# PERFORMANCE EVALUATION AND ENHANCEMENT OF DENDRO

BY

JAYANTA MUKHERJEE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor William D. Gropp

# Abstract

DENDRO [1] is a collection of tools for solving Finite Element problems in parallel. This package is written in C++ using the standard template library (STL) and uses the Message Passing (MPI) [2]. Dendro uses an octree data-structure to solve image-registration problems using finite element techniques. For analyzing the behavior of the package in terms of speed-up and scalability, it is important to know which part of the package is consuming most of the execution-time. The single node performance and the overall performance of the package is dependent on the code-organization and class-hierarchy. We used the PETSC [3] profiler to collect the performance statistics and instrument the code to know which part of the code takes most of the time. Along with the function-specific execution timings, PETSC profiler also provides the information regarding how many floating point operations is being performed in total and on average (FLOP/second). PETSC also provides information related to memory usage and number of MPI messages and reductions being performed to execute that particular function. We have analyzed these performance-statistics to provide some guidelines to how we can make Dendro more efficient by optimizing certain functions. We obtained around 12X speedup over the performance of (default) Dendro by using compiler-provided optimizations and achieved more than 65% speedup over compiler optimized performance (20X over the naive Dendro performance) by manually tuning some-block of code along with the compiler-optimizations.

*Dedicated to my parents and family*

# Acknowledgments

I would like to thank my adviser, Professor William D. Gropp, who read my numerous revisions and helped me with his valuable suggestions and experience. I am grateful to NSF for funding this work (Grant Number 0849301). I am thankful to my parents, my elder brother, my cousins and my extended family. I would like to thank my roommates Nikhil, Abhimanyu, Subhabrata, Raj, Rannesh, Aditya, Prannoy for being patient and providing me a home away from home. I really love you buddies. I would like to thank Sastry, Amit, Anoop, Aftab, Rajarshi, Soumi, Pradeep for making my stay at Urbana-Champaign so special. I would like to thank Prof Indranil Gupta, Prof Ralph Johnson for their encouragement. I would like to thank the developers of Dendro Rahul Sampath and Hari Sundar who helped me during the performance study by their constructive advices and suggestions. I would also like to thank the PETSC developers and support group for their valuable feedbacks. I would like to thank Michael Campbell and the Turing Support-team for their help.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

Dendro is designed to solve image registration problems arise from medical applications. **Image registration** [4] is the process of transforming different sets of data into one coordinate system. Registration is necessary in order to be able to compare or integrate the data obtained from these different measurements. Registering and summing multiple exposures of the same scene improves signal to noise ratio, allowing to see things previously impossible to see.

**Medical image registration** [4] (for data of the same patient taken at different points in time such as change detection or tumor monitoring)) often additionally involves elastic (also known as nonrigid) registration to cope with deformation of the subject (due to breathing, anatomical changes, and so forth).

The Dendro package [1] is written to solve finite element problems. It has specific modules to improve scalability. The package contains geometric multigrid solvers which solves elastic problems. Dendro also contains some visualization modules. The elasticity solver reads or initializes input points and create an octree using those points. Then, it balances the octree and solve it using PETSC.

## 1.1 Motivation

The objective of this work is to figure out the performance bottlenecks that affects the speedup and scalability of the applications. More specifically finding the causes which impedes the scalability of a parallel solver in Dendro and eradicate them to achieve better performance was the goal of this work. The project aims to figure out some of the issues related to I/O which we observed as a serious problem to improve scalability. The code is written in an Object-Oriented fashion. Dendro can be viewed from the top as a collection of the following components:

- Source codes, libraries, header files and examples
- Data and scripts to run the package

In order to extract the profiling informations for Dendro, the PETSC profiler has been used, as gprof and other common profiling tools are difficult to use as the different modules of Dendro are parallel in nature and uses message-passing to communicate between different processes. As different processes have separate address space and they are working on different part of the memory (distributed memory), the conventional profiling tools (for serial applications) is not suitable for profiling Dendro. Rather than looking at the libraries in Dendro, a better approach can be looking at the overall (functional) organization of Dendro.

## 1.2 Organization of Dendro

Dendro is a parallel Geometric Multigrid-based finite element method solver [1]. It performs the following 4 operations in order to solve for the displacements at the non-hanging octree vertices.

1. **Main Stage** : Deals with the object creation and reading (or generating) input points.
2. **Points to Octree Creation (P2O)** : In this stage, the code creates the octree out of the points it read or generated (using Gaussian Distribution).
3. **Balance (Bal)** : In this stage, the code performs 2:1 balancing [5].
4. **Solve** : This is the actual solver which calculates the displacements for the non-hanging nodes using matrix-free method [6].

The solver workflow has been shown in the Figure 1.1 The solver starts with reading and splitting up the points to be used by each processor. The points then converted into the octree in a linearised form to improve cache performance. The octree is balanced in 2:1 ratio as described by Sundar et al [5].

