2008

Efficient data access techniques for large structured data files

Payal Patel

Master's Project Final Report

Efficient Data Access Techniques for Large Structured Data Files

Payal A Patel

pap1851@cs.rit.edu

Rochester Institute of Technology

Chairman: Hans Peter Bischof ___________________________

Reader : Rajendra K. Raj ___________________________

Observer : Reynold Bailey ___________________________

Table of Contents

Chapter 1

Introduction

GRAPEcluster is a project carried out at RIT through the collective efforts of students, post-doctorate workers and faculty from the Astrophysics and Computer Science departments. This group mainly concentrates on the formation and evolution of galactic nuclei and super massive black holes by developing new algorithms, creating visualization software and performing other scientific research. An extremely powerful and extensible visualization program called Spiegel has been developed by this group to visualize the data in 3-D thereby facilitating the exploration and study of the data through time and space.

Data used for the visualization program is provided by the Astrophysics department. This raw data has to be processed and visualized for understanding the intricate behavior of astrophysical strong gravity systems. A distinct characteristic of astrophysics data is its enormously large volume. The raw data file can typically be on the order of gigabytes or terabytes. Some type of High Performance File System (HPFS) or Specialized File System (SFS) is used for storing and managing such a large quantity of data.

A HPFS is a file system that is installed in the operating system (OS) to provide faster access to large hard drives. From a performance point of view a HPFS provides excellent throughput by using advanced data structures, intelligent caching, contiguous memory allocation, read-ahead and deferred write mechanisms. But for applications that require faster data access many factors apart from the disk throughput like network delays, I/O bottlenecks etc may affect the performance of file I/O operations.

The project focuses on programming solutions like multi-threaded reading, preprocessing of data, and using memory mapped files provided by the Java New I/O API to reduce I/O bottlenecks which is one of the major performance degrading factors for faster data access. The project also explores the possibility of utilizing databases as a solution to study and compare the performance of database querying over flat file reading.

Chapter 2

Project Statement

This section describes the problem with the Spiegel project and presents the goal of this project and how it intends to resolve the defined problem.

2.1 The Problem

The background for the problem on hand is accessing the large volume of raw data produced by the Astrophysics department efficiently in order to enable a faster visualization process in the Spiegel project. The use of better processing power, improved network bandwidth, HPFS etc could not address the underlying I/O problems associated with the file system which is one of the major performance limiting factors. For performance gains specifically file I/O problems have to be resolved. The project studies these issues and suggests solutions to minimize their effects.

2.2 The Project Goal

The goal of this project is to provide solutions that may reduce the time taken by I/O operations accessing data in the Spiegel project.

The following approaches are suggested to solve this purpose.

1. Databases
2. Single Thread Implementation
3. Multi-threading with Java I/O and Java API
4. Java New I/O
5. Data preprocessing

Section 4 describes these solutions and section 6 further describes their result and performance.

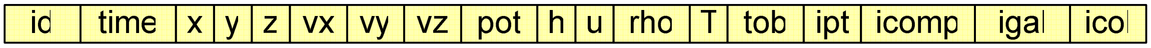
Chapter 3

Project Overview

This section gives an overview of the client-server architecture used for the Spiegel simulation, the format of the file generated by the simulation, its structure, size, field names etc. It also describes how the proposed solutions will fit in the existing Spiegel project.

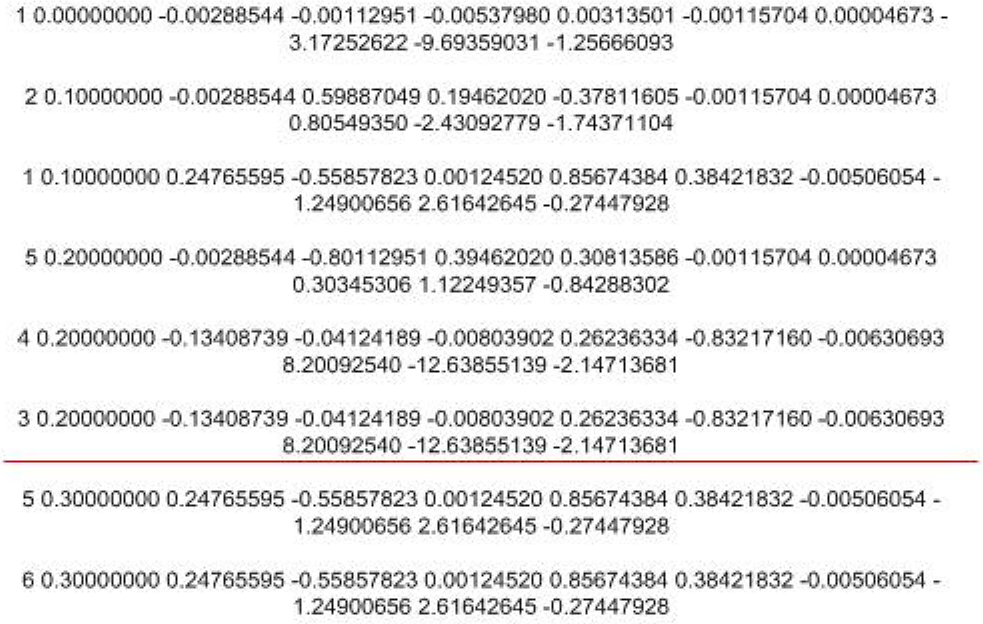
3.1 File Structure

Data generated by the Spiegel project simulation program is structured. It has a fixed format and length. The data contains information about particles in space such as the particles time of presence, mass, position, density, time of birth etc. Every particle is identified by a unique Id and a Time marker that indicates the time at which that particle was last observed. The diagram below gives a graphical view of the data format. Not all particles are equally important and needed for creating visualization at all times. This results in other issues discussed later.



id	time	x	y	z	vx	vy	vz	pot	h	u	rho	T	tob	ipt	icomp	igal	icol

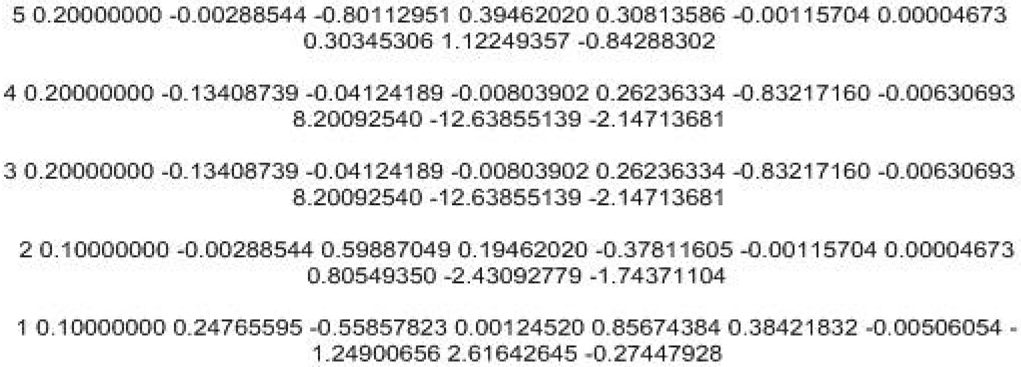
Data in the file is sorted in ascending order on the basis of time at which the particles were observed. The data is not uniformly distributed throughout the file i.e. not all particles are observed at all times. For the visualization process it is necessary to have data of all particles starting from time 0, where all particles are assumed to be present, till the time requested by the client. In the event a particle is not present at the specified time the missing particles have to be fetched from the time they were last observed. Figure below presents a snapshot of a file containing raw data.



```
1 0.00000000 -0.00288544 -0.00112951 -0.00537980 0.00313501 -0.00115704 0.00004673 -
3.17252622 -9.69359031 -1.25666093

2 0.10000000 -0.00288544 0.59887049 0.19462020 -0.37811605 -0.00115704 0.00004673
0.80549350 -2.43092779 -1.74371104

1 0.10000000 0.24765595 -0.55857823 0.00124520 0.85674384 0.38421832 -0.00506054 -
1.24900656 2.61642645 -0.27447928

5 0.20000000 -0.00288544 -0.80112951 0.39462020 0.30813586 -0.00115704 0.00004673
0.30345306 1.12249357 -0.84288302

4 0.20000000 -0.13408739 -0.04124189 -0.00803902 0.26236334 -0.83217160 -0.00630693
8.20092540 -12.63855139 -2.14713681

3 0.20000000 -0.13408739 -0.04124189 -0.00803902 0.26236334 -0.83217160 -0.00630693
8.20092540 -12.63855139 -2.14713681

5 0.30000000 0.24765595 -0.55857823 0.00124520 0.85674384 0.38421832 -0.00506054 -
1.24900656 2.61642645 -0.27447928

6 0.30000000 0.24765595 -0.55857823 0.00124520 0.85674384 0.38421832 -0.00506054 -
1.24900656 2.61642645 -0.27447928
```

Figure: Snapshot of raw data file.

