one runqueue and it also contains per-processor scheduling information. It is accessible through a group of macros such as `cpu_rq(processor)` returns a pointer to the runqueue of that processor.

- SMP applies to cores as well treating them as separate processors.

With all these new advances and improvements the underlying NUMA architecture is not taken into picture and optimizations are not performed. Operating System scheduling on multicore

architecture looks similar to the one single core design above and a naive approach would be to use N runqueues, but this has several limitations under the hood and a few issues would be that level 1 cache is managed by each core (L3 cache in recent architectures) and it becomes even worse if the architecture is not just multicore but NUMA as memory accesses can be very costly if the process was scheduled on wrong core in terms of where the memory was local to that core. Linux recently changed the scheduler to CFS (completely fair scheduler) which uses Red-Black Trees (Time ordered rb trees) and the CFS algorithm chooses the leftmost task from the tree. Linux load-balancer takes care of different cache models and computing architectures but at the moment not necessarily of performance asymmetry behaviours. The underlying model of the Linux load balancer is the concept of scheduling domains, which was introduced in Kernel version 2.6.7 due to the unsatisfying performance of Linux scheduling on SMP and NUMA systems in prior versions. Basically, it is a hierarchical set of computational units where scheduling is possible. The scheduling domain architecture is constructed based on the actual hardware resources of the computing element.

Scheduling domains are hierarchically nested – there is a top-level domain containing all other domains of the physical system the Linux is running on. Depending on the actual architecture, the sub-domains represent NUMA node groups, physical CPU groups, multi-core groups or SMT groups in a respective hierarchical nesting. This structure is built automatically based on the actual topology of the system and for reasons of efficiency each CPU keeps a copy of every domain it belongs to. For example, a logical SMT processor that at the same time is a core in a physical multi-core processor on a NUMA node with multiple (SMP) processors would totally administer 4 `sched_domain` structures, one for each level of parallel computing it is involved in.

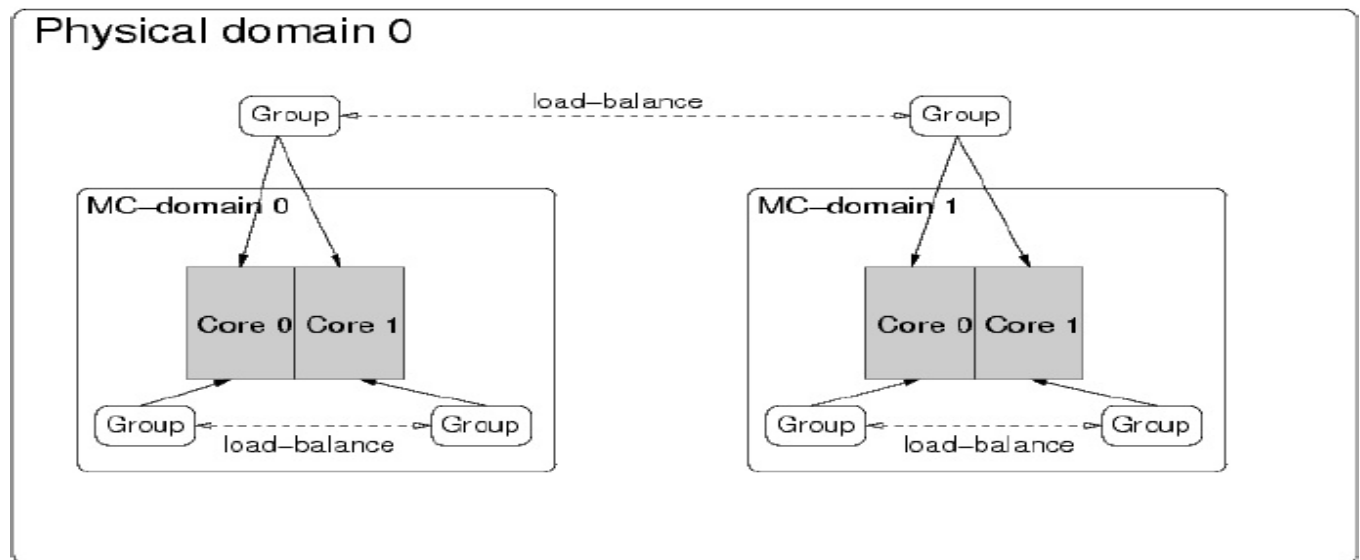


Figure 6.7 : Example of hierarchy in linux Scheduling domains

CHAPTER 07

IMPLEMENTATION

7.1 Implementation Details

In order to analyze and perform experiments with sample programs that utilize both CPU and memory resources we first come with a way to understand and pin a process to a particular core using tasker commands which explicitly forces the process to run on that core and this can be specified using the htop commands, using the numa control flags available as command line options in Linux we can go one step further and force remote or local access when running a program. The basic command of numactl -H outputs the NUMA hardware structure inside and how it is used and viewed by the software. So by using other command line options that can be specified for the numactl command we can control the way the program runs and make sure it makes more or all local memory accesses or remote memory accesses. --cpunodebind=node is an command line option that can be specified in order to tell the operating system that the process has to run on the specified node(a group of cores,any one of them).The flag physcpubind is another command line options that can be specified in order to tell the operating system to run the process on the specified cores in the cpu. --membind=node is another command line options which specifies that the program uses the memory from the node given.

numactl --cpunodebind=0 --membind=0 ./a.out 1000 would make the program run on any core in node 0 and use the memory associated with node 0 which would mean that this is the case of local memory accesses and this is how it can be specified with numactl command.

numactl --cpunodebind=0 --membind=1 ./a.out 1000 would make the program run on any core in node 0 and use the memory associated with node 1 which would mean that this is the case of remote memory accesses and this is how it can be specified with numactl command.

Now with more advanced tools like pcm(process counter monitor) we can gain more knowledge about the way the program runs and get detailed information about the exact things happening

the program. With the use of pcm-numa we can get the IPC, cycles, instructions, number of local DRAM accesses and number of remote DRAM accesses and this can be controlled at the level of time interval to write new information into the log file, so the command `pcm-numa.x 1 -csv=test.log` will write the output every second to a file called test.log. With the use of pcm-memory we can monitor memory bandwidth (per-channel and per-DRAM DIMM rank).

```

≡ local_access_10_6.log x
1 |Time elapsed: 10003 ms
2 |Core,IPC,Instructions,Cycles,Local DRAM accesses,Remote DRAM accesses
3 |0,0.93,1445800676,1553910006,1045489,85432,
4 |1,0.42,452913713,1074586445,1323247,273842,
5 |2,0.24,371246797,1570023757,1240086,14146,
6 |3,0.96,2177110302,2257586859,972243,8472,
7 |4,1.01,2195575390,2182847687,1324411,42491,
8 |5,1.04,1520860494,1460107247,943951,11436,
9 |6,1.00,1832657773,1827213697,759939,5377,
10|7,1.01,781570059,770639459,1170849,37944,
11|8,0.41,5477889,13332740,8495,3225,
12|9,0.39,7264808,18662890,5301,3151,
13|10,0.37,4601948,12411023,4011,1578,
14|11,0.38,3403680,8947026,3251,1592,
15|12,0.23,2304327,10167061,3684,1687,
16|13,0.32,1464157,4647323,1195,163,
17|14,0.29,1361340,4695475,1213,1475,
18|15,0.29,704413,2470626,559,119,
19|16,1.31,427739086,326245474,882913,5301,
20|17,1.24,256127152,207304790,723696,3242,
21|18,1.18,249972280,211140974,717383,4918,
22|19,0.73,9309774849,12750527186,10757621,5094,
23|20,1.24,177574644,143329412,514043,2525,
24|21,0.96,1857573887,1932130438,501456,4687,
25|22,1.05,869249483,826795004,647210,6204,
26|23,1.26,256348681,203803088,725286,6700,
27|24,0.42,8610945,20401209,6307,344,
28|25,0.33,2659113,8000226,1620,230,
29|26,0.28,870889,3115624,1628,1629,
30|27,0.28,690213,2453183,656,165,
31|28,0.29,2406179,8380632,1484,71,
32|29,0.28,635410,2292487,529,76,
33|30,0.28,596262,2113995,567,179,
34|31,0.34,946785,2760687,549,261,

```

Figure 7.1 Sample output from pcm-numa

So in order to run the program with local accesses and then use the pcm-numa and pcm-memory to perform analysis we run the following commands

Performance Impact of NUMA architecture on Redis

- `numactl --cpunodebind=0 --membind=0 ./a.out 1000`
- `pcm-numa.x 1 -csv=report.log`

For using the same with remote accesses

- `numactl --cpunodebind=0 --membind=1 ./a.out 1000`
- `pcm-numa.x 1 -csv=report.log`

Now in order to use the same with redis programs , we have two components which can be forced with this behaviour. One component is the redis server which can be run using the

redis-server command and the other one is the custom C program that uses the hiredis library and use data structures like key-value pairs,lists and sets and the running of this program will be analyzed for time,cycles,instructions,number of local DRAM accesses and remote DRAM accesses. So we use the hiredis C library to connect to the redis-server and run redis commands from the C program.The program can be compiled with gcc using “gcc program.c -lhiredis” command. So after this compilation we can run ./a.out with the numactl commands but this is just the C client program that talks to the redis-server and runs the specified commands in the program and therefore the numactl commands need to be used with running the redis-server command. What needs to be done effectively is firstly run the redis-server command by using the numactl flags and then compile and run the C program with performance monitoring tools like time command, pcm-numa and pcm-memory commands. So we will see examples of how this can be done.