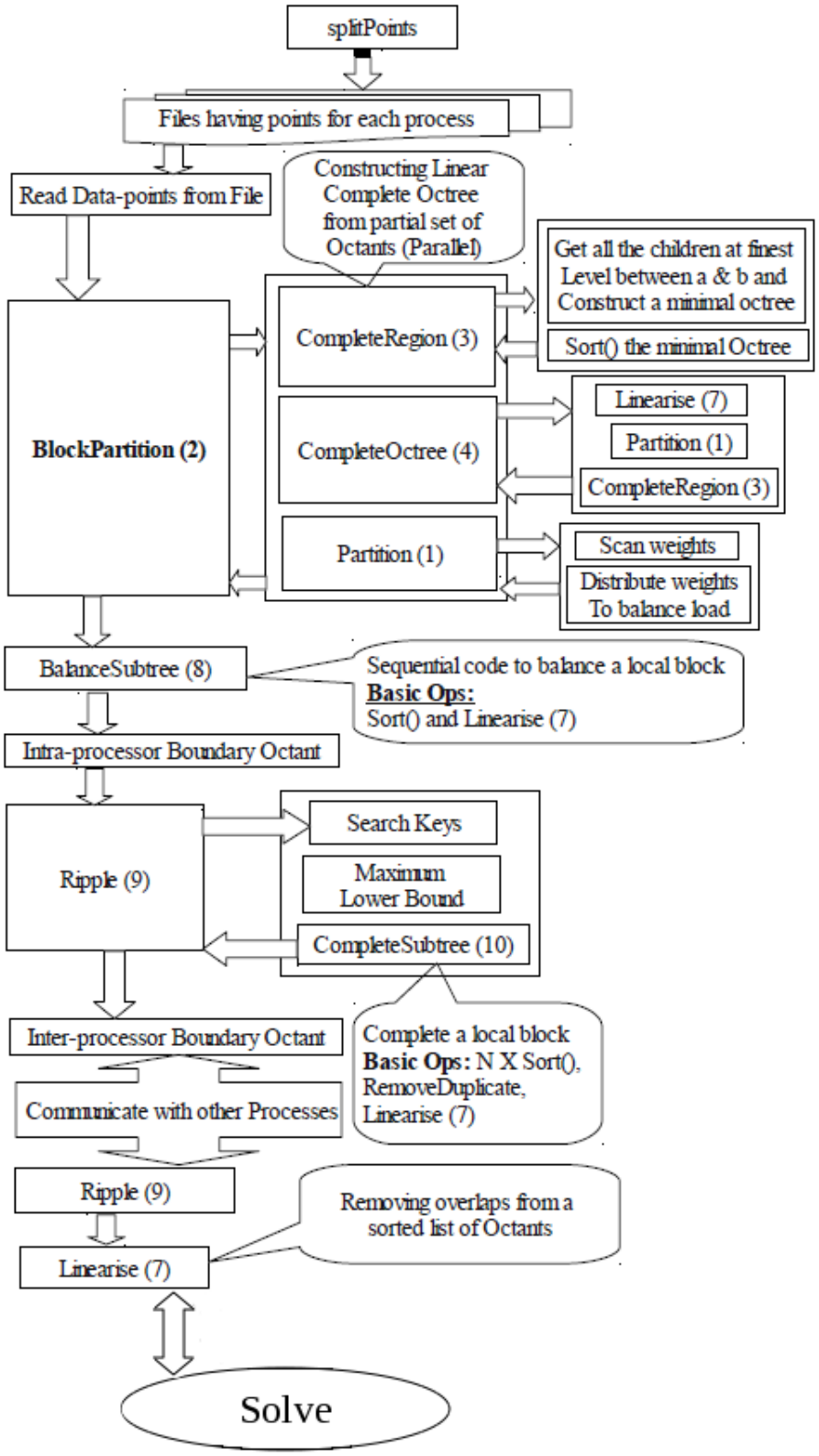


Figure 1.1: Workflow of Dendro

# Chapter 2

# Related Work

Performance optimization and modeling [7] for different well-known, popular packages throw various new challenges to the developers. Understanding and enhancing the performance of large-scale scientific applications is a crucial component of the high-performance computing. One of the major problem in modeling such complex softwares lies in identifying the costly (time-consuming) modules and the dependencies correctly and efficiently. Memory hierarchy and cache behavior [8] makes the performance analysis more complicated.

Using sophisticated data-structures the memory access overheads can be reduced significantly. Brodal, G. described [9] data structures that automatically apply to multi-level memory hierarchies to become cache-oblivious. Dendro [1] is used for solving medical image registration problems. Brown L. described a broad range of techniques [10] that have been developed for various types of registration techniques which can most appropriately be applied. She categorized the the variations in three major types. The first type are the variations due to the differences in acquisition which cause the images to be misaligned. To register images, a spatial transformation is found which will remove these variations. The class of transformations which must be searched to find the optimal transformation is determined by knowledge about the variations of this type. The transformation class in turn influences the general technique that should be taken. The second type of variations are those which are also due to differences in acquisition, but cannot be modeled easily such as lighting and atmospheric conditions. This type usually effects intensity values, but they may also be spatial, such as perspective distortions. The third type of variations are differences in the images that are of interest such as object movements, growths, or other scene changes. Variations of the second and third type are not directly removed by registration, but they make registration more difficult since an exact match is no longer possible. In particular, it is critical that variations of the third type are not removed. Knowledge about the characteristics of each type of variation effect the choice of feature space, similarity measure, search space, and search strategy which will make up the final technique. She also mentioned that [10] all registration techniques can be viewed as different combinations of these choices. This framework is useful for understanding the merits and relationships between the wide variety of existing techniques and for assisting in the selection of the most suitable technique for a specific problem.

Schumacher et al presented [11] a novel method for iterative reconstruction of high resolution images. The method is based on the observation that constant regions in an image can be represented at much lower resolution than region with fine details and combined adaptive refinement based on quadtrees with iterative reconstruction to reduce the computational costs. They have got a speed up factor of approximately two compared to a standard multi-level method.

Haber et al introduced [12] the concept of octrees for registration which drastically reduces the number of processed data and thus the computational costs. They showed how to map the registration problem onto an octree and presented a suitable optimization technique. Furthermore, we demonstrated that the performance (computational time) can be improved by a factor of 10 compared with standard approaches. In 2007 Haber et al [13] come up with an adaptive mesh refinement approach using discretization of the variational form which significantly reduce the problem size and thereby reduce the computational time by a factor of 10 or so compared to the non-adaptive approach.

Modeling the performance and enhancing it on high-end computing systems at an enormous scale, increasing architectural complexity, and increasing application complexity is non-trivial. There are models like LogP [14], LogGP [15] which address the communication overhead associated with parallel softwares, but, none of them capture the overlapping effects of computation and communication as well as the synchronization overheads. The LogGP model [15] is an extension of the LogP model [14] for parallel computation which abstracts the communication of fixed-sized short messages through the use of four parameters: the communication latency (L), overhead (o), bandwidth (g), and the number of processors (P). The LogP model can accurately predict communication performance when only short messages are sent. However, many existing parallel machines have special support for long messages and achieve a much higher bandwidth for long messages compared to short messages. Hoefler et al [16] developed a new low-overhead measurement method which also assesses protocol changes in the underlying transport layers. They used the gathered parameters to simulate LogGP models of collective operations and demonstrate the errors in common benchmarking methods for collective operations. The simulations provide new insight into the nature of collective algorithms and their pipelining properties. They showed that the error of conventional benchmark methods can grow linearly with the system size. An ambitious research plan can include encompassing performance modeling and prediction, automatic performance optimization and performance engineering of high profile applications. The principal new component is a research activity [17] in automatic tuning software, which is spurred by the strong user preference for automatic tools.

Various all-to-all personalized communication (AAPC) algorithms dominate the communication patterns of the overall applications. Communication patterns affect the scalability and overall performance of an ap-

plication. Shan H. et al [18] developed a sequence of performance models using a series of micro-benchmarks. Multithreading can improve performance on a single SMP node [19], but the overall performance is dependent on the communication patterns. Better computation-communication overlap can lead to better scalability and performance of the application. They found that for SMP based systems the most important performance constraint is node-adapter contention, while for 3D-Torus topologies good performance models are not possible without considering link contention. The best average model prediction error is very low on SMP based systems with of 3% to 7%. On torus based systems errors of 29% are higher but optimized performance can again be predicted within 8% in some cases. These excellent results across five different systems indicate that this methodology for performance modeling can be applied to a large class of algorithms [18].

In Dendro, the octree-nodes are placed in linearized format to improve locality to achieve better performance. The octree is balanced in 2:1 ratio as described by Sundar et al [5]. But as Campbell et al demonstrated that [20] the ordering based on Hilbert Curves is preferred to the Morton ordering.

# Chapter 3

# Experiments

We have performed experiments to determine which part of the software package is consuming most of the time and resources. The PETSC profiler has been used to find which functions are taking how much time and how frequently they are called. In the following sections, we have explained the experimental strategies being adopted with the rationale of performing them to extract more detailed informations.