From the file snapshot shown above, it is observed that at time 0.00 only particle with Id 1 was visible, at time 0.1 particle with Id 1 and Id 2 were visible, at time 0.2 particles with id 3, 4 and 5 and at time 0.3 particles with Id 5 and 6 only were visible. This is what is meant by not “uniformly distributed”. Certain gaps may exist in the file as clearly seen in the file snapshot above. These gaps indicate that either the particle with a particular Id was never seen till that time or it ceased to be visible after some time. For the visualization process, all particles that were visible till the requested time need to be fetched which mean that these particle gaps have to be filled. In order to do so the file is read from time 0 till the requested time. For example for time request 0.2 the particle fetched will be as shown below:



```
5 0.20000000 -0.00288544 -0.80112951 0.39462020 0.30813586 -0.00115704 0.00004673
0.30345306 1.12249357 -0.84288302

4 0.20000000 -0.13408739 -0.04124189 -0.00803902 0.26236334 -0.83217160 -0.00630693
8.20092540 -12.63855139 -2.14713681

3 0.20000000 -0.13408739 -0.04124189 -0.00803902 0.26236334 -0.83217160 -0.00630693
8.20092540 -12.63855139 -2.14713681

2 0.10000000 -0.00288544 0.59887049 0.19462020 -0.37811605 -0.00115704 0.00004673
0.80549350 -2.43092779 -1.74371104

1 0.10000000 0.24765595 -0.55857823 0.00124520 0.85674384 0.38421832 -0.00506054 -
1.24900656 2.61642645 -0.27447928
```

Figure: Resultant data for the request of data time 0.2

Particles with particle Id 1 and 2 were seen at time 0.1 but were not visible at time 0.2. So to serve the request for data at time 0.2, the file is read from time 0 till time 0.2. This way all particles seen till time 0.2 will be fetched from the time they were last visible.

3.2 Client-Server Architecture

Figure 1 depicts the client server architecture of the Spiegel project used and simulated in this project.

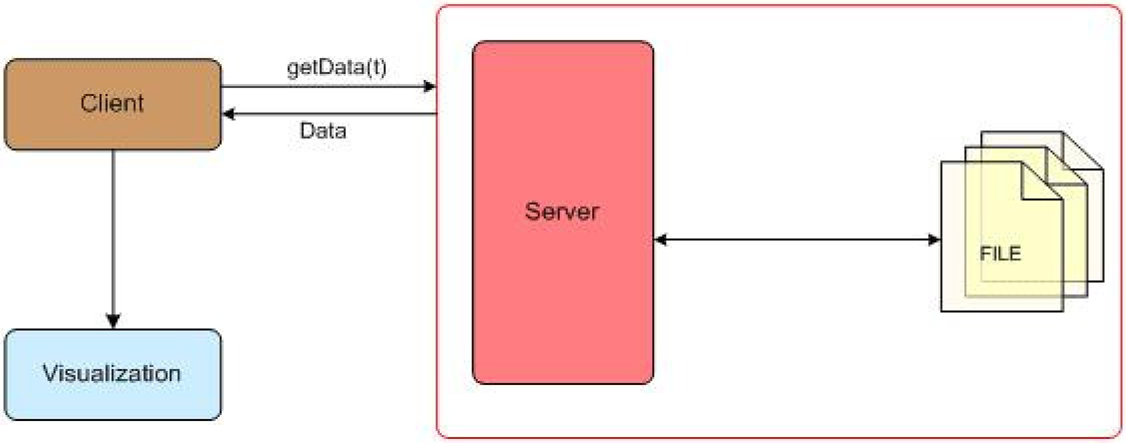


Figure 1: Client-Server Architecture

The client and the server can be located on the same or different machines. The visualization simulates behavior of particles over a specified range of time. Depending on the time frame, the client requests data from the server which in turn fetches data from files stored locally on the server or on some remote server and responds back with the requested information.

3.3 Integration of Solutions

The proposed solutions are applicable on the server side and will guide the server for file reading. Based on the implementation chosen sequential file reading, multi-threaded file reading, memory mapped file reading or database querying will be executed by the server to serve the client request.

Diagram 2 below shows how the solutions are integrated on the server side. The project provides an additional layer between the existing server and the files guiding the server through the file reading process.

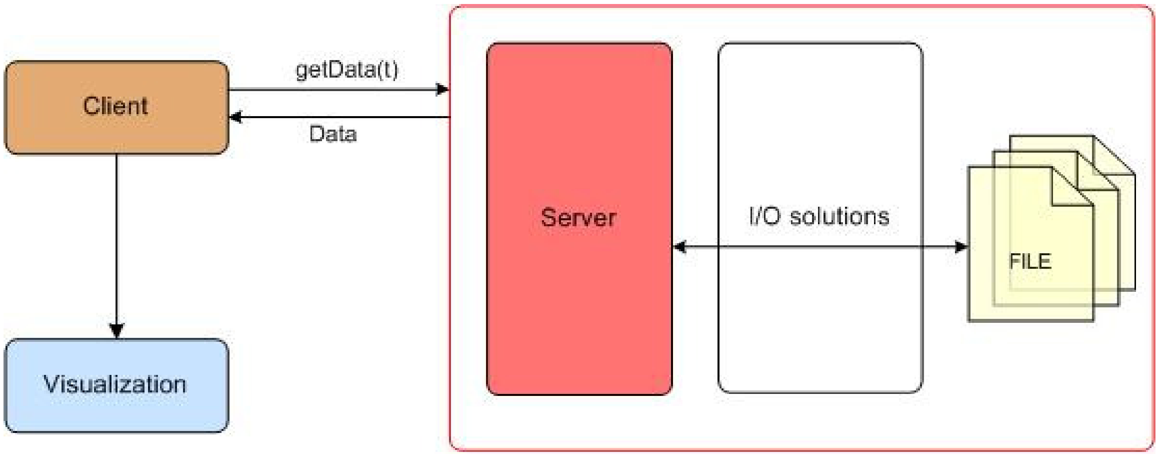


Figure 2: Integration of Solutions in the existing Spiegel project client-server framework

Chapter 4

Proposed Solutions

The main goal of the project is to improve I/O performance of the system used for the visualization process. The following ideas are proposed to improve the current performance.

1. Database Implementation
2. Single Thread Implementation
3. Java Multithread Implementation
4. Java new I/O
5. Preprocessed data Implementation

Figure 3 depicts the working of the client-server model in the Spiegel project after the integration of the proposed solutions. The server takes requests from the client and based on the implementation chosen by the client the file is accessed through the proper mechanism. Irrespective of the method used by the server to read the file, the data received from and sent to the client is always in the same format. An exception exists for the Preprocessing implementation (Section 4.5) where the client specifies fields of interest.