In order to analyze local memory accesses bound with redis-server and then run time and pcm commands for the C program

- `numactl --cpunodebind=0 --membind=0 ./redis-server`
- `pcm-numa.x 10 -csv=report.log -- ~/path/to/executable/program args`
- `pcm-memory.x 10 -csv=mem.log -- ~/path/to/executable/program args`

In order to analyze remote memory accesses bound with redis-server and then run time and pcm commands for the C program

Performance Impact of NUMA architecture on Redis

- `numactl --cpunodebind=0 --membind=1 ./redis-server`
- `pcm-numa.x 10 -csv=report.log -- ~/path/to/executable/program args`
- `pcm-memory.x 10 -csv=mem.log -- ~/path/to/executable/program args`

The following runs the redis-server with either forced local or remote accesses and then runs the pcm-numa tool and writes values every 10 seconds which would be analyzed and then normalised in order to plot graphs and understand the things in details.

In order to clearly understand the things happening under the hood we need to be sure that the analytics done by the tool and values are only due to the programs that are being run by us and therefore we isolate a few cores or one core in order to run redis-server and only analyze the results for that core only. So in order to isolate a core we use the add the following line to the `/etc/default/grub` file

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=1"
```

The following command would isolate core 1 and only kernel-space tasks would run on it and no user-space tasks would be scheduled on this core unless we explicitly schedule it to that core using taskset or --physcpubind option using numactl command.

In Order to isolate a core and then run the same analysis for local memory accesses we use the following commands in the same order

- Add `GRUB_CMDLINE_LINUX_DEFAULT="quiet splash isolcpus=20"` line in the `/etc/default/grub` file.
- `sudo grub-update`
- `sudo reboot`
- `numactl --physicalcpubind=19 --membind=0 ./redis-server`
- `pcm-numa.x 10 -csv=report.log -- ~/path/to/executable/program args`
- `pcm-memory.x 10 -csv=mem.log -- ~/path/to/executable/program args`

7.2 Intel Performance Counter Monitor

The tool we have use pcm (process counter monitor) has variety of options to run like pcm.x,pcm-numa.x and therefore the output of these commands are shown which makes it more clear to understand the same also gives a clear picture of the machine we are working with all the hardware infrastructure the machine has

```
Number of physical cores: 16
Number of logical cores: 32
Number of online logical cores: 32
Threads (logical cores) per physical core: 2
Num sockets: 2
Physical cores per socket: 8
Core PMU (perfmon) version: 3
Number of core PMU generic (programmable) counters: 4
Width of generic (programmable) counters: 48 bits
Number of core PMU fixed counters: 3
Width of fixed counters: 48 bits
Nominal core frequency: 2100000000 Hz
Package thermal spec power: 85 Watt; Package minimum power: 44 Watt; Package maximum power: 170 Watt;
Socket 0: 2 memory controllers detected with total number of 5 channels. 2 QPI ports detected.
Socket 1: 2 memory controllers detected with total number of 5 channels. 2 QPI ports detected.
```

Figure 7.2 pcm.x output showing the machine configuration

```
root@ubuntu:/home/prabha/project/pcm# numactl -H
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 6 7 16 17 18 19 20 21 22 23
node 0 size: 32061 MB
node 0 free: 31226 MB
node 1 cpus: 8 9 10 11 12 13 14 15 24 25 26 27 28 29 30 31
node 1 size: 32253 MB
node 1 free: 31125 MB
node distances:
node  0  1
  0: 10 21
  1: 21 10
```

Figure 7.3 The hardware abstraction at numactl level

Socket 0		Socket 1	
Memory Channel Monitoring		Memory Channel Monitoring	
Mem Ch 0: Reads (MB/s):	19.95	Mem Ch 0: Reads (MB/s):	28.37
Writes(MB/s):	22.42	Writes(MB/s):	20.63
Mem Ch 1: Reads (MB/s):	17.07	Mem Ch 1: Reads (MB/s):	24.67
Writes(MB/s):	19.53	Writes(MB/s):	16.94
NODE 0 Mem Read (MB/s) :	37.02	NODE 1 Mem Read (MB/s) :	53.04
NODE 0 Mem Write(MB/s) :	41.95	NODE 1 Mem Write(MB/s) :	37.57
NODE 0 P. Write (T/s):	44976	NODE 1 P. Write (T/s):	57791
NODE 0 Memory (MB/s):	78.97	NODE 1 Memory (MB/s):	90.61
System Read Throughput(MB/s):		90.06	
System Write Throughput(MB/s):		79.52	
System Memory Throughput(MB/s):		169.58	

Figure 7.4 : pcm-memory.x output showing memory configurations

Performance Impact of NUMA architecture on Redis

```

Detected Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz "Intel(r) microarchitecture codename Broadwell-EP/EX" stepping 1

EXEC : instructions per nominal CPU cycle
IPC : instructions per CPU cycle
FREQ : relation to nominal CPU frequency='unhalted clock ticks'/'invariant timer ticks' (includes Intel Turbo Boost)
AFREQ : relation to nominal CPU frequency while in active state (not in power-saving C state)='unhalted clock ticks'/'invariant timer ticks while in C0-state' (includes Intel Turbo Boost)
L3MISS: L3 cache misses
L2MISS: L2 cache misses (including other core's L2 cache *hits*)
L3HIT : L3 cache hit ratio (0.00-1.00)
L2HIT : L2 cache hit ratio (0.00-1.00)
L3MPI : number of L3 cache misses per instruction
L2MPI : number of L2 cache misses per instruction
READ : bytes read from main memory controller (in GBytes)
WRITE : bytes written to main memory controller (in GBytes)
L3OCC : L3 occupancy (in KBytes)
LMB : L3 cache external bandwidth satisfied by local memory (in MBytes)
RMB : L3 cache external bandwidth satisfied by remote memory (in MBytes)
TEMP : Temperature reading in 1 degree Celsius relative to the TjMax temperature (thermal headroom): 0 corresponds to the max temperature
energy: Energy in Joules