## 3.1 Experiment 1: Instrument the Code

We started with the example codes in the Dendro package. Dendro is designed to solve image-registration problem by solving elastic problems. The elasticity solver has been instrumented with `MPI_Wtime` to get timing information. We have plotted the timing distribution for the 4 basic blocks in Figure 3.1 for running it on varying number of processor while keeping the number of local points per processor equal to 1000. We can see that the **Solve** part of the routing dominates the timing. To investigate the performance of the solver (`newElasSolver`)the number of points local to each of the processor is being taken as an input (**weak-scalability** approach). That is equivalent to provide same work-load on each of the processor.

We observe that, even with the increase in number of processors, of the number of local points to a processor has been kept constant, then, the timing and the Flops does not vary much. From Figure 3.1, it is evident that Solve (`DAMGSolve` in PETSC) takes most of the time. So, we went deeper into the modules of the Multigrid Solver to find which functions contribute to most of the time. The issue is both time, efficiency and scalability. The I/O overheads can be impediment to the scalability of any application. The I/O overheads and parallel I/O issues has been analyzed in Section A.2 of the Appendix.

## 3.2 Experiment:2: Performance Statistics of Dendro

We have used the PETSC profiler output to analyze the performance statistics. By looking at Figure 3.1 we can say that the Solver takes most of the time.We have analyzed the performance statistics for Dendro multigrid Solver and plotted them in Figure 3.2.

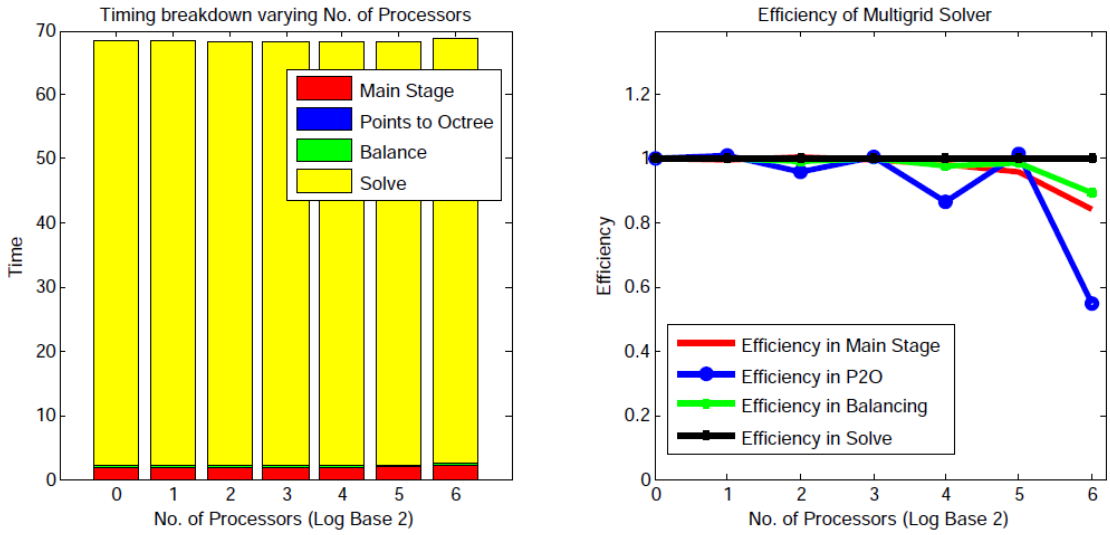


Figure 3.1: Scalability Study of the Multigrid Solver of Dendro

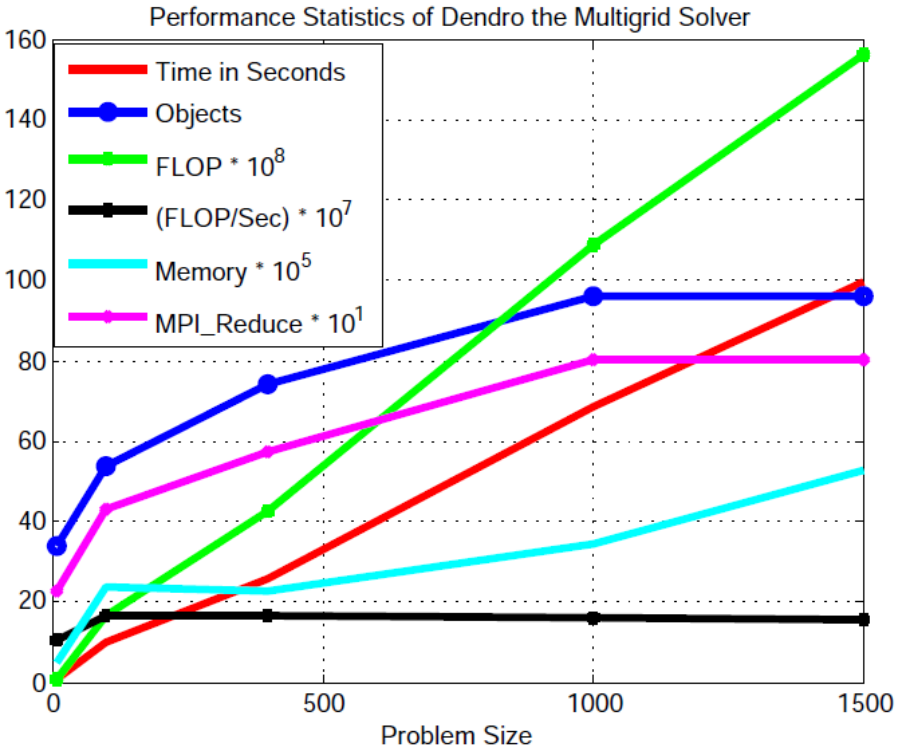


Figure 3.2: Performance Statistics of the Multigrid Solver of Dendro

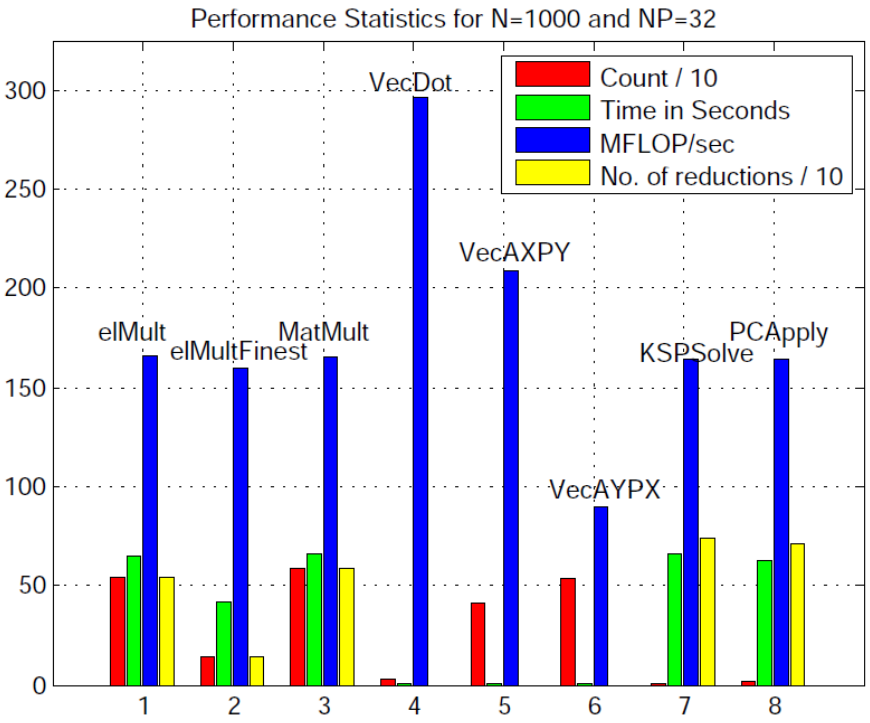


Figure 3.3: Breakdown of the timing and performance of the Multigrid Solver of Dendro