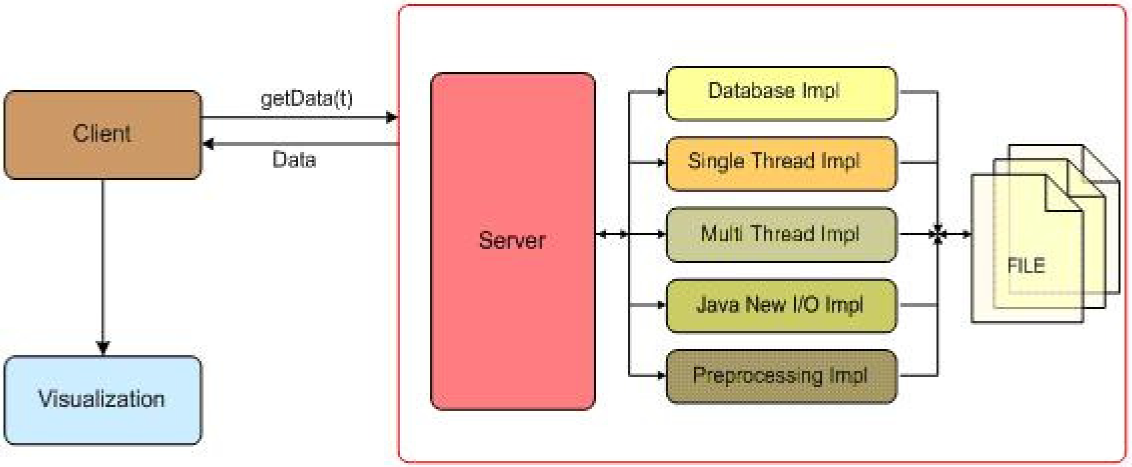


Figure 3: Client-Server Architecture with proposed solutions integrated.

Figure 4 presents the interface design used for implementing the proposed solutions in this project.

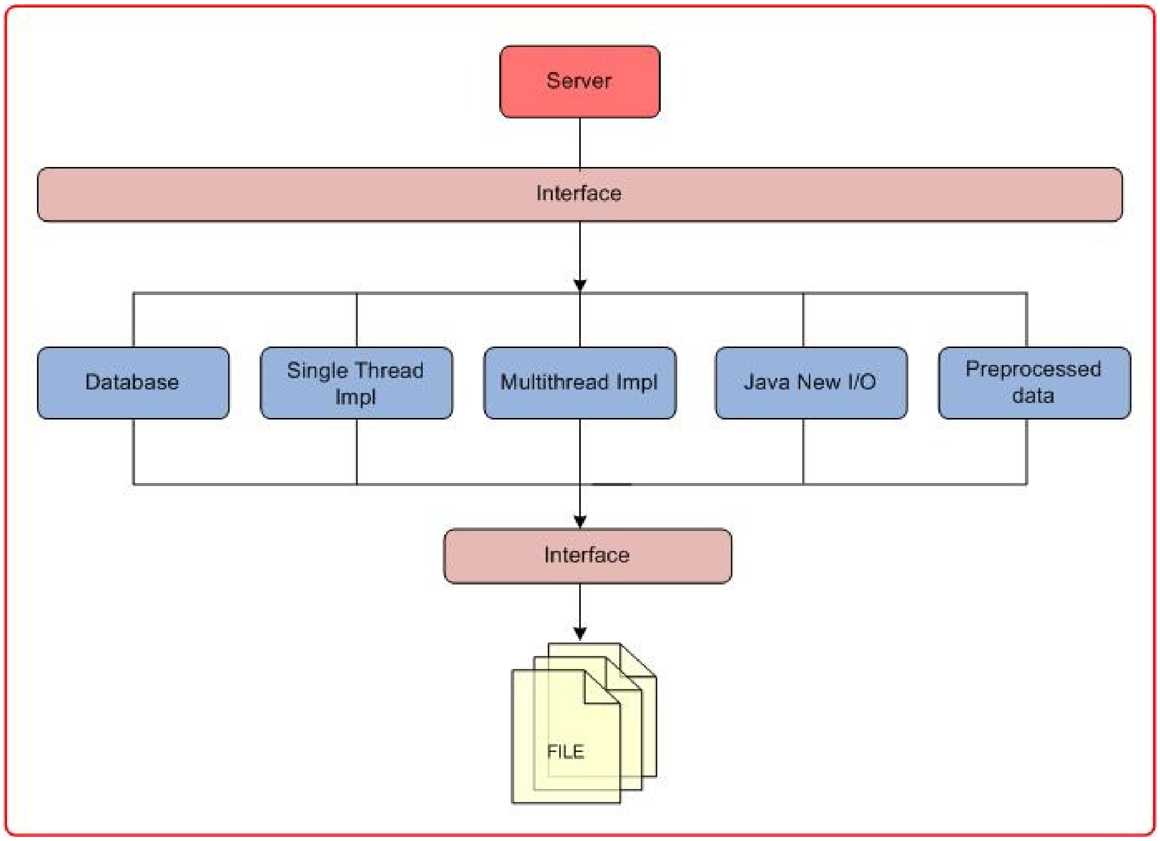


Figure 4: Interface diagram for the proposed solution.

The interface diagram in Figure 4 shows how this project can be expanded in the future by incorporating different solutions. Any class by implementing both the interfaces can communicate between server and file. If implementing the Server interface it can receive arguments from the server that the server gets from the client and by implementing the File interface it fetches data from the file in a particular fashion based on the received parameters and sends it back to the server. This way irrespective of knowing how other methods are implemented allows new solutions to be added to this framework without making major modifications to the architecture.

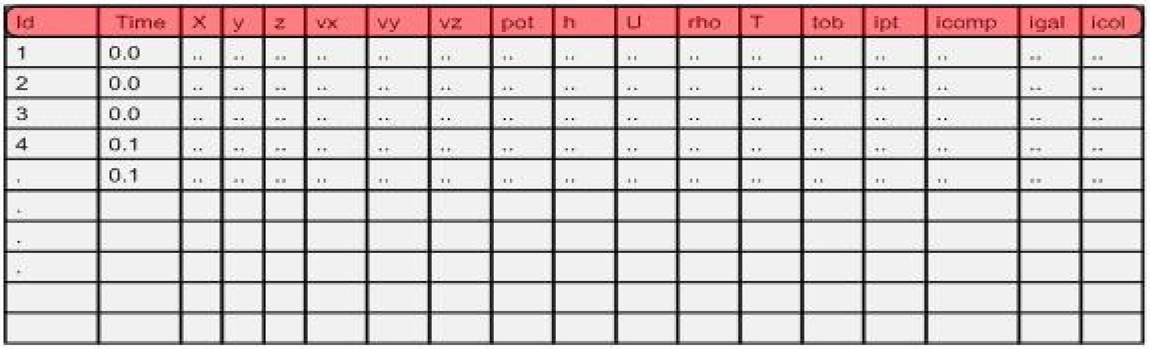
4.1 Database Implementation

It was thought that database querying would be more efficient than reading a flat file because database uses mechanisms like indexing, hashing, clustering etc to optimally store data so as to enable faster data access and minimize disk reads while querying. Moreover the physical layer of the databases can also be designed to reduce the number of disk reads during database querying.

This implementation involves extra processing in the form of creating tables and loading the then raw data into appropriate tables before a query is sent to the database. Firstly the raw data file is loaded into the database in a table having the same name as that of the filename with Time and Id fields together forming a composite key. The reason for using Time and Id as the composite key is because at any given time t a particle with a certain Id will only be seen once. Hence this combination is always going to be unique and thusly can serve as a key for the table. Once data is loaded in the database, a table called `TimeId` containing the Time and Id columns is created. The table associates time and all the particles present at that time in one single row. This table is created so that whenever a client sends a request for particles at a particular time t, the

TimeId table is first checked for missing particles at that time. The raw file, illustrated in figure 5, is stored in the database and figure 6 depicts the TimeId table.

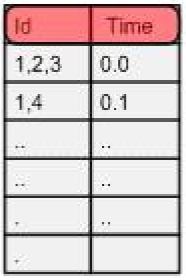
Table: Simulation



Id	Time	X	y	z	vx	vy	vz	pot	h	U	rho	T	tob	ipt	icomp	igal	icol
1	0.0	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
2	0.0	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
3	0.0	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
4	0.1	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
.	0.1	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..	..
.																	
.																	
.																	

Figure 5: Data stored in the database

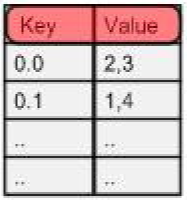
Table: TimeId



Id	Time
1,2,3	0.0
1,4	0.1
..	..
..	..
.	..
.	