Core (SKT) | EXEC | IPC | FREQ | AFREQ | L3MISS | L2MISS | L3HIT | L2HIT | L3MPI | L2MPI | L3OCC | LMB | RMB | TEMP
0 0 0.00 0.61 0.00 0.57 22 K 136 K 0.84 0.10 0.00 0.02 5056 0 1 68
1 0 0.00 0.27 0.00 0.57 1780 10 K 0.83 0.20 0.00 0.01 160 0 0 67
2 0 0.00 0.19 0.00 0.57 998 7530 0.87 0.20 0.00 0.01 320 0 0 68
3 0 0.00 0.22 0.00 0.57 1387 9570 0.86 0.19 0.00 0.01 128 0 0 68
4 0 0.00 0.23 0.00 0.57 1060 7198 0.85 0.25 0.00 0.01 256 0 0 68
5 0 0.00 0.14 0.00 0.57 1147 7198 0.84 0.18 0.00 0.01 608 0 0 68
6 0 0.00 0.23 0.00 0.57 1855 11 K 0.84 0.18 0.00 0.01 1216 0 0 67
7 0 0.00 0.32 0.00 0.57 3031 18 K 0.84 0.31 0.00 0.01 160 0 0 66
8 1 0.01 0.99 0.01 0.67 460 K 557 K 0.17 0.37 0.00 0.01 544 25 21 65
9 1 0.01 1.38 0.01 0.61 448 K 516 K 0.13 0.34 0.00 0.00 704 22 20 65
10 1 0.01 0.81 0.01 0.65 529 K 657 K 0.20 0.37 0.01 0.01 384 31 21 66
11 1 0.01 1.19 0.01 0.62 311 K 356 K 0.13 0.38 0.00 0.00 544 15 12 66
12 1 0.01 1.49 0.01 0.61 379 K 434 K 0.13 0.37 0.00 0.00 384 19 19 66
13 1 0.01 1.10 0.01 0.87 428 K 475 K 0.10 0.39 0.00 0.00 512 28 20 67
14 1 0.01 1.08 0.01 0.63 365 K 438 K 0.17 0.39 0.00 0.00 5056 18 17 67
15 1 0.01 0.17 0.07 0.99 778 K 1433 K 0.46 0.43 0.01 0.01 9792 124 24 65
16 0 0.00 0.23 0.00 0.57 1627 8648 0.81 0.13 0.00 0.02 224 0 0 68
17 0 0.00 0.28 0.00 0.57 1385 6606 0.79 0.15 0.00 0.01 160 0 0 67
18 0 0.00 0.24 0.00 0.57 1759 12 K 0.86 0.13 0.00 0.02 512 0 0 68
19 0 0.00 0.23 0.00 0.57 1299 9594 0.86 0.13 0.00 0.02 192 0 0 68
20 0 0.00 0.24 0.00 0.57 1552 9319 0.83 0.16 0.00 0.01 32 0 0 68
21 0 0.00 0.27 0.00 0.57 2886 27 K 0.90 0.15 0.00 0.02 192 0 0 68

```

Figure 7.5 Detailed output of pcm.x

7.3 Experiments on Intel Broadwell

7.3.1 Experiment 1

Objective

To understand the impact of continuous remote memory accesses (including read and writes) and check if it has a significant impact on performance in terms of the time of execution.

Design of experiment

Create an array of size 'n' integers by dynamically allocating memory from the heap area of the process. This array is initialised with random numbers at the time of creation. These numbers are then sorted using the quicksort algorithm in $O(n \log n)$ time.

We can now bind this process to a particular node and specify the memory allocation for the array using the numactl command as described above.

Having bound the process to local and remote memory, we can then time the process takes to complete i.e initialise the array and sort the random number. This process is repeated for different array sizes and the time taken for local and remote memory allocations is recorded.

By varying the allocation of memory for the array from local to remote, we expected to see a degradation in performance, because the array was being accessed from the remote memory of the node on which the process was running. Thus, we anticipated a lower time of execution for the local memory allocation case.

Pseudocode

generate_rand_numbers(a, n):

```
// Generates 'n' random numbers and populates into array a  
// Input : Pointer to an array 'a'  
// Output : Array 'a' populated with random numbers  
  
seed <- getpid()  
Initialise random number generator seed  
i <- 0  
while i < n do  
    a[i] <- generated random number  
    i <- i + 1  
return a
```

compare_function(a, b):

```
//compares two integers 'a' and 'b' and return 1, 0, -1 in case of a > b,  
// a=b and a < b respectively  
//Input : Two integers 'a','b'  
//Output : 0 or -1 or 1  
  
if a > b  
    return 1  
if a = b  
    return 0  
if a < b  
    return -1
```

main:

```
n <- input from command line  
// n is the size of the array
```

```

a <- malloc an 'n' sized array
generate_rand_numbers(a, n)
//use quicksort library function to sort the number and pass compare_function as
comparator
qsort(a,n,sizeof(int),compare_function);
    
```

Observations

Size of array	Local memory access		Remote memory access		Ratio TR/TL
	Core	Time (in seconds) -TL	Core	Time (in seconds) -TR	
100000000	5	14.83	26	14.94	1.007
200000000	11	30	3	29.32	0.977
300000000	10	45	11	44	0.977

Table 7.1 Wall clock time observations for experiment 1

Results

As we observe, there is not much variation in the time of execution in case of local and remote memory allocation of the array.

Inference

This can be attributed to the effect of caches in the execution of a process. Essentially, when we access the first element of the array, it is likely to page fault as the array is not present in the

cache. However, after the first element is fetched, the rest of the block of the array should ideally be fetched owing to the spatial locality rule of caching. Thus, the entire array or a large part of it is already present in the cache of the core on which the process is running. Since no other process is accessing the same data, this content remains valid throughout the process. Thus, there is no need for the core to request a DRAM access for the data. Remote memory and local memory refer to DRAM accesses, which is likely not plentiful in this experiment. As a consequence, we do not see much difference in the execution time.

7.3.2 Experiment 2

Objective

As seen above, the effect of caches is influencing our study of performance. However, in real world applications, it is highly unlikely that consecutive memory accesses are sequential in nature. To simulate a genuine of random scenario of random memory accesses, we considered accessing the random memory locations wherein consecutive memory accesses may or may not be part of the same cache block.

Design of the experiment

Create an array of size 'n' and populate it with random integers. The elements of the array are once again allocated from the heap area of the process. Now, generate a random index of the array to be accessed and perform a simple update operation on the randomly chosen element of the array. We can now expect the performance to be a function of both the array size 'n' as well as the number of update operations performed on randomly generated array indices.

As mentioned above, we bind the process to a particular node and force the memory allocation to be local or remote, as the case may be using the numactl command. We can then time the process to check the performance in case of local and remote memory allocations of the array.

Since the indices of the array elements to be updated are generated randomly, we expect at least a few of them to be in different cache blocks. Thus, at least a few consecutive memory accesses will not be under the influence of caching and we expect to see a degradation in performance or equivalently a higher execution time due to the remote memory access latencies.

Pseudocode

generate_rand_numbers(a, n):

// Generates 'n' random numbers and populates into array a

// Input : Pointer to an array 'a'

// Output : Array 'a' populated with random numbers

seed <- getpid()

Initialise random number generator seed

i <- 0

while i < n do

a[i] <- generated random number

i <- i + 1

return a

perform_rand_access(a, n):

// Randomly accesses elements of array 'a' and increments them by 1

//Input : An array 'a' and 'n' as the size of the array

//Output: Random elements of array incremented

seed <- getpid()

Initialise random number generator seed

i <- 0

while i < n do

Performance Impact of NUMA architecture on Redis

```
index <- generate random number
```

```
a[index] <- a[index] + 1
```

main:

```
n <- command line argument input
```

```
Allocate an array 'a' from the heap of size n*sizeof(int)
```

```
generate_rand_numbers(a,n)
```

```
perform_rand_access(a,n)
```

Observations


Size of array	Number of accesses	Local memory access		Remote memory access		
		Core 	Time (in seconds) -TL	Core	Time (in seconds) -TR	Ratio TR/TL
3×10^8	3×10^8	-	3.4	-	3.7	1.088
9×10^8	10^7	11	9.7	3	9.8	1.01
10^{10}	10^{10}	26	76	3	110	1.45
10^{11}	10^{11}	26	68	19	96	1.412
10^{10}	10^{11}	11	66	17	104	1.58

Table 7.2 Wall clock time observations for experiment 2

Results

As we can see from the table, for small array sizes, there is not much significant difference in the time of execution. However, for the last entry where array size is 1010, and number of update operations or equivalently number of accesses being 1011, we see the ratio of 1.58. This is likely to be approaching the expected NUMA distance ratio of 2.1. As mentioned above, since the

NUMA distance is only an approximation of the relative decrease in performance, we can only expect to approach a value close to it as opposed to the exact ratio of 2.1.

Inference

It can be inferred that when memory accesses are random, i.e nullifying the effect of caches, remote memory accesses introduce a penalty in performance. This penalty is significant only in case of large data structures as well as large number of memory accesses. This is the kind of memory access pattern we expect to see in actual application dealing with real world data. Thus, there is a potential for improvement.

7.3.3 Experiment 3

Objective

Having observed a difference in a simulated situation, we can now check for performance impact on a real world application such as Redis. As mentioned above, Redis is an in memory database. So we can proceed to create some basic data structures in redis and perform some operations on them to check for NUMA architecture impact.

Design of the experiment

Redis is essentially a key-value pair database wherein the value can be of several data structures. In this experiment, we deal with the list data structure. Lists in redis are essentially implemented as doubly-linked lists with $O(1)$ insertion and deletion operations while insert and update operations take $O(n)$ time in the worst case.

We create three lists, and insert randomly generated strings into each of the three list alternatively. This can be done using the LPUSH command in redis

i.e LPUSH mylist1 'abc' pushes the element 'abc' into the list 'mylist' and also creates it if it does not already exist.

Having pushed random strings into the three lists, we can now access each of the lists sequentially and perform an update operation. To access the elements of the list sequentially, we use the LRANGE command as follows

```
LRANGE mylist1 0 -1
```

To perform the update operation we use LSET command as follows

```
LSET mylist1 old_value, new_value
```

We now vary the size of the list and check the time of execution for local and remote memory accesses. We can also monitor the number of local DRAM accesses and remote DRAM accesses using the pcm numa tool as described above.

Pseudocode

generate_rand_string():