The units for each curve in the figure are not same. The legends in the plot signify how much each quantity has been scaled to be put in the same graph for facilitating comparisons. Also, it will be interesting to know which components of the Solver take more time. From Figure 3.3, we can definitely observe that, `MatMult()` and `KSPSolve()` are taking most of the time. The performance statistics of the underlying functions also need to be analyzed in this section. For convenience of analysis, We have classified the functions into following 3 groups and plotted them separately.

1. **Matrix functions**
   - MatMult
   - elMultFinest
2. Vector functions
   - VecDot
   - VecAXPY
   - VecAYPX
3. Krylov Solver

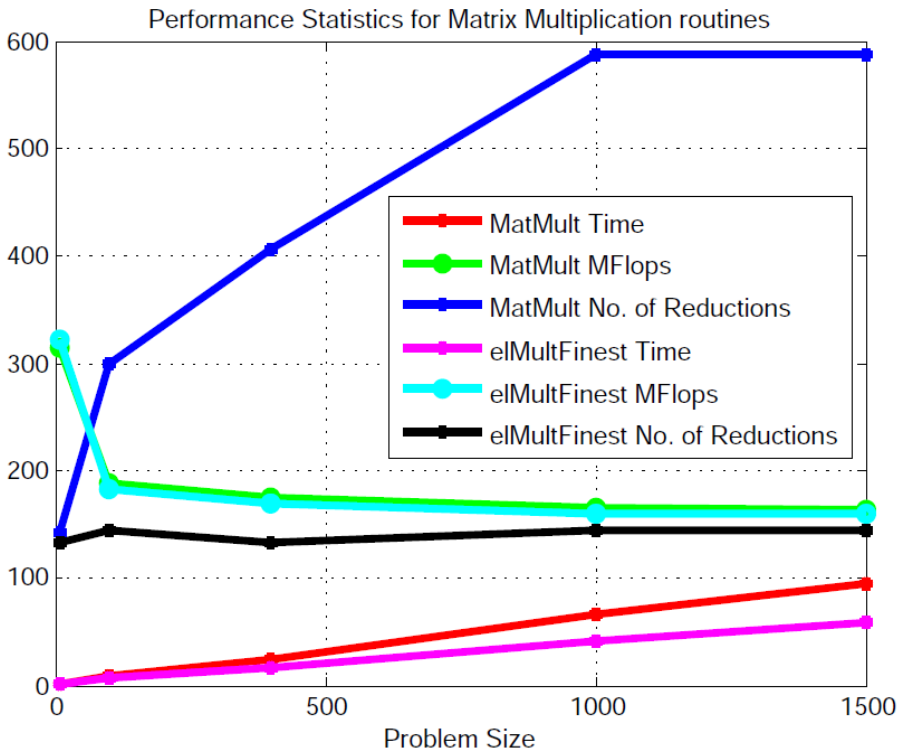


Figure 3.4: Performance Statistics for Matrix functions

- KSPSolve
- PCApply

We have plotted the Performance Statistics for Matrix Multiplication routines in Figure 3.4. One nice thing to observe here is that the `MatMult` and `elMultFinest` have similar Flops and timing patterns, but, there is a significant different in the number of reductions they perform. The `elMultFinest` is a part of the `MatMult` operation being performed. In Figure 3.5, the performance statistics for Vector operations has been performed with the required scaling to fit the data in the plot.

Figure 3.6 contains the performance statistics for the `KSPSolve` and `PCApply` routine.

The `KSPSolve` and `PCApply` has almost similar characteristics as being shown in Figure 3.6.

Impact of the memory requirements of different functions on performance needs to be addressed to optimize performance. In the next section we have studied the memory requirement for different modules of Dendro.