Figure 6: Data in the TimeId table

When the client makes a request for data at a particular time t, the server first searches for the missing particles at that requested time t in the TimeId table by comparing the rows for every time t' less than the requested time t. The gaps in the data file are filled this way by table comparison. The information about the particles at time t and the information regarding the missing particles at time t are stored in a map. For example the request of data at time 0.1, the map would look like the one illustrated in the figure 7.



Key	Value
0.0	2,3
0.1	1,4
..	..
..	..

Figure 7: Map created when client requests for data

From the map the server knows that particles 2 and 3 are missing at the requested time 0.1 and hence will have to be fetched from the time they were last seen, in this case 0.0. The server uses this information stored in a map to form a composite key i.e. `<Time,Id>`. As mentioned above `<Time,Id>` together serve as a composite primary key into the database table. This combination will always be unique for every record in the table.

Based on this fact, the server uses the `<Time,Id>` information in the map as a key to query the database. The advantage of creating the `TimeId` table for compensating the gaps in the raw data file is that we reduce the number of queries on the database if the data is not skewed. For example the SQL query for the request to get data at time 0.1 is shown below:

```
Select * from Simulation where time = 0.1 and id = 1 OR id = 4
Select * from Simulation where time = 0.0 and id =2 OR id =3
```

In the above example we did not read data of particle 1 at time 0.0 from the database table. If all particles are mostly visible at all times, this approach can save excessive database querying time by avoiding unnecessary database reads.

In spite of its advantages, this implementation has its own limitations. Firstly the time taken to create a huge query is considerable if the number of particles at a given time is large. Secondly, the time taken for querying the database is minimal, but as the `Resultset` object created is not serializable we have to extract information from the `Resultset` on the server side and send the resultant string to the client. Unfortunately, iterating through the `Resultset` turns out to be an expensive operation and does not yield better performance as compared to reading a flat file.

4.2 Single Thread Implementation

This is the current implementation used for accessing data from files for the visualization process in the Spiegel project. In this approach the client sends a request for information about all particles present at time t. The server then reads from time 0 till time t into a map and sends it to the client side. When a subsequent request for the next time comes, the server updates the previous time map with new data and sends that to the client.
Using this approach; the need for reading the file from the beginning for every subsequent request is avoided.

The performance of this approach can be improved if data is pre-fetched so that when data for one time request is being read, another thread can start reading the next time block. This way for consecutive requests the client doesn't have to wait long. This idea is explained further in the multi-threaded implementation (section 4.3). Another modification to the single thread implementation to reduce the I/O bottleneck can be done by reducing the number of bytes read and by reading in pre fetching fashion. This idea is explored by implementing the preprocessed data implementation (section 4.5). The reduction of file I/O as a solution was examined by utilizing databases previously.

4.3 Java Multithread Implementation

The motivation behind this implementation was to achieve the advantage of data pre-fetching similar to the way done by the operating system for reading blocks of data from the disk. Through providing multiple blocks pre-fetching instead of reading just one block ahead this implementation allow the user to vary the number of threads and pre-fetch as many blocks as desired at a time

When the client connects to the server it provides information about the request including the source file containing data, the start, end and interval times for the subsequent request. On the server side, based on these time values a predefined number of threads are instantiated. Before individual `Thread` objects are created though, the server first checks if the position file for that source file exists. If this file does not exist, a file containing the positions is created. This file contains the seek position for the beginning of every new time block in the source file. This way each thread according to the time supplied to it knows from where it is supposed to start reading the source data file based on the seek position for that time from the position file. Figure 8 shown below describes the entire architecture of this approach in detail.

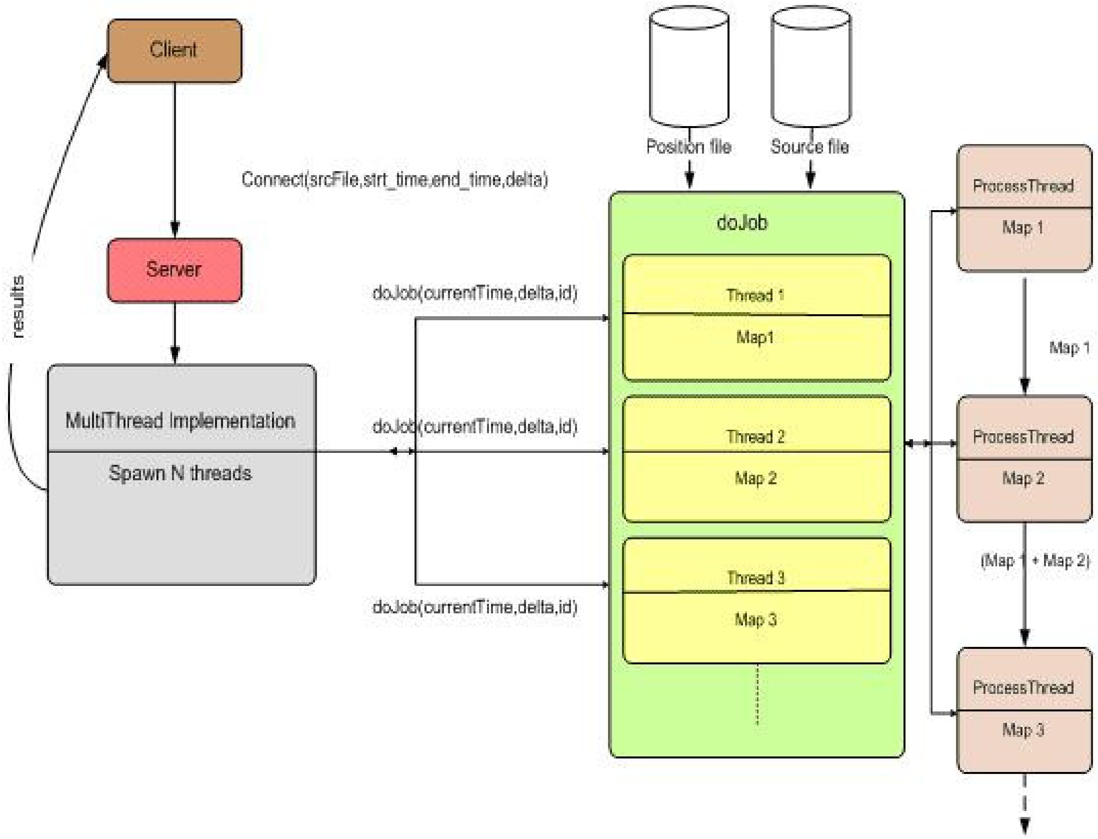


Figure 8: Java Multi-Thread implementation diagram

When the client sends the first request, N thread objects are spawned. N can be varied but can never be greater than the total number of time blocks requested by the client. This is calculated by subtracting the start time from end time and dividing it by the interval/delta.

```
Max # of Thread = ( end_time - start_time)/delta
```

This means that only one thread is responsible for reading a single block of time.

The following pseudo code explains how this implementation works.

Step 1: Client connects to the Server.

```
        Connect(source_file,start_time,end_time,delta)
```

Step 2: Server checks if the position file is created for that source file.

```
    If (not)
    Create position file
    Else
    Spawn N threads and start them.
    currTime = start_time+delta;
    Thread 0(currTime,delta, thread id:0).start()
    currTime+=delta;
    Thread 1(currTime,delta, thread id:1).start()
    currTime+=delta;
    Thread 2(currTime,delta, thread id:2).start()
              :
              :
    Thread N (currTime,delta, thread id:N).start()
```

Step 3: When the threads finish reading they set a `Boolean` array bit to true.

This is done to indicate that the thread has completed reading its assigned time.

```
    Boolean READ_ARR[N]
    READ_ARR[thread id:0] = true;
    READ_ARR[thread id:1] = true;
         :
    READ_ARR[thread id:N] = true;
```

Step 4: Set the `Boolean` process array bit to true for `Thread` id 0

```
    Boolean PROCESS_ARR [N]
    if (thread id:0)
    PROCESS_ARR [thread id:0] = true;
    ELSE
    if (READ_ARR[thread id: X] AND
                        READ_ARR[thread id:X-1]) = = true)
    thread (X-1).map = thread(X).map
    PROCESS_ARR[thread id:X] = true;
```

`Step 5: Getdata(t)` Get request from the client.

```
    if ( PROCESS_ARR[N] = true)
    return map.data();
    else
    Wait();
```

The most attractive aspect of this implementation is that it provides the flexibility to vary the number of threads. This way if system resources are limited the number of threads spawned can be reduced yet still benefit from some level of pre fetching or parallel reading. If there are no restrictions on the system resources, then the max number of threads can be spawned.