```
seed <- process_id + num_calls
```

```
// num_calls is the number of times this function has been called so far
```

```
set random number generator seed
```

```
string_len <- generate random number in the range of 1-20
```

```
rand_string <- ""
```

```
i <- 0
```

```
while i < string_len:
```

```
do
```

```
rand_ascii_code <- generate random number in the range of 65 to
```

122

```
char_rand <- convert rand_ascii_code to character
```

Performance Impact of NUMA architecture on Redis

```
    append char_rand to rand_string
    i <- i + 1
append '\0' to the rand_string
return rand_string
```

main:

```
n <- command line argument input
// n indicates the size of the list to be created

// Establish connection to redis server
// This is done using the hiredis API

set timeout for redis connection to be 1.5 seconds
hostname <- 127.0.0.1
port <- 6379
context <- redisConnectWithTimeout(hostname, port, timeout)

// We can now redis commands use the redisCommand API

/*
We have to create three lists and populate them
Let's name the lists as mylist1, mylist2, mylist3
Initially, we have to delete these lists if they already exist
*/
redisCommand(context, "DEL mylist1")
redisCommand(context, "DEL mylist2")
redisCommand(context, "DEL mylist3")
```

Performance Impact of NUMA architecture on Redis

/*

Now that we have deleted the lists, we can create the three lists, mylist1, mylist2, mylist3 and insert elements into them using the LPUSH command

Note that the LPUSH command inserts elements to the head of the list always

*/

i <- 0

while i < n do

command <- "LPUSH mylist1 " + generate_rand_string()

redisCommand(context, command)

command <- "LPUSH mylist2 " + generate_rand_string()

redisCommand(context, command)

command <- "LPUSH mylist3 " + generate_rand_string()

redisCommand(context, command)

i <- i + 1

/*

Now we can access the elements of the list using the LRANGE command and replace it with a new random string that is generated

To perform the update operation we use the LSET command

*/

reply <- redisCommand("LRANGE mylist1 0 -1")

i <- 0

while i < reply->elements->size() do

command <- "LSET mylist1 reply->element[i] " + generate_rand_string()

redisCommand(context, command)

i <- i + 1

reply <- redisCommand("LRANGE mylist2 0 -1")

i <- 0

while i < reply->elements->size() do

Performance Impact of NUMA architecture on Redis

```
command < "LSET mylist2 reply->element[i] " + generate_rand_string()
redisCommand(context, command)
i <- i + 1
```

```
reply <- redisCommand("LRANGE mylist3 0 -1")
i <- 0
while i < reply->elements->size() do
  command < "LSET mylist3 reply->element[i] " + generate_rand_string()
  redisCommand(context, command)
  i <- i + 1
```

Observations

Sl.No	Size of the list	Local memory allocation execution time (in seconds)	Remote memory allocation execution time (in seconds)
1	10^5	17	16
2	10^6	197	201
3	10^7	497	501

Table 7.3 Wall clock time observations for experiment 3

Results

As we observe, there is not much significant difference in the execution times in case of local and remote memory allocation.

Inference

As described in the sections above, redis is an in memory data store i.e it acts as a cache store for commonly accessed keys. In our experiment, we are accessing just three keys corresponding to the three list values. We can easily expect redis application to retrieve the entire list and cache it when the first element of the list is being accessed. This explains why the read operation is fast. However, we are also performing a an update or equivalently a write operation. The data that is present in the cache is updated and since no dependency exists as the elements of the list are all independent, it can be written to memory in a separate thread in parallel. This is why even the update operations are not slowing down a lot.

7.3.4 Experiment 4

Objective

To understand and imitate an actual real-time workload in redis, it is necessary that we generate a large number of keys perform insert and update operations. Redis indeed has a very efficient caching mechanism, but it needs to work on a large number of keys as well.

Design of the experiment

To insert a large number of keys into the redis database, we considered a large text file which could be tokenized. The content was read from the file and simple tokenization based on spaces was performed to yield tokens, which could be inserted into the redis database. Before inserting a token into the database, we check to see if it already exists. If it already exists, then we just increment the value corresponding to the token by one. Essentially redis is expected to behave like a token count store for the tokens read from the database.

To insert a token with value 1 into the database, the following command was used

Performance Impact of NUMA architecture on Redis

SET <token_name> <token_value>

To check if a token already exists in the database, the following command was used

EXISTS <token_name>

This command returns 1 or 0 depending on whether the token is present in the database or not.

The command to increment the value of the token in case it is present in the database, i.e if EXISTS returns 1 is

INCR <token_name>

Using the above commands, we were able to achieve the token count functionality. We considered a large text file to be tokenized and analysed the execution times in case of local and remote memory allocation of the keys.

Pseudocode

main:

//Establish connection to redis server

// This is done using the hiredis API

set timeout for redis connection to be 1.5 seconds

hostname <- 127.0.0.1

port <- 6379

context <- redisConnectWithTimeout(hostname, port, timeout)

Performance Impact of NUMA architecture on Redis

```
// We can now redis commands use the redisCommand API

/*
   We now read through a text file, tokenize it and insert it into redis
   Redis will now behave as a word count store.
*/

filename <- command line input
fptr <- create file pointer to filename
counter_insert <- 0
counter_update <- 0
while fptr is not eof do:
    token <- read next token from the file
    // check if it already exists in the database using the EXISTS command
    command <- "EXISTS " + token
    reply <- redisCommand(command)
    if (reply == 1) do
        // We increment the counter of the token using the INCR command
        command <- "INCR " + token
        redisCommand(context, command)
        counter_update <- counter_update + 1

    else if (reply == 0) do
        // We insert this new token into the redis data store using the SET
command
        command <- "SET " + token + "1"
        redisCommand(context, command)
        counter_insert <- counter_insert + 1

print "Number of keys inserted : ", counter_insert
```



```
print "Number of keyes updated : ", counter_update
```

Observations

With local memory allocation

Inserted 81397 keys...

Updated 1014298 keys...

Time of execution

Real : 49.95 seconds

User : 6.764 seconds

Sys : 23.76 seconds

With remote memory allocation

Inserted 81397 keys...

Updated 1014298 keys...

Time of execution

Real : 49.58

User : 6.328

Sys : 24.02

Results

Keeping the number of insert and update operations to be the same, as we are tokenizing the same text file in both the cases, we see that the time of execution in both the cases is almost identical.

Inference

We have eliminated the possibility of caching by introducing a large number of distinct keys i.e 81397 to be precise. However, we are reading from a file to insert these keys into the database. We know that any file operation slows down a process and hence clearly file input is acting as a

bottleneck in this case. Since a large number of cycles are probably being spent waiting for file input, we do not observe a significant difference in the actual memory accesses for the keys. The file input is necessary as it is the source of the keys being inserted. Therefore, we need to either isolate the file input from the insertion and updating operations or find another source for the keys to be inserted into the database.

7.3.5 Experiment 5

Objective

As learnt from the previous experiment, we need to find a way to generate random keys that can be inserted into the database. To achieve this, we need to generate our own random strings and insert them into the database. We can then access the keys inserted and update their values with random strings again.

Design of the experiment

The key of the experiment is to generate a random string. This can be done using the following pseudocode. Using the above code to generate random keys, we now insert these keys into the Redis database with the value set as 1. We then iterate through all the keys in the database, and perform an update operation on each one of them by incrementing the value by 1.

We then vary the number of keys generated, and analyse the performance.

Pseudocode

generate_rand_string():

seed <- process_id + num_calls

// num_calls is the number of times this function has been called so far

Performance Impact of NUMA architecture on Redis

```
set random number generator seed
string_len <- generate random number in the range of 1-20
rand_string <- ""
i <- 0
while i < string_len:
    do
        rand_ascii_code <- generate random number in the range of 65 to
122
        char_rand <- convert rand_ascii_code to character
        append char_rand to rand_string
    append '\0' to the rand_string
return rand_string

main:

//Establish connection to redis server
// This is done using the hiredis API

set timeout for redis connection to be 1.5 seconds
hostname<- 127.0.0.1
port <-6379
context <- redisConnectWithTimeout(hostname, port, timeout)

//We can now redis commands use the redisCommand API

n <- command line argument input
// n indicates the number of keys to be inserted into Redis
/*
We can now generate 'n' random strings and insert them as keys into Redis using
SET
```

The value corresponding to each of these keys is set to an arbitrary value, say 1 in this case.