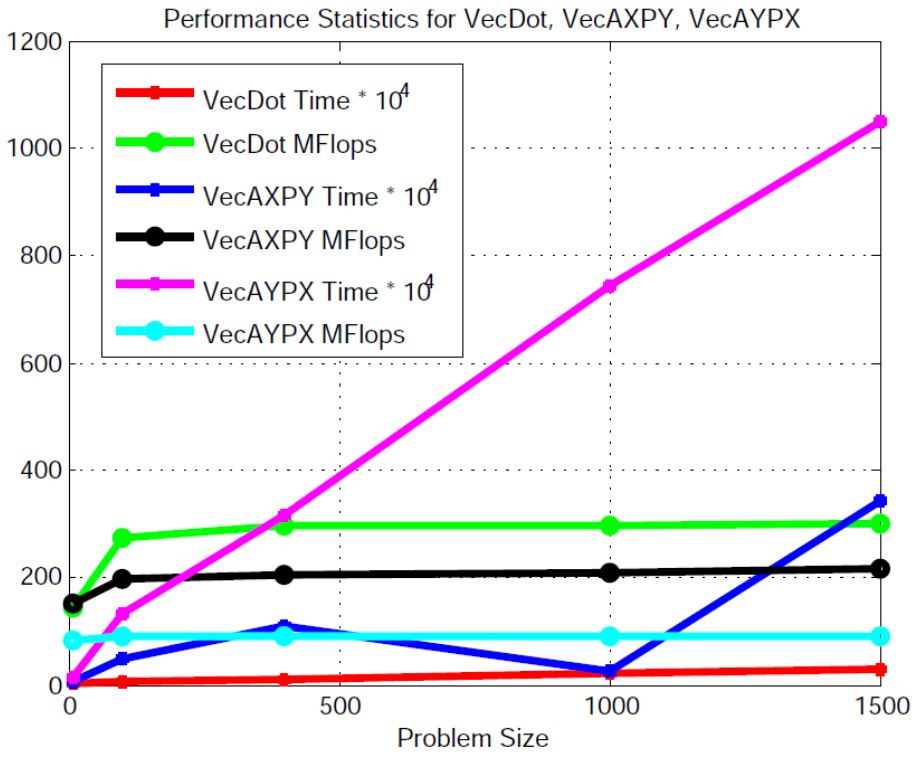


Figure 3.5: Performance Statistics for Vector functions

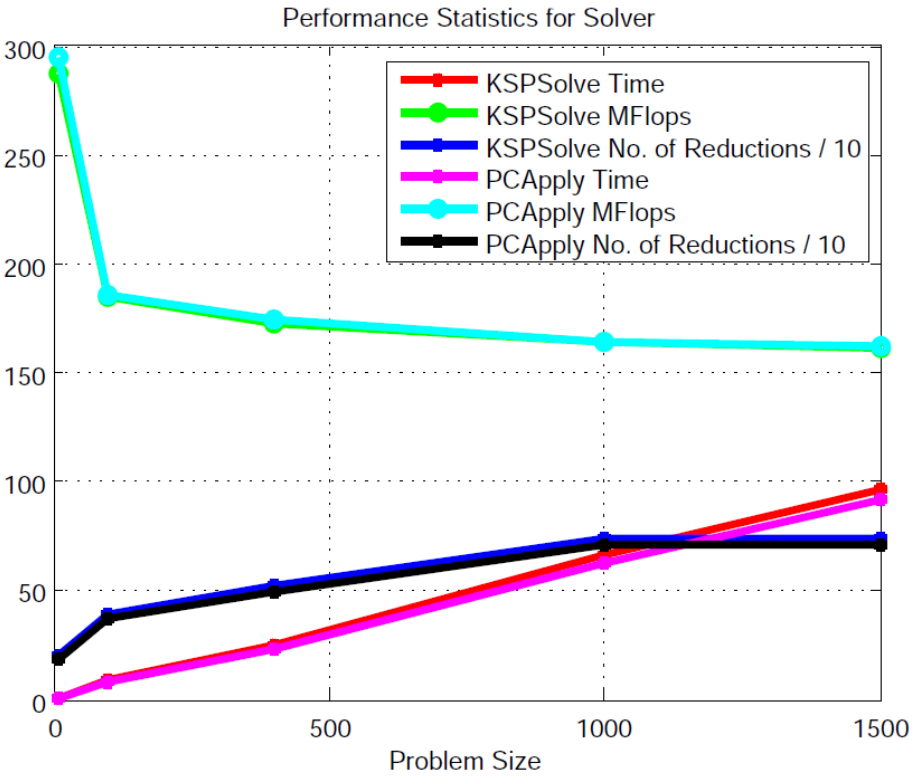


Figure 3.6: Performance Statistics for Krylov Solver

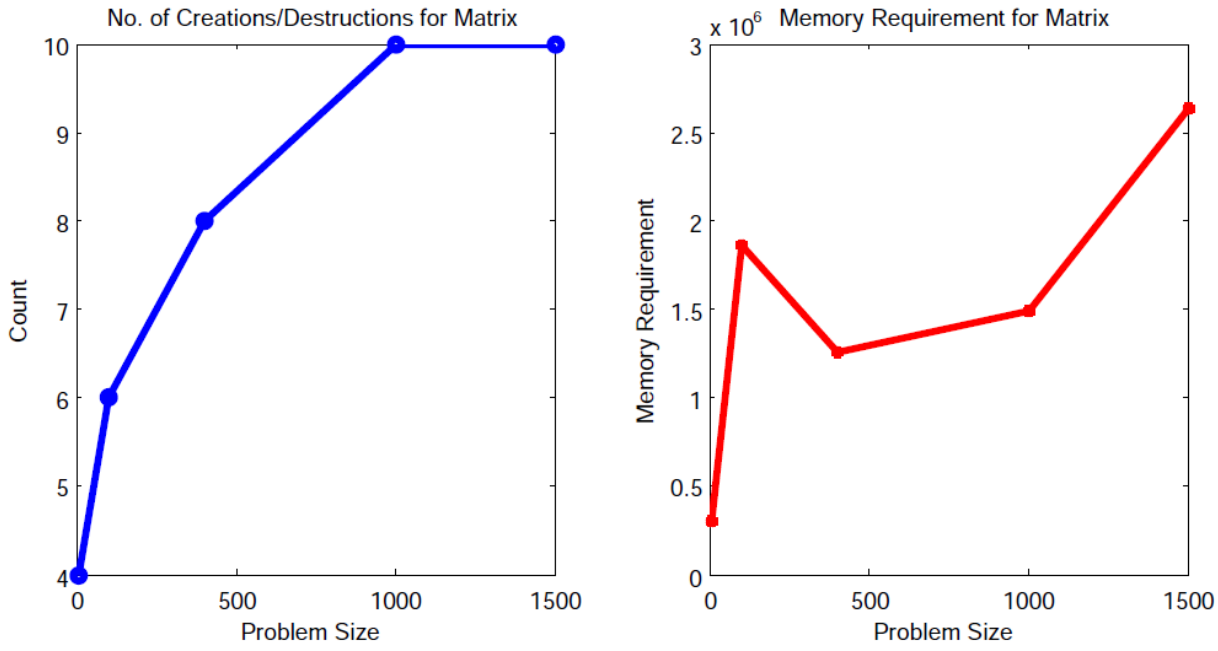


Figure 3.7: Memory Requirements for performing Mat

## 3.3 Experiment-3: Memory Requirements and Performance Statistics

In this section, We have plotted the number of times the operations are being performed and the associated memory-requirement for each of them. The operations being plotted are as follows:

1. Mat
2. Vec
3. Krylov Solver

We have plotted in Figure 3.7, the number of times the Creations/Destructions for Matrix (`Mat`) has been performed with the required memory for performing that. We have plotted in Figure 3.8, the number of times the Creations/Destructions for Vector (`Vec`) has been performed with the required memory for performing that.

From the Figure 3.8, it is shown that as expected, the memory requirement for performing Vec grows linearly with the problem-size. In Figure 3.9, we have plotted the count and memory requirement for performing the Krylov Solve.