4.4 Java New I/O Implementation

This implementation explores the performance of using memory mapped files. Java New I/O provides a facility for using memory mapped files in file I/O operations. Figure 9 demonstrates memory-mapped I/O concept.

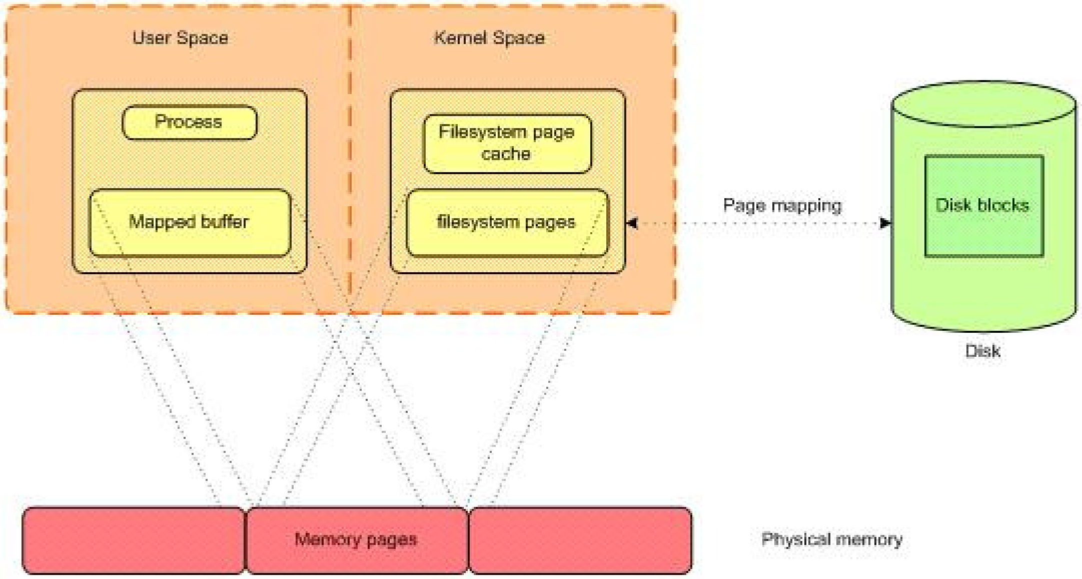


Figure 9: User-memory mapped to file system memory [1]

Memory-mapped I/O and the filesystem together create a virtual memory mapping between user space and the applicable filesystem pages. One of the benefits of this mapping is that very large files can be mapped without consuming large amounts of memory to copy the data and as such, the user process can view the file data as memory and expensive `read()`and `write()` system calls can be avoided. This makes file accessing via memory mapping mechanisms more efficient than the conventional time consuming `read()` and `write()` system calls. Moreover, the virtual memory system of the operating system automatically caches memory pages using the system memory and thus, no space is consumed from the JVM[1]'s memory heap.

[1] The FileChannel class of Java New I/O API creates a virtual memory mapping between an open file and MappedByteBuffer via the map() method. The MappedByteBuffer object resulting from the map() method behaves like a memory-based buffer with the data elements stored in a file on disk. The data visible through this type of mapping is exactly the same as that obtained by reading the file by conventional means, but with the added benefit of direct access.

This implementation is similar to the Single Thread implementation except that it uses the Java New I/O API for file operations. Instead of using random access file as in the single thread implementation, this implementation uses the MappedByteBuffer class.

In spite of its advantages, it carries an inherent problem due to the internal architecture of Java. This implementation has a tendency of throwing a "Not enough Memory" exception. If the same program is executed consecutively multiple times or if many programs are executed simultaneously this implementation may throw this memory exception. Although the exact cause is difficult to pinpoint based on a literature survey for this problem the cause is narrowed down to possibly being the inability of the garbage collector to free the ByteBuffers that are used internally for this mechanism. The way MappedByteBuffer is designed is such that even if the file channel is closed after completion of the task the mapping between memory and the file is not released.

According to the documentation given by Sun on the Java New I/O API this was done for some data security and integrity reasons. Therefore until the garbage collection is done, we cannot free the memory mappings. Moreover, these MappedByteBuffer

[1] Java Virtual Machine

do not have a `close()` method which means that unlike other Java I/O objects such as Streams etc it cannot release the resources it is utilizing.

Unfortunately even though this implementation outperforms other, it could give unexpected results based on the amount of free memory available to it.

4.5 Preprocessed Data Implementation

For the visualization process only certain attributes or fields are required from the enormous amount of data. For example only x,y,z coordinates along with Time and Id or the vx,vy,vz coordinates and Time and Id etc are needed for certain visualizations. The other data fields are just discarded. As per the problem analysis where file I/O was found to be one of the major bottlenecks, the motivation behind this approach is to reduce the number of bytes read so as to improve the data access rate.

As shown in the Figure 10, the raw data file is split and all fields are stored separately in different files. Based on the usage of specific fields like Time and Id which are a must for any visualization, these are grouped and put together in a single file. The x,y,z or the vx,vy,vz coordinates will always be used together and not individually so x,y,z attributes are put together in one file and vx,vy,vz attributes are stored in another single file.

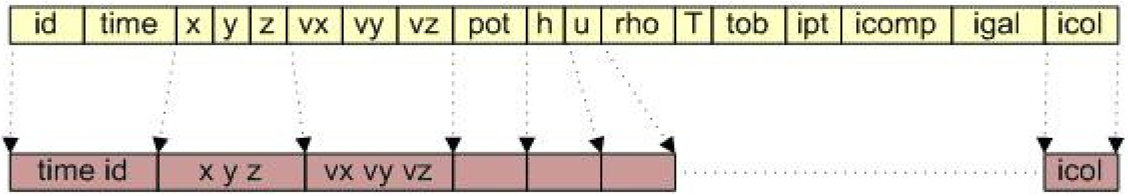


Figure 10: Preprocessing of raw data file.

When the client sends request for data, the server checks for a <*sourcefile*>_<*timeid*>`.txt` file. If this file exists, it means that the raw data file is preprocessed otherwise the server requests the client to send the names of the column in the source file in the order in which the data is stored in the source file.

Based on these column names, the server creates file names of the format <*sourcefile*>_<*columnName*>`.txt` for each attribute. The source file is read and splits the data in the order in which the client supplied the column information. The client must wait till the files are generated. Figure 10 illustrates this process.

The implementation of this approach is very similar to the multi-threaded implementation except that when the client connects to the server, it informs the server its columns of interest. The server while serving the request fetches data only for those columns from their respective files and creates a data string and sends that to the client. The working and design of this implementation is very similar to the multi-threaded implementation. This approach also gives the flexibility of using pre fetching or parallel reading like the multi-threaded approach.

The main goal of this implementation was to access fewer bytes improving I/O performance through less work rather than addressing actual I/O limitations. The performance detail of this implementation is discussed in detail in the Test and Result section.