```
*/
```

```
i <- 0
```

```
while i < n do
```

```
  rand_string <- generate_rand_string()
```

```
  command <- "SET " + rand_string + 1
```

```
  redisCommand(context, command)
```

```
  i <- i + 1
```

```
/*
```

*We can now iterate through all the keys using the command keys **

The value of each of the keys is then incremented by an arbitrary number again,

say 1

This can be done by using the INCR command again

```
*/
```

```
command <- "keys *"
```

```
reply <- redisCommand(context, command)
```

```
i <- 0
```

```
while i < n do
```

```
  token <- reply->elements[i]
```

```
  command <- "INCR " + token
```

```
  redisCommand(context, command)
```

```
  i <- i + 1
```

Observations

Sl.No	Number of keys inserted	Local memory allocation execution time (in seconds)	Remote memory allocation execution time (in seconds)
1	10^3	0.048	0.047
2	10^4	0.43	0.48
3	10^5	4.6	4.89
4	10^6	45.22	45.91
5	10^7	443.28	459.98

Table 7.4 Wall clock time observations for experiment 5

Results

As we can see from the table, there is not much difference in the time of execution for small number of keys inserted. However, when the number of keys inserted is in the order of 10^7 , there is a difference of almost 17 seconds, which can be significant in critical workloads.

Inference

As the number of keys increases, it is likely that the key being updated is not present in the cache. This results in a remote memory access which is visible in the execution time. However, for smaller sized number of keys, this is not so significant as most of the keys are probably already present in the cache.

7.3.6 Experiment 6

Objective

To understand the impact of local and remote memory allocation with respect to another redis data structure - sorted sets. We have already seen the impact of NUMA architecture on several of redis data structures such as strings and lists.

Design of the experiment

Redis sorted sets are simply collections of distinct values, just like any other traditional set data structure. However, every element in the sorted set is associated with a score. The elements in the set are then sorted in ascending order of the score. Sorted sets also provide an operation to update a score of an existing or non existing element. The sorted set is then updated to reflect the change in the score of the updated element.

Internally, redis sorted sets are implemented using skip lists. They make use of two data structures to hold the same elements. These two data structures are then used to obtain $O(\log N)$ insert and remove operations into the sorted data structure. Simultaneously, the elements are added to a skip list mapping scores to Redis objects.

In order to study the impact of local memory allocation and remote memory allocation in case of sorted sets, we perform an experiment similar to that of the redis lists.

We create three sorted sets and then insert random strings into the sorted with a random score in the range of zero to hundred. We then iterate through the string inserted in each of the lists and increment the score of the strings by a randomly generated value again. This should ideally result in a number of internal update operations to reflect the new position of the updated element in the sorted set.

To create a sorted set and insert elements into it, we use the following command

```
ZADD myzset1 <score> <value>
```

To iterate through the elements of a sorted set, we can use the range command similar to the LRANGE command

```
ZRANGE myzset1 0 -1
```

For debugging purposes, we can use the ZRANGE command with the WITHSCORES options to view the scores of the elements inserted as well. Of course, in this case, it will simply the randomly generated initial scores with which they were inserted in the first place.

To update the score of an existing element in the sorted set, we use the ZINCRBY command as follows

```
ZINCRBY myset1 <new_score> <element>
```

The size of the sorted set is then varied and the pcm numa tool and time of execution is used to study the performance of redis sorted sets on NUMA architecture.

Pseudocode

generate_rand_string():

```
seed <- process_id + num_calls  
// num_calls is the number of times this function has been called so far  
set random number generator seed  
string_len <- generate random number in the range of 1-20  
rand_string <- ""
```

Performance Impact of NUMA architecture on Redis

```
i <- 0
while i < string_len:
    do
        rand_ascii_code <- generate random number in the range of 65 to
122
        char_rand <- convert rand_ascii_code to character
        append char_rand to rand_string
    append '\0' to the rand_string
    return rand_string
```

generate_rand_score():

```
seed <- process_id + num_calls
// num_calls is the number of times this function has been called so far
set random number generator seed
return random number generated % 100
```

main:

```
//Establish connection to redis server
// This is done using the hiredis API

set timeout for redis connection to be 1.5 seconds
hostname <- 127.0.0.1
port <- 6379
context <- redisConnectWithTimeout(hostname, port, timeout)

//We can now redis commands use the redisCommand API

/*
n <- command line argument input
```


Performance Impact of NUMA architecture on Redis

```
// n indicates the number of number of elements to be inserted into the sorted
set

/*
/*
We have to create three sorted sets and populate them
Let's name the lists as myset1, myset2, myset3
Initially, we have to delete these set if they already exist
*/

redisCommand(context, "DEL myset1")
redisCommand(context, "DEL myset2")
redisCommand(context, "DEL myset3")

/*
Now that we have deleted the lists, we can create the three lists, myset1, myset2,
myset3 and insert elements into them using the ZADD command
*/

i <- 0
while i < n do
  command <- "myset1 " + generate_rand_score + generate_rand_string()
  redisCommand(context, command)
  command <- "myset2 " + generate_rand_score + generate_rand_string()
  redisCommand(context, command)
  command <- "myset3 " + generate_rand_score + generate_rand_string()
  redisCommand(context, command)
  i <- i + 1
/*
Now we can access the elements of the sorted set using the ZRANGE command and
increment the score of the element using the ZINCRBY command
```

**/*

```
reply <- redisCommand("ZRANGE myset1 0 -1")
i <- 0
while i < reply->elements->size() do
    command < "ZINCRBY myset1" + generate_rand_score() +
reply->element[i]
    redisCommand(context, command)
    i <- i + 1

reply <- redisCommand("LRANGE mylist2 0 -1")

i <- 0
while i < reply->elements->size() do
    command < "ZINCRBY myset2" + generate_rand_score() +
reply->element[i]
    redisCommand(context, command)
    i <- i + 1

reply <- redisCommand("LRANGE mylist3 0 -1")
i <- 0
while i < reply->elements->size() do
    command < "ZINCRBY myset3" + generate_rand_score() +
reply->element[i]
    redisCommand(context, command)
    i <- i + 1
```

Observations

Sl.No	Size of the sorted set	Local memory allocation execution time (in seconds)	Remote memory allocation execution time (in seconds)
1	10^5	17.78	16
2	10^6	182	201
3	2×10^6	368	697

Table 7.5 Wall clock time observations for experiment 6

Results

As we can see, there is not much difference with the number of elements being in the order of 10^5 . However, with increase in number of elements in the sorted sets to the order of 10^6 , we see a significant difference between the execution times for local and remote memory allocation.

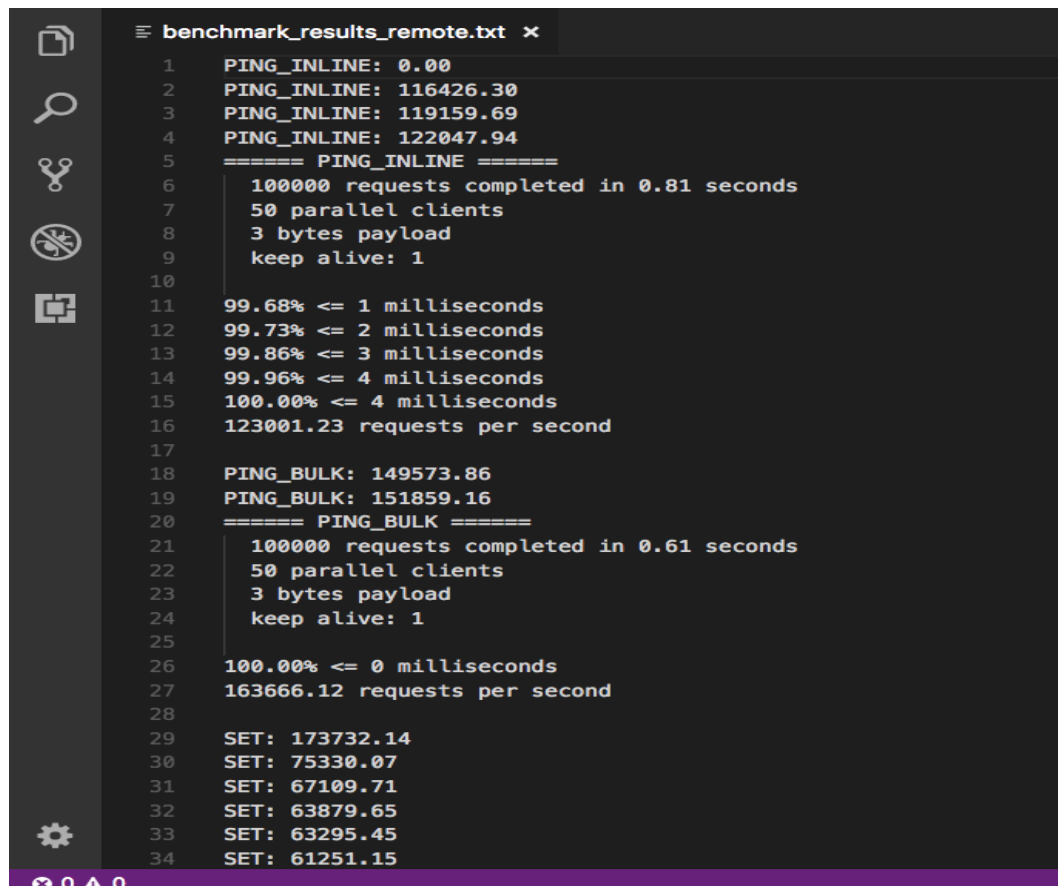
Inference

As the sorted set requires the elements to be sorted based on the scores at all instances, after every update operation on the score of the elements of the sorted set, there is an internal refactoring that is performed to reflect the updated score of the element. This proves to be expensive if the elements are stored in the remote memory, which is why we are seeing an appreciable difference in the execution times.

7.3.7 Experiment 7

Running Redis Benchmark

Redis benchmark is a standard benchmark tool available with redis that does a series of operations and times and analysis of those and therefore this would be a good tool to run with the redis server running with numactl local and remote memory access controls. We can then analyse the running of this benchmark tool with the output it generates and also by using the pcm-numa tool to understand what is happening internally as well with respect to the CPI's, number of local DRAM accesses and number of remote DRAM accesses. The benchmark program uses all the data structures to analyse each one of them, it starts off with ping inline, ping bulk and then continues with operations like SET, GET, INCR, LPUSH, RPUSH, LPOP, RPOP, SADD, HSET, SPOP, LRANGE and MSET.