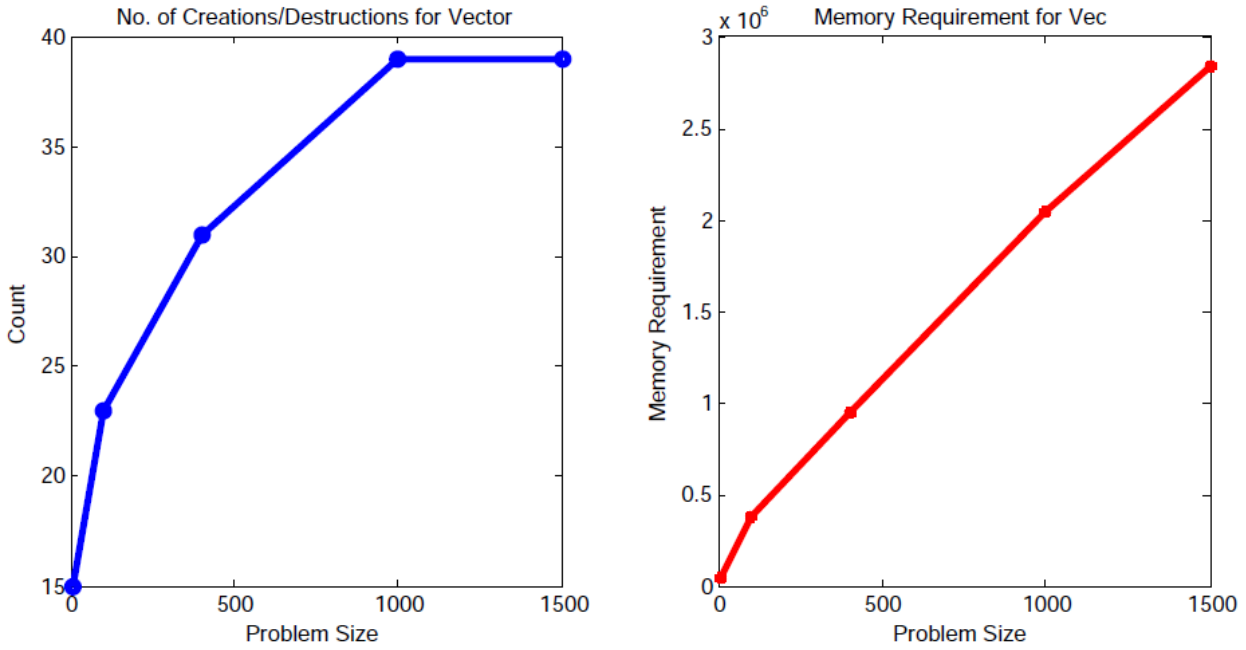


Figure 3.8: Memory Requirements for performing Vec

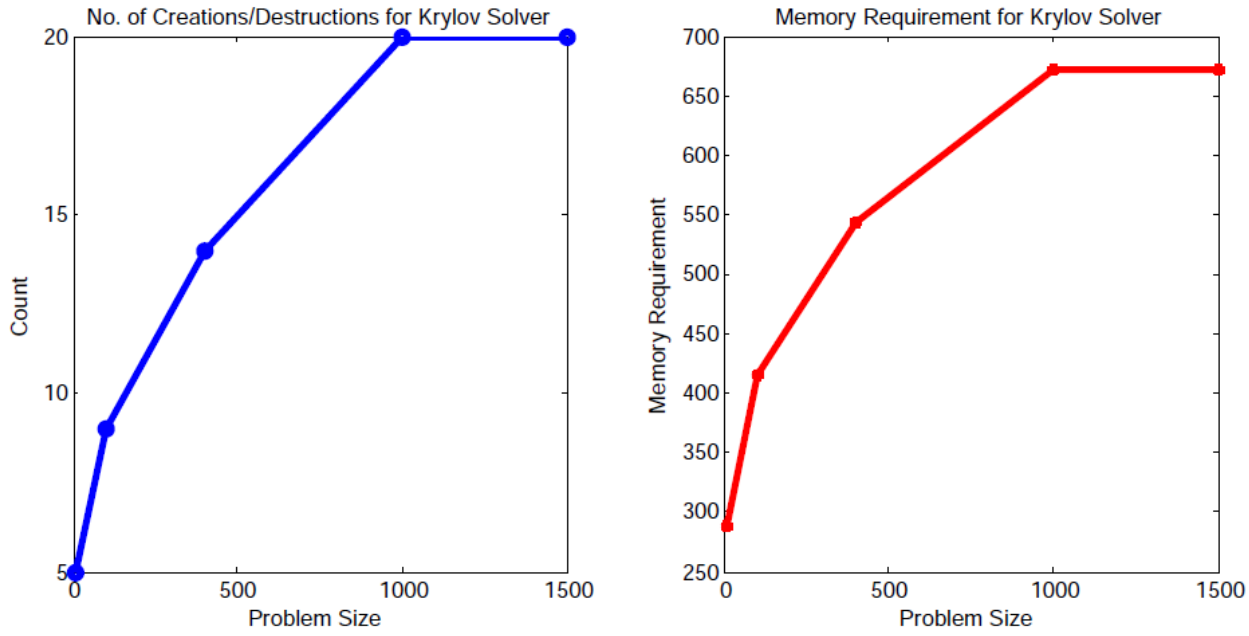


Figure 3.9: Memory Requirements for performing Krylov Solver

But, here the memory requirement does not grow rapidly from problem-size (number of points local to any processor) = 1000 to problem size = 1500.

## 3.4 Analyzing the Matrix-Vector Multiplication (Matrix-free)

From the experiments, it was clear that the KSPSolve is the solution module which takes most of the time and the `elMult` and `MatMult` is taking the majority of the solution time. So, we analyzed the code, which performs the `elMult` and `MatMult`. It is a small block of code (less than 50 lines) which is performing a matrix-vector multiplication in a matrix-free fashion and being called hundreds of times for a reasonable problem-size per processors. The `elMult` and `MatMult` is called 540 times for local number of points = 1000. The count of `elMult` indicates how many times the the Matrix-Vector multiplication routine `ElasticityMatMult()` is executed. If number of points = N, the approximated computation time is $\Theta(8N^2)$. Thus, on a 2.3 GHZ Power PC (assuming one operation per cycle), it should take

$$Time = 540 \times \Theta(8N^2) \times \frac{1}{2.3} \times 10^{-9} seconds \tag{3.1}$$

So, for a problem size (number of local points) of 1000, the time required will be

$$Time = 540 \times (8 \times 1000^2) \times \frac{1}{2.3} \times 10^{-9} seconds = 1.878 seconds \tag{3.2}$$

With the default build parameters (no optimization), the `elMult` is taking 74.175 seconds. That means it can be around 39 times faster. With compiler optimization (O3), for a problem size of 1000 (points per processor), `elMult` takes around 5.575 seconds as shown in Figure 3.10. That means the code can still be 2.97 times faster for number of points per processor is 1000. Here, we neglect the memory access related issues involved as the vectors need to be loaded and hence, cache-performance will affect the theoretically achievable time. This analysis helps us to understand how much time such an operation should theoretically take and from the experimental results determine if the performance achieved is reasonable.