Chapter 5

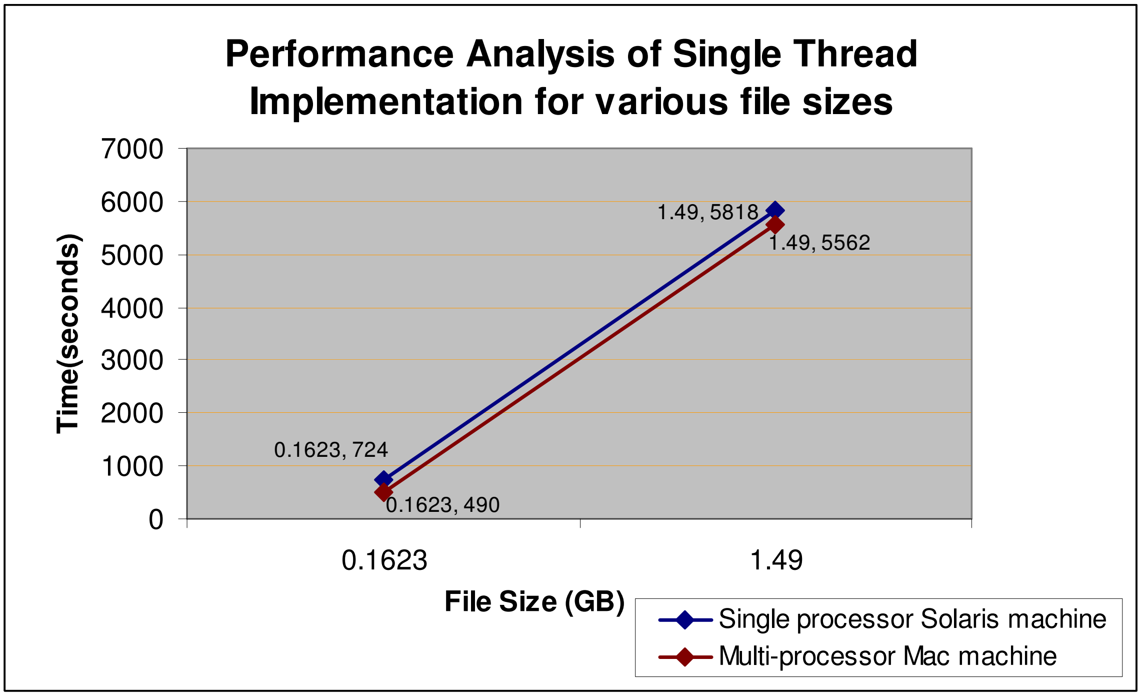
Testing and Performance Analysis

The goal of the project was to improve the I/O performance. The result of project implementation is supposed to improve the overall time taken to serve client requests in Spiegel project. Experiments were carried out for serving client requests by varying file sizes and time intervals using different methods implemented in the project.

Section 5.1 explains the performance of each implementation employed in this project.

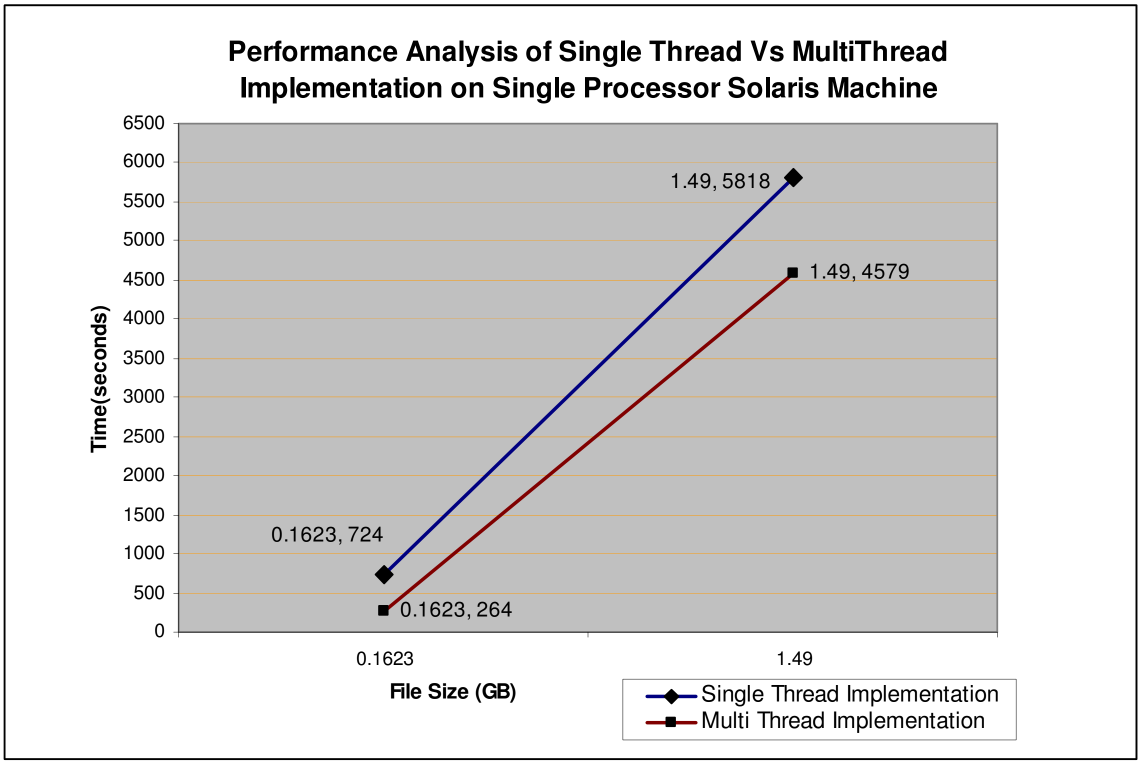
5.1 Results

1. Performance analysis of Single Thread Implementation on single processor Solaris and multi-processor Mac machines.



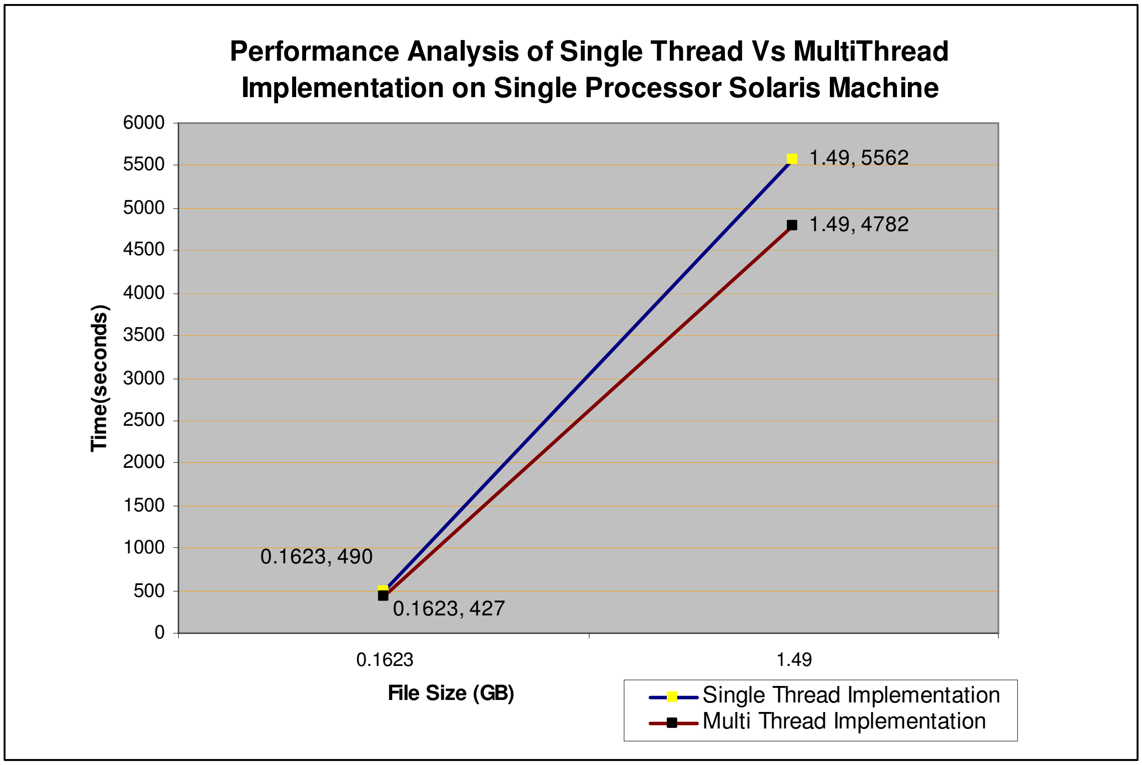
Single Thread implementation is currently used for file accessing in the Spiegel project. The graph of Test 1 shows the performance of Single Thread implementation for different file sizes carried out on multi-processor Mac and single-processor Solaris machines. The time values plotted in the graph are the average of four tests executed on the same file for different time delta. The graph shows that negligible time difference is observed based on whether the file is executed on multi-processor machine or a single-processor machine.