```
benchmark_results_remote.txt x
1  PING_INLINE: 0.00
2  PING_INLINE: 116426.30
3  PING_INLINE: 119159.69
4  PING_INLINE: 122047.94
5  ===== PING_INLINE =====
6      100000 requests completed in 0.81 seconds
7      50 parallel clients
8      3 bytes payload
9      keep alive: 1
10
11 99.68% <= 1 milliseconds
12 99.73% <= 2 milliseconds
13 99.86% <= 3 milliseconds
14 99.96% <= 4 milliseconds
15 100.00% <= 4 milliseconds
16 123001.23 requests per second
17
18 PING_BULK: 149573.86
19 PING_BULK: 151859.16
20 ===== PING_BULK =====
21 100000 requests completed in 0.61 seconds
22 50 parallel clients
23 3 bytes payload
24 keep alive: 1
25
26 100.00% <= 0 milliseconds
27 163666.12 requests per second
28
29 SET: 173732.14
30 SET: 75330.07
31 SET: 67109.71
32 SET: 63879.65
33 SET: 63295.45
34 SET: 61251.15
```

Figure 7.6 : Benchmark results snapshot

The outputs of the pcm-numa has been captured and graphs for the same have been plotted and analysed.

CHAPTER 08**RESULTS AND DISCUSSIONS**

The number of local memory accessed was monitored using the pcm numa tool as mentioned above. The graphs depicted are evidence to the fact that the memory allocation was indeed in the local memory of the node executing the process. In case of forced remote memory allocation, the number of local memory accesses is very low indeed indicating that most of the process memory was in the remote memory of the node in which the process was executing.

Number of local memory accesses in local and remote memory allocation

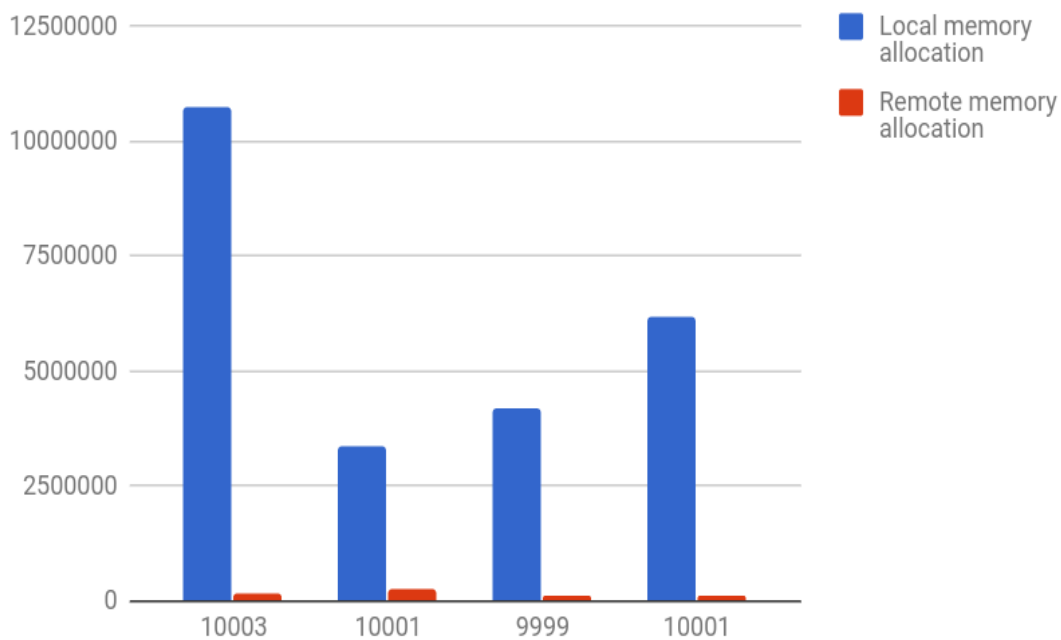


Figure 8.1 Number of local accesses performed by experiment 5 with number of keys inserted as 10^6

Number of local memory accesses in local and remote memory allocation

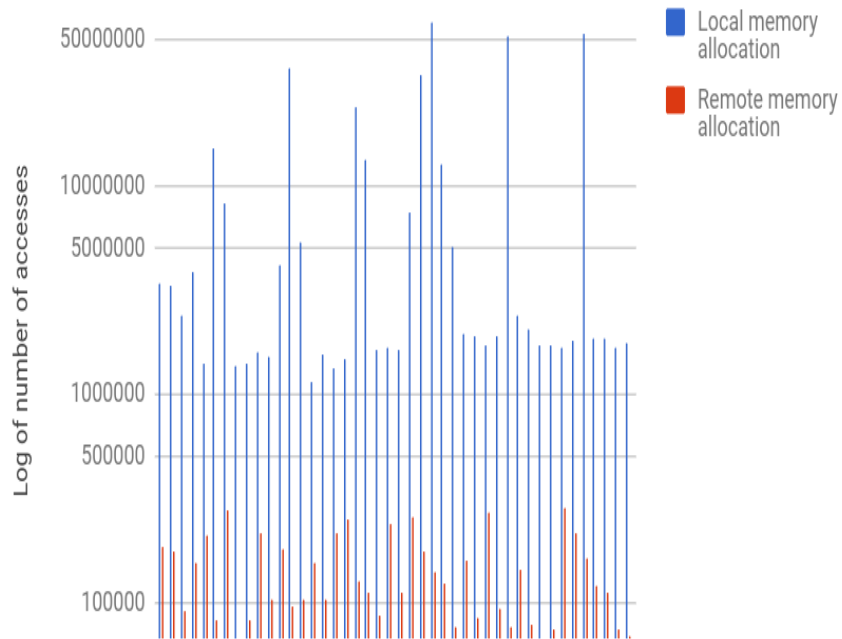


Figure 8.2 Number of local accesses performed by experiment 5 with number of keys inserted as 10^7

Likewise the number of remote memory accesses was also monitored using the pcm numa tool to verify that the forced remote memory allocation was indeed taking place. The significant decrease in the number of remote memory accesses in case of local memory allocation indicates that the node on which the process is executing hardly accesses the remote memory of the other node.

Number of remote memory accesses for local and remote memory allocation

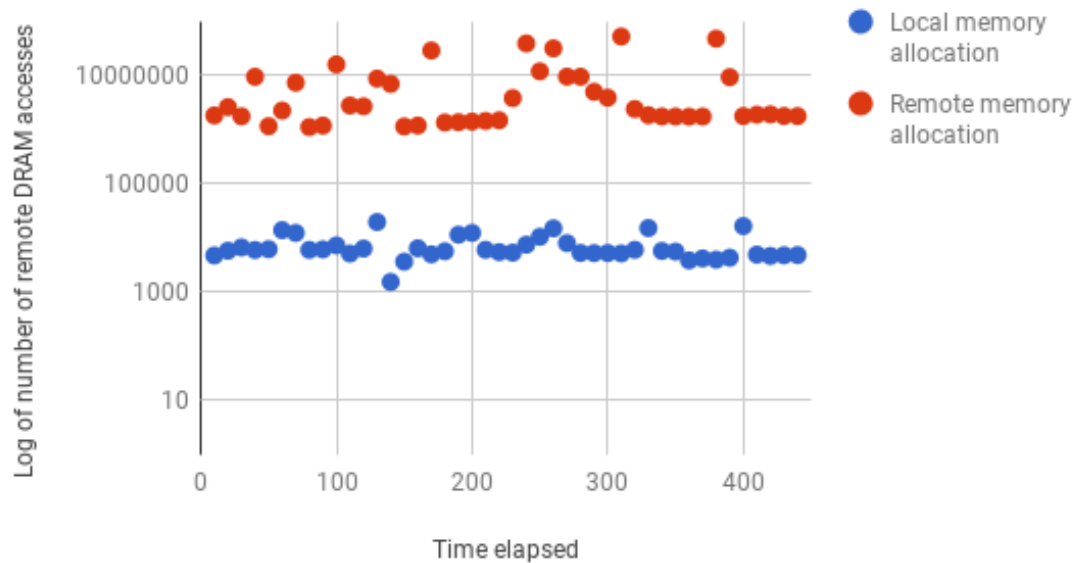