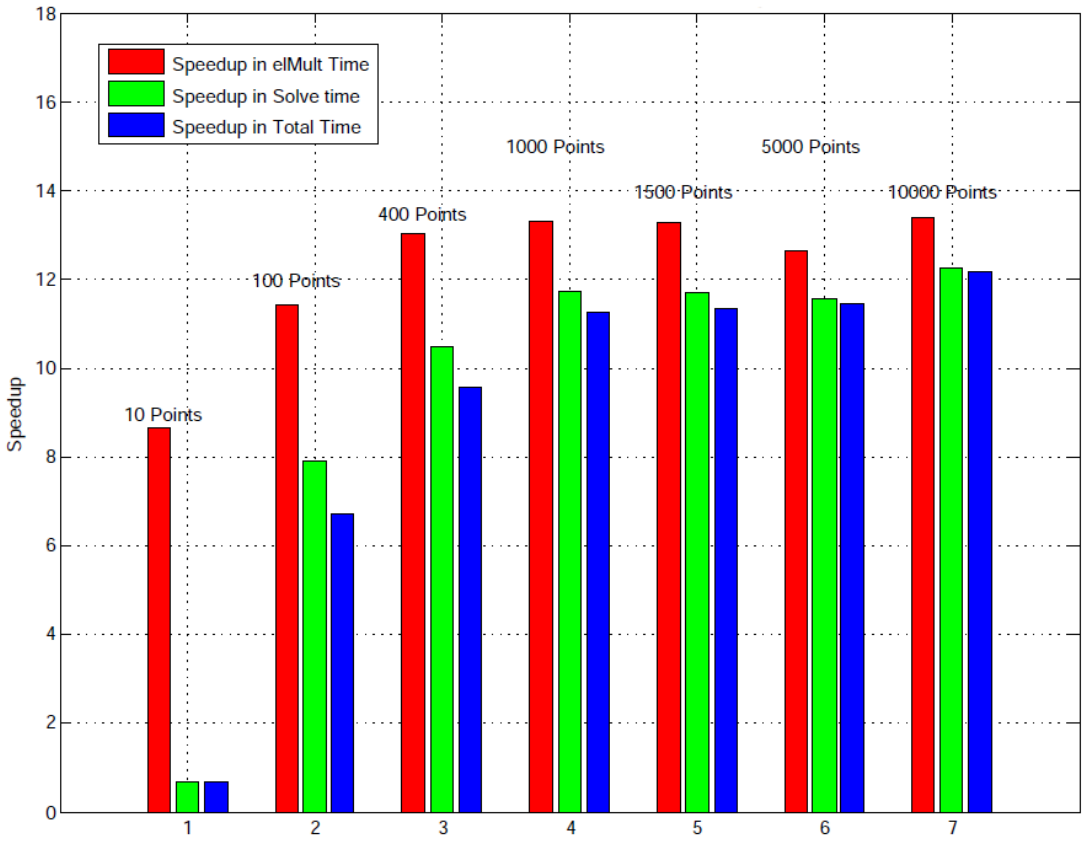


Figure 3.10: Performance Improvement by using -O3 over default Dendro Elasticity Solver (`newElasSolver`)

# Chapter 4

# Optimization Strategies

## 4.1 Performance Optimizations

Now from the analysis in the previous chapter we can say that, there is a significant order of difference in the estimated and actual performance. Although default Dendro does not compile with -O3 option, but, to provide a fair comparison, we compared the performance achieved using all the optimization strategies against the performance of Dendro using -O3 compiler optimization. We optimized the performance by adopting different strategies which will be elaborated in the following section with performance statistics.

1. Compiler directed unrolling of loops along with the -O3: Loop-unrolling should be able to provide more scope of instruction rearrangement in order to fill the no-ops to reduce stalls in the CPU-cycle.

2. Compiler-directed prefetching of loop-array with loop-unrolling and the -O3.

3. Manually unroll loops along with compiler directed prefetching of loop-array, loop-unrolling and the -O3.

4. Manually unroll loops and use temporary variables to reduce memory-access along with compiler directed prefetching of loop-array, loop-unrolling and the -O3.

5. Explicitly prefetch arrays and bind it to the cache-line, manually unroll loop and use temporary variables to reduce memory-access along with compiler directed prefetching of loop-array, loop-unrolling and the -O3.

For performing explicit prefetching of arrays we analyzed the appropriate location at which to call `__builtin_prefetch` in order to minimize the cold-misses in the cache-line. The analysis has been provided in Section 4.1.5.

From, Experiment A.2, it can be said that I/O can impede the scalability of the application. Parallel I/O provided by MPI [2] can resolve the issue by concurrent access of the file. In Section A.3, we have

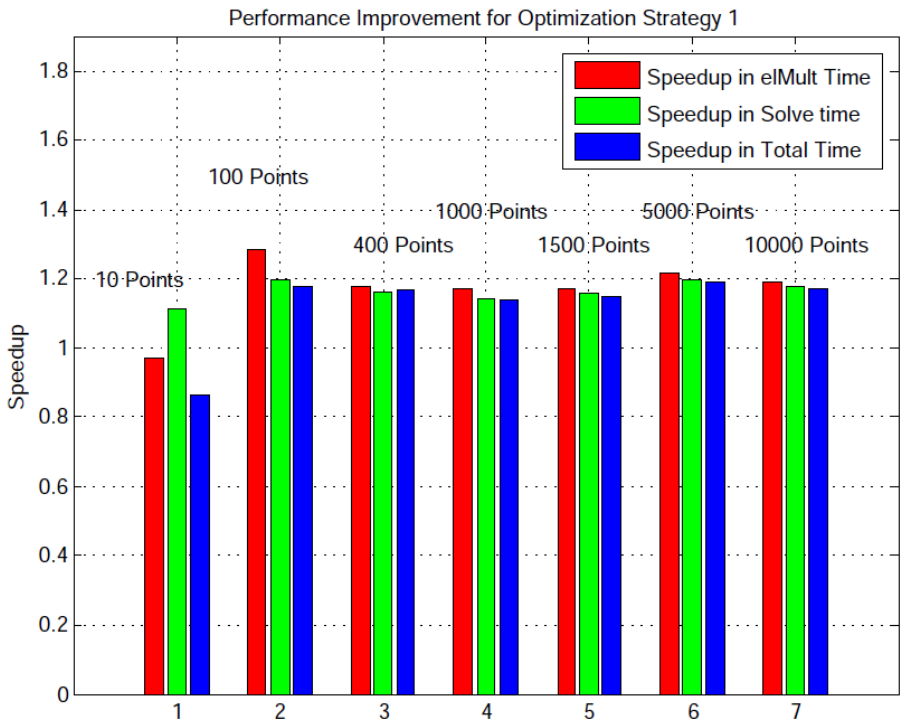


Figure 4.1: Performance Improvement by Adopting Optimization Strategy 1

explained the details about the performance improvements possible with relative merit and demerits and system-requirements to leverage the parallel I/O in order to make effective use of parallel I/O in Dendro.

Currently, Dendro is developed based on PETSC 2.3.3. We made appropriate changes to make it compatible with the latest PETSC version (3.1). We listed some of the necessary changes in order to make it compatible in the Appendix Section A.4.

### 4.1.1 Optimization Strategy 1: Compiler directed Loop-Unrolling

Modern compilers support loop-unrolling which provides more instructions to the compiler to optimize to minimize stalls (no-ops) due to dependencies on data and operations, and thereby improve performance. In Figure 4.1, we plotted the performance improvement we obtain using `-funroll-loops` compiler flag with -O3. For number of points local to a processor equal to 1000 and 10000, we have obtained more than 17% and 19% speedup for the `elMult` module which contributes to more than 13% and 17% speedup respectively.