2. Performance of Single Thread vs. Multi-thread Implementation on single processor Solaris machine.



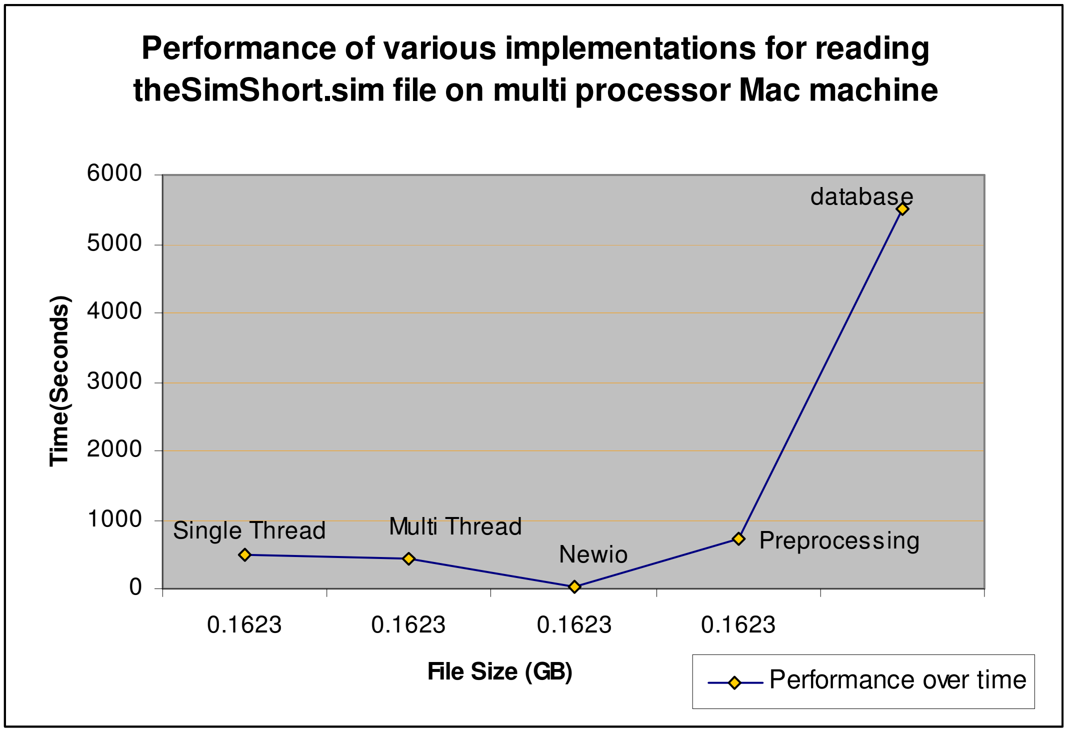
The above graph shows the performance gain of Multithread Implementation over Single Thread Implementation for any file size. The above results were collected by using 10 threads in the Multithread Implementation and the tests were carried out on two files with file sizes 0.163GB and 1.49 GB. For any file, we can conclude that the performance of Multithread Implementation will be better than Single Thread Implementation because in Multithread Implementation 10 threads in this case start reading at the same time. So for consecutive requests the wait time is significantly lowered.

3. Performance of Single Thread vs. Multi-thread Implementation on multi-processor Mac machine.



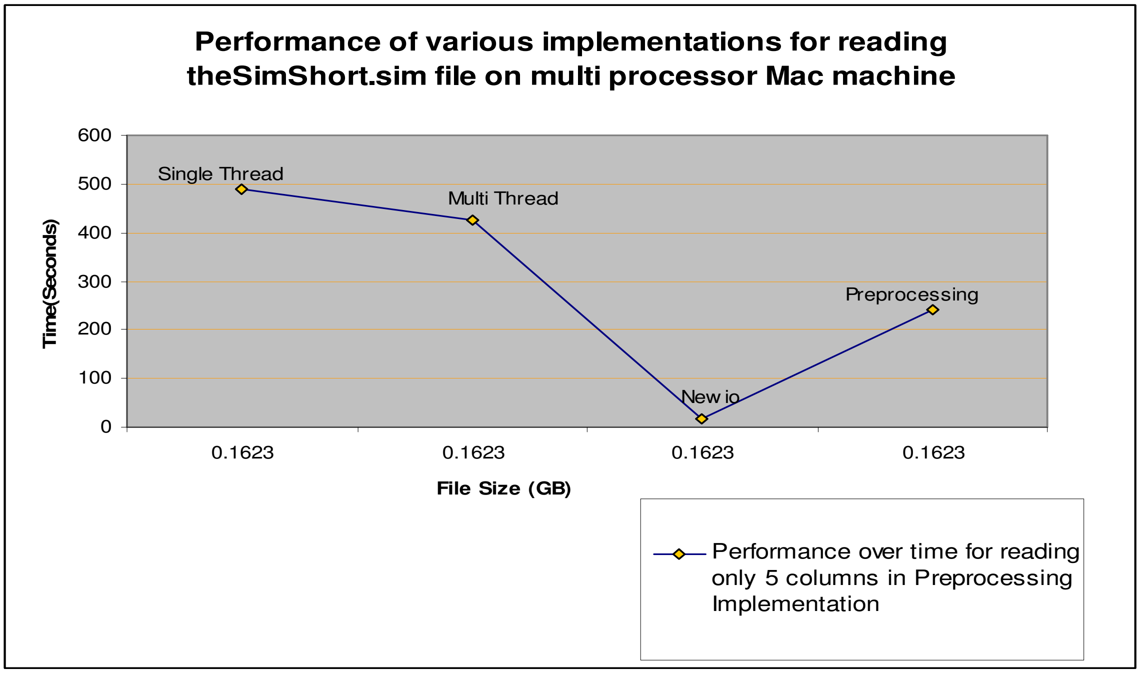
The performance gain of Multithread implementation over Single thread implementation for small files is negligible on multi-processor machines. But for larger size files, Multithread Implementation performs significantly better than Single Thread implementation.

4. Performance analyses of accessing theSimShort.sim file.



The above graph depicts the performance of various implementations for reading theSimShort.sim file which is of size 0.162 GB. The New I/O implementation performs the best possibly due to the memory mapping mechanism supported by Java New I/O API classes. Database implementation performs the worst because of ResultSet iterating which has been done as the ResultSet is not serializable, to send the resultant data to the client. Preprocessing implementation results plotted in the graph above are for reading all the 18 columns in the field. The increased overhead of file I/O for reading data from 18 files as opposed to reading data from a single file is possible the reason for its performance as seen in the graph.

6. Performance analyses of accessing theSimShort.sim file by all the proposed implementations in the project with variation in data reading for Preprocessing Implementation.

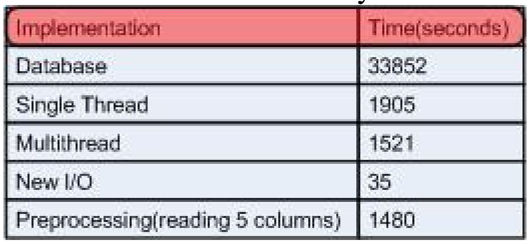


Preprocessing Implementation performs better than Multithread and Single Thread implementation when time, particle id, x, y and z columns are read and sent to the client. This shows that by reducing the number of bytes read, we can improve I/O performance.

5.2 Result Summary

Based on the various experiments carried out for testing, the following table gives the summary about the average time taken by each implementation on any platform to read 1 GB of data and send it to the client.

Result Summary



Implementation	Time(seconds)
Database	33852
Single Thread	1905
Multithread	1521
New I/O	35
Preprocessing(reading 5 columns)	1480

Time taken by Preprocessing implementation varies because the Client has the option to specify the attributes of interest. The test results proved that by reducing the number of bytes read, Preprocessing implementation can perform better than Multithread and Single Thread implementation.

The performance of Database implementation for reading 1 Gb of data is inferred from the tests performed in a 0.169 Gb file. As it did not perform well on a smaller size file further tests on that implementation were not carried out for large size files.

Chapter 6

6. Lessons Learnt

The project proposed and implemented 5 methods namely Database, Single Thread, Multithread, New I/O and Preprocessing implementations successfully. But the results of some of the implementations didn't appear as anticipated.