Figure 8.3 Number of remote accesses performed by experiment 5 with number of keys inserted as 10^7

Number of remote memory accesses for local and remote memory allocation

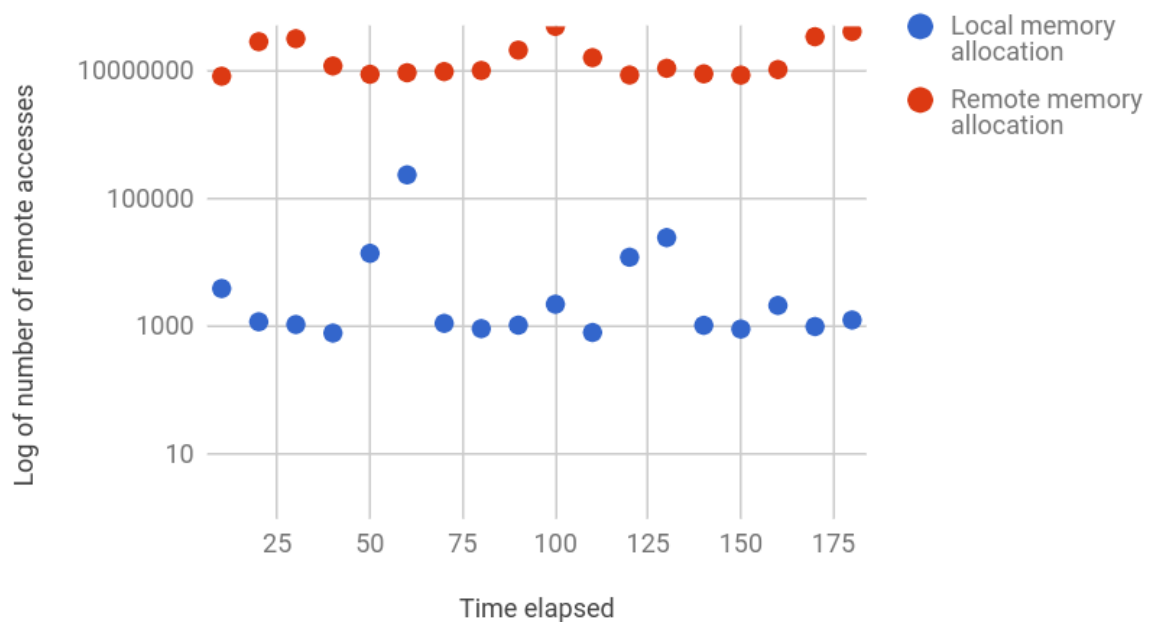


Figure 8.4 Number of remote accesses performed by experiment 6 with set size of 10^6

CPI vs Time elapsed

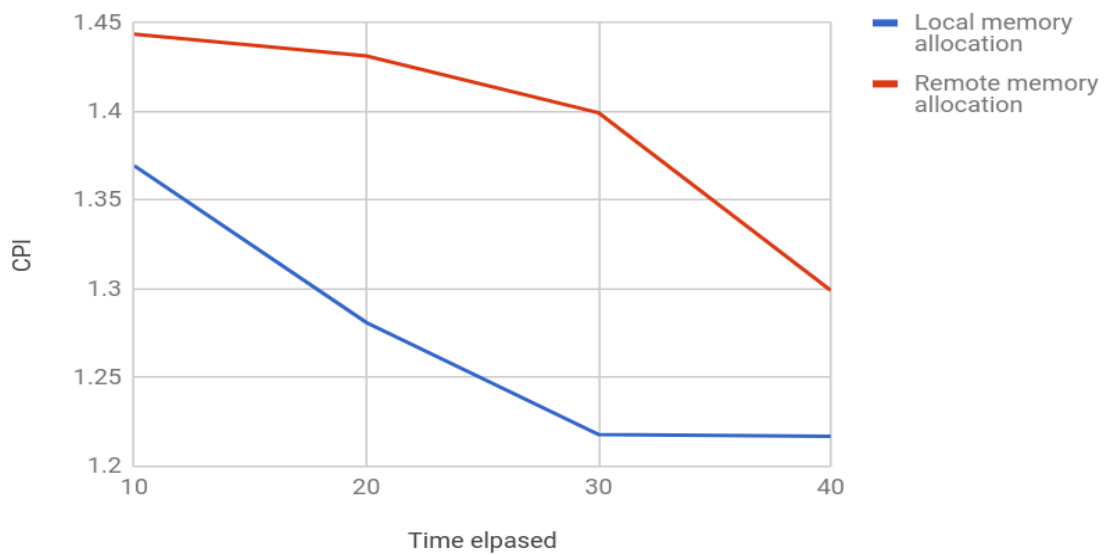


Figure 8.5 Cycles per instruction vs time performed by experiment 5 with number of keys inserted as 10^6

CPI vs. Time elapsed

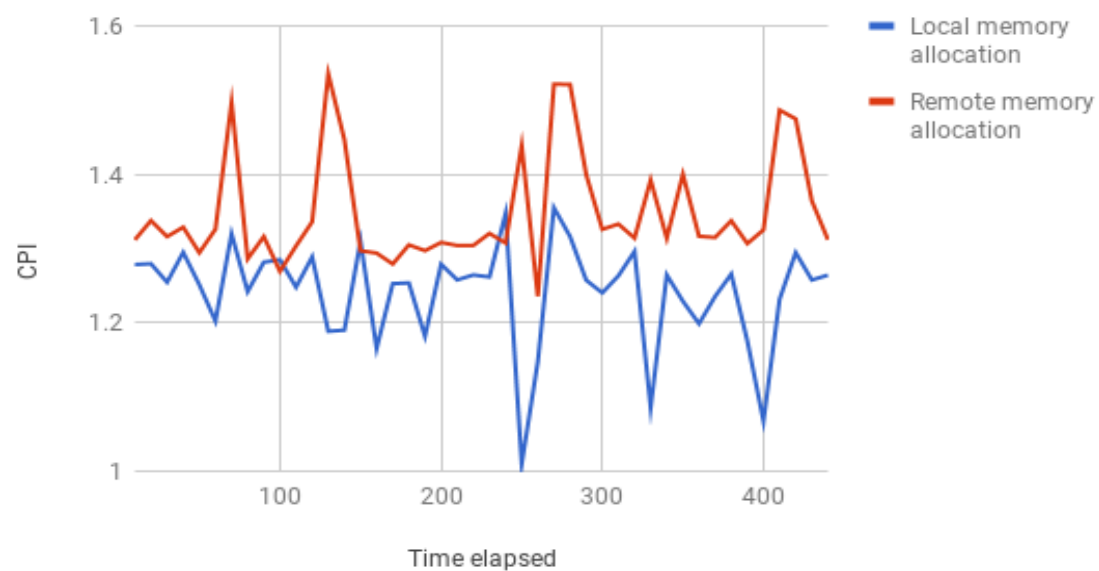


Figure 8.6 Cycles per instruction vs time performed by experiment 5 with number of keys inserted as 10^7

The number of cycles taken to execute an instruction is a good measure of the overall performance of the system. This information was again derived from the pcm numa tool. As we know, that a lower value of cycles per instruction (CPI), is an indicator of good performance of the system. From the graphs above, it is evident that local memory allocation is performing better with respect to CPI. From the experiments above, though there was not much difference in the execution times, CPI clearly indicates that the CPU is better utilised in case of local memory allocation.

IPC vs time elapsed

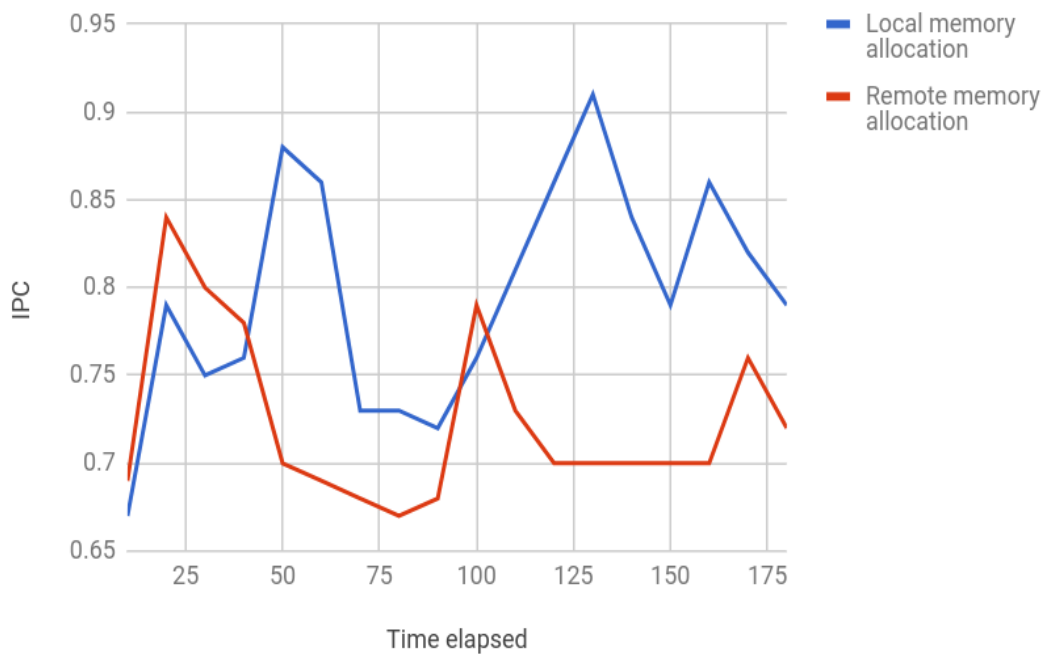


Figure 8.7 - IPC vs Time elapsed for experiment 7 on sorted set size of 10^6

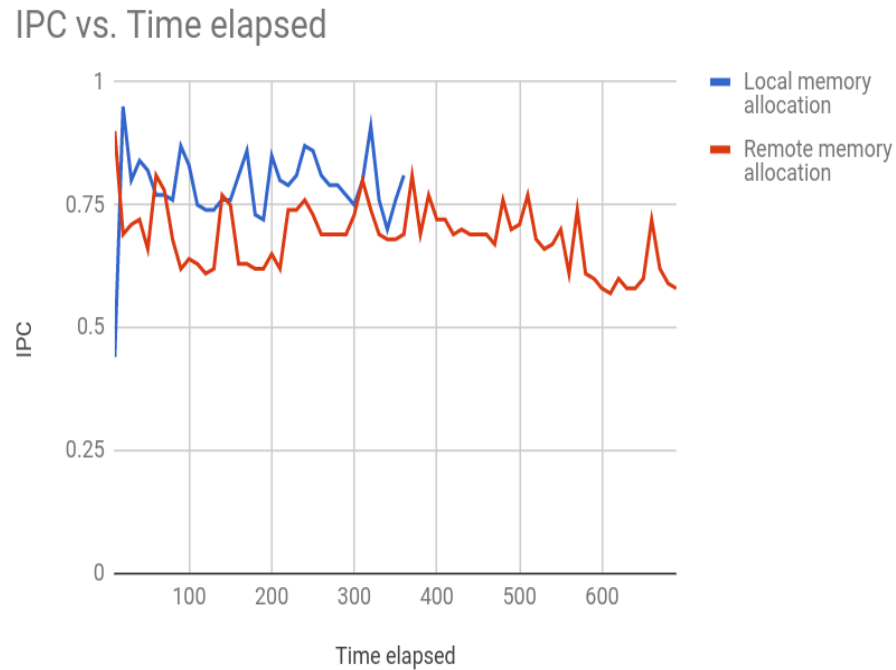


Figure 8.7 IPC vs Time elapsed for experiment 7 on sorted set size of 10^6

Another good metric for performance is the instructions executed per cycle (IPC). We associate a higher IPC value as an indicator of good performance of the system. From the results that are visualised above, it can be deduced that local memory allocation generally shows a higher value for IPC.

Overall we observe that though there is not much observable difference in the execution times between local and remote memory allocations, metrics such as CPI and IPC clearly indicate that efficient CPU utilisation is more likely in case of local memory allocation. In other words, there is an appreciable difference in performance when the memory accesses are forced to be remote.

Redis Benchmark

For the two output files generated by the benchmark program where the redis-server was tied to local and remote accesses, picking one command(LRANGE) that does quite a few reads we see the difference as follows