The Database implementation was anticipated to perform well because databases are designed to store, manage and retrieve large amounts of data. This fact was partially true because in the project, querying the database took the least amount of time to fetch all the data of one time block. The unfortunate part of this implementation was query formation and `Resultset` iteration. To avoid redundant database querying for those particles that are present at every time block, a map was created which contained information about all the unique particles to be fetched for that time request and the database was queried based on that map information. To avoid multiple database query overhead, a large query was created based on the information stored in the map.

This action added a query formation delay in serving the request. It took a significant amount of time to create a large query. Apart from this, to send data to the client side data had to be extracted from the `Resultset` in a string format and then sent over to the client. Iterating through the `Resultset` was another significant bottleneck of this approach.

By introducing some redundancy by querying all particles from the database till the requested time and by implementing the file reading mechanism similar to the Single Thread implementation the query formation delay could be eliminated. But this would add the multiple database querying overhead and there is no solution to the `Resultset` iteration problem. Even an enhancement in the caching mechanisms won't significantly improve the performance of this implementation because the `Resultset` iteration bottleneck cannot be addressed by it. There is a possibility that this implementation can be as good as Single Thread implementation by using some type of caching and buffering mechanisms and file reading mechanism similar to Single Thread implementation. As the project goal was to provide a solution better than the Single Thread implementation variations on the Database implementation were not explored.

The Multithread Implementation was proposed to give better performance as compared to Single Thread implementation because multiple threads can start reading the file in parallel. The performance of this implementation depended on the number of threads initialized to start reading the file in parallel. It may seem that the time taken to serve a request would be directly proportional to the number of threads initialized. But this behavior was not observed during testing. When the testing was performed on a megabyte file, there was some performance saturation after increasing the number of threads above a certain value. This value cannot be fixed as it depends on the size of file and has to be found out by trial and error methods where the performance is the best. For the megabyte test file used in the project the performance increased by three times when the number of threads were increased from 2 to 10, but the performance improvement was minimal when the number of threads were increased from 10 to 30 which was the MAX thread number where each thread is responsible to read one time block. When tests were performed on a gigabyte file there were "heap overflow" errors when the value of threads was increased to the MAX thread value. To avoid such errors the value of thread numbers have to be chosen carefully considering the file size and the available resources.

So while using this implementation for larger files and greater number of threads, the heap size of the JVM has to be increased by passing one o the -X flags to Java during program start. Due to limited resources all the tests performed in this project for gigabyte files were executed with 10 threads and to maintain consistency among the results, tests on megabyte files were also executed with 10 threads maximum. For a two gigabyte file the heap space limit required was approximately more than 1.5 gigabyte. So for larger files, the number of threads must be increased carefully based on the available heap space for the JVM.

The performance of this implementation varied to a considerable extent on a Single processor machine and a Multiprocessor machine. The difference in performance gain on a Single processor machine between Single Thread implementation and Multithread implementation was approximately by three times for a megabyte file whereas on a Multiprocessor machine it was very minimal. The reason for this could be the chip architecture of Multiprocessor systems [Ref. Hans-Peter Bischof]. But for gigabyte files the performance was approximately two times on both the systems when compared between Multithread and Single Thread implementations.

The New I/O implementation performed very well but had specific memory requirements. The performance of this implementation depended on the garbage collection performed by the JVM. There is no way except garbage collection that the mapping between memory and the files be removed. So if the same program is executed consecutively multiple times a "`Not Enough Memory`" exception is thrown. There is no solution to this except wait for sometime till the garbage collections is done and then re-execute the program. Though this is a limitation of this implementation the performance of this implementation has no match. It outperforms all the proposed implementations in this project. This implementation works well for 2 gigabyte files but has not been tested for terabyte files, because of memory limitations.

The Preprocessing implementation was implemented with the thought that by reducing the number of bytes read we can improve I/O performance. And as expected by reducing the number of columns to be read from the file the time taken to serve client request reduced accordingly. But if all columns from the file are requested by the client then this implementation will perform worst than Single Thread implementation because the File I/O overhead increases drastically.

Chapter 8

8. Future Work

This project can be extended by implementing these solutions on a distributed system. To obtain the advantage of multiple Servers/Nodes the data file should be distributed in a specialized way on all the Servers. The main Server fetches client requests and knows how to distribute the task to different Servers to read the file and get the maximum performance benefit. The diagram below presents the architecture of this approach. The performance of this approach will significantly depend on how the file will be distributed among all the work stations.

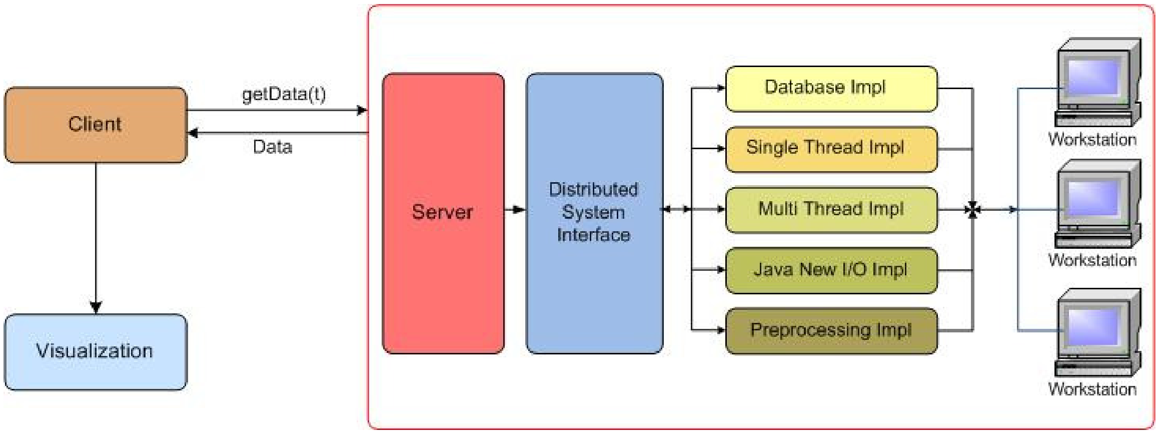


Figure: Proposed future enhancement solution for the project.

Apart from this, some type of high performance file system could also be implemented to improve I/O performance. [3] Such file systems have specialized data storage formats, efficient data search algorithms, intelligent caching techniques etc. These file systems are developed for managing very large amount of data. So these features may be explored to achieve performance gain by I/O reduction in this project.

Due to strict time constraints, this project could not explore these possibilities. But they are anticipated to give better performance as compared to the current implementation.

Chapter 9

9. Conclusion

The project identified the possible causes for poor performance of I/O and successfully implemented the proposed solutions to improve the I/O performance. Among the proposed solutions, the Database solution did not perform as intended and gave extremely poor results. New I/O implementation performed extremely good but requires considerable amount of system resources in the form of memory to obtain the intended performance.

Multithread implementation and preprocessing implementation performed well based on the specific parameters like thread number, columns to be read etc and the platforms on which they were executed. The performance of Multithread implementation on single processor Solaris machine was much better than its performance on multiprocessor Mac machine. All the results presented in the report were taken with 10 threads executing in parallel. The number of threads could be increased or decreased based on the file size or system resources. By increasing the number of threads it may happen that `“heap over flow”` exceptions will be thrown.

The performance of Preprocessing implementation depends on the platform on which it is executed and the size of the request sent to it by the client. If the number of columns requested by the client is more, the performance will be nearly as good as Multithread implementation or even worst if all file attributes are requested. The reason for this is that by doing so the overhead of file I/O increases and it performs poorly even as compared to Single Thread implementation.

The project proposed, implemented and tested five different implementations namely Database, Single Thread implementation, Multithread Implementation, New I/O and Preprocessing Implementation successfully. It proved to be a great learning experience and offered an in depth understanding about Java, New I/O, Databases and many other new technologies.

Chapter 10

10. Reference

[1] "Java New I/O" by Ron Hitchens, Publisher- O'Reilly First Edition August 2002

[2] http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html

[3] http://pages.cs.wisc.edu/~bolo/shipyard/hpfs.html

[4] http://spiegel.cs.rit.edu/~hpb/grapecluster/Spiegel/index.html