```
benchmark_results_local.txt x
362 LRange_600 (first 600 elements): 13614.79
363 LRange_600 (first 600 elements): 13392.16
364 LRange_600 (first 600 elements): 13443.35
365 LRange_600 (first 600 elements): 13358.10
366 LRange_600 (first 600 elements): 13542.57
367 LRange_600 (first 600 elements): 13459.80
368 LRange_600 (first 600 elements): 13470.00
369 LRange_600 (first 600 elements): 13379.63
370 LRange_600 (first 600 elements): 13405.72
371 LRange_600 (first 600 elements): 13379.20
372 LRange_600 (first 600 elements): 13474.17
373 LRange_600 (first 600 elements): 13469.89
374 LRange_600 (first 600 elements): 13554.23
375 LRange_600 (first 600 elements): 13537.24
376 LRange_600 (first 600 elements): 13602.04
377 LRange_600 (first 600 elements): 13573.13
378 LRange_600 (first 600 elements): 13574.84
379 LRange_600 (first 600 elements): 13509.53
380 LRange_600 (first 600 elements): 13492.04
381 LRange_600 (first 600 elements): 13441.41
382 LRange_600 (first 600 elements): 13426.50
383 LRange_600 (first 600 elements): 13383.44
384 LRange_600 (first 600 elements): 13392.26
385 LRange_600 (first 600 elements): 13381.10
386 LRange_600 (first 600 elements): 13424.67
387 LRange_600 (first 600 elements): 13421.93
388 LRange_600 (first 600 elements): 13461.08
389 LRange_600 (first 600 elements): 13444.08
390 ===== LRange_600 (first 600 elements) =====
391 100000 requests completed in 7.44 seconds
392 50 parallel clients
393 3 bytes payload
394 keep alive: 1
```

Figure 8.9 Comparison of benchmark results

CHAPTER 9

CONCLUSION

Running all the performance analysis for many programs including redis client programs that perform CPU intensive and memory intensive programs we were able to time the programs initially by just running the programs bound to local and remote memory accesses but the variations of time were not significant at lower values of array or the data structure used and makes a little but not extremely significant difference when the array sizes are large enough and therefore while using other performance tools like pcm-numa we were able to analyze more deeply on how the program runs and the output of pcm-numa writes the values like cycles,instructions,number of local DRAM accesses and number of remote DRAM accesses and therefore calculating cycles per instructions and plotting the graphs for the same then we understand that most of the times the CPI is less for the cases when the redis-server was bound to use local memory and as expected the number of remote accesses compared to local accesses are much much higher. Running various data structures and running for various sizes of data structure we were able to use pcm-numa with all these and plot graphs for the same and also use redis benchmark programs to run and analyze the same. So running the redis-server with numactl commands that tie the redis-server to use local memory we can get better CPI and also a slight time improvement but it is assumed that it would be significant when redis is actually used in production with real time data like streaming is fed into the database and it has to access memory in real time with lots of data to handle.

CHAPTER 10

FUTURE WORK

The future work would involve adding the changes into the source code of application like redis in order to take care of the numactl commands that are run with the options of binding it to local and remote work explicitly and automatically take care of this in the program to understand the core on which the program is running and then make sure to allocate the memory from the same node so that the advantage of faster local memory access can be used.

Also we would like to put this abstraction into the operating system level and add this functionality into the scheduler of the operating system so that this advantage can be taken in the scheduler itself and it can take care of making sure that the scheduler understands where the memory that the process uses resides and schedule the process in the same core so that it can access local memory. This would also involve better understanding of how the redis allocate memory initially when trying to make these changes at the application level which is redis and then trying to understand the linux scheduler scheduling and make changes to that in order to understand which memory the process would use and allocate the process on that node or the core and then recompile it.

CHAPTER 11

REFERENCES

[1] Local and Remote Memory: Memory in a Linux/NUMA System - Christoph Lameter

[2] Optimizing Google's Warehouse Scale Computers: The NUMA Experience - Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune.

[3] Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead -Zoltan Majo, Thomas R Gross.

[4] Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, Pierre-André Wacrenier.

[5] NUMA-Aware Java Heaps for Server Applications - Mustafa M. Tikir Jeffrey K. Hollingsworth.

[6] Thread and Memory Placement on NUMA Systems: Asymmetry Matters - Baptiste Lepers, Vivien Quema, Alexandra Fedorova.