### 4.1.2 Optimization Strategy 2: Compiler-directed prefetching with Strategy 2

Prefetching of arrays reduces the cold-miss at the cache and hence improve the cache-performance by increasing cache hit/miss ratio. We tried compiler-supported prefetch support to prefetch the loop-arrays using

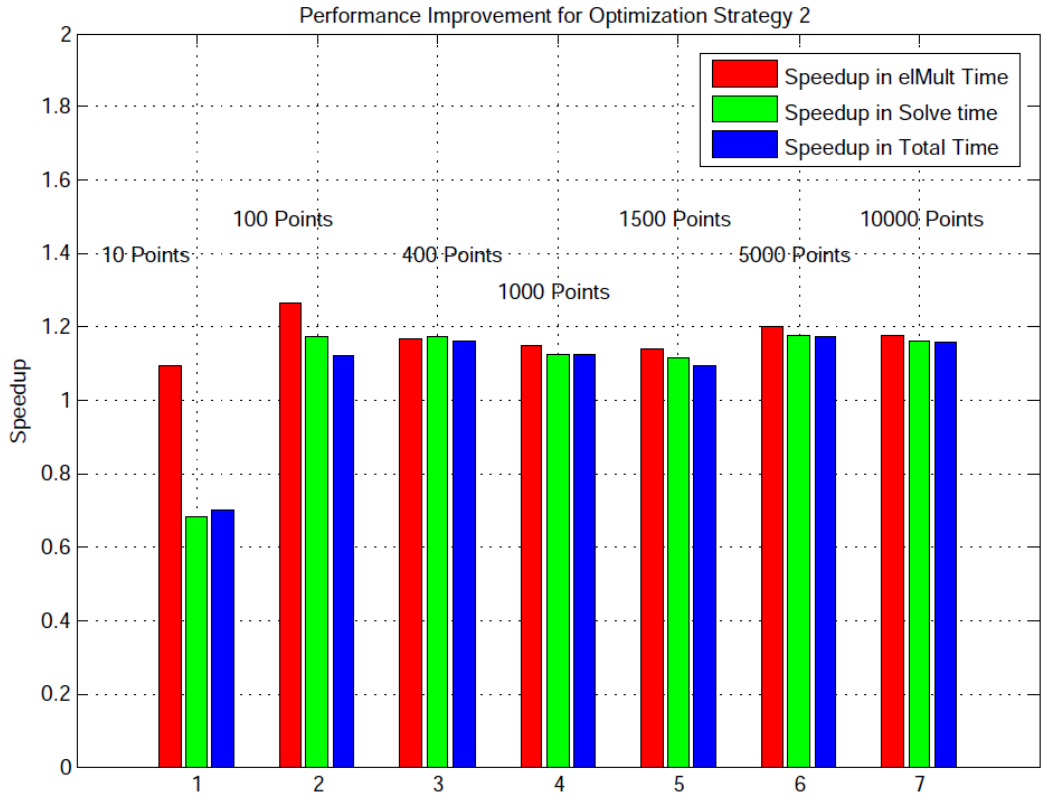


Figure 4.2: Performance Improvement by Adopting Optimization Strategy 2

the `-fprefetch-loop-arrays` flag to achieve better performance. In Figure 4.2, we present the experimental data showing the performance improvement we achieved. Although we obtained a modest performance improvement (around 12% for 1000 points and around 16% for 10K points), by prefetching the loop-arrays, it is not significantly better. This is because prefetching is a costly operation and in turn it removes some cache-lines which may be used. Thus, the performance improvement by reducing cold-misses at cache has been minimized by the extra misses that occur due to replacement of some cache-lines that may be used in the future. Also, some arrays may not fit in the cache, and for such a complex scientific application like Dendro that internally relies on the PETSC solver, it is difficult to avoid cold-misses. Thus, compiler provided prefetching may not be significantly beneficial for the elasticity solver of Dendro.

### 4.1.3 Optimization Strategy 3: Manually unroll loop with Strategy 2

Along with the optimization achieved by the compiler, we unroll the inner loop performing the Matrix-vector multiplication and rearrange the instructions in order to provide better locality. In Figure 4.3, the speedup is presented. This optimization gives us more than 21% improvement for `elMult` for local points = 1000 with an overall speedup more than 14.5%. For 10K points, it achieves more than 22% overall speedup. As

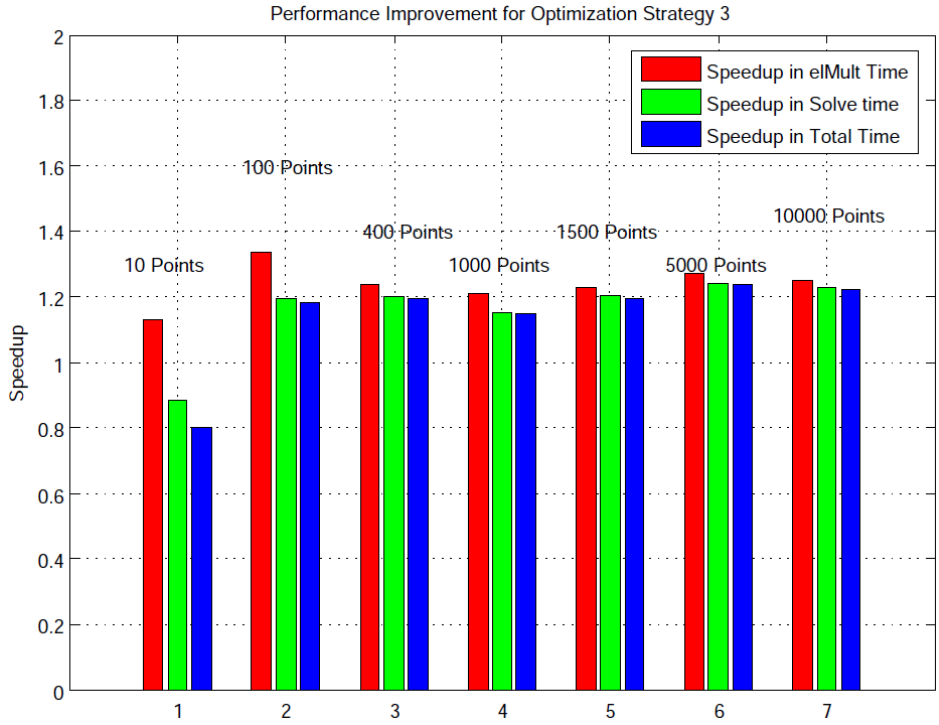


Figure 4.3: Performance Improvement by Adopting Optimization Strategy 3

complier is already doing the unrolling, this optimization is only important from the point of view of code rearrangement so that we can provide a little better locality. More importantly, it leads to the possibility of further optimization as depicted in Strategy 4 and 5.

### 4.1.4 Optimization Strategy 4: Use temporary variables to reduce memory-access with Strategy 3

After we manually unroll, we reduce the memory access by using temporary variables instead of writing to a particular location in array. After finishing the block, the temporary variable is written back to the array. This reduces memory motion and improves performance as less memory motion leads to fewer cache-line replacement and less cache misses, thereby achieving better cache performance. In Figure 4.4, we observed significant performance improvement over the compiler obtained speedups. The most important thing to note here is the modification is made to less than 50 lines of code out of thousands lines of code in Dendro. We achieved more than 71% speedup over the default Dendro `elMult` which results in more than 48% of overall speedup for 1000 points and around 58% overall speedup for 10K points per processor. We achieve 19X speedup in Solve time compared to the default Dendro version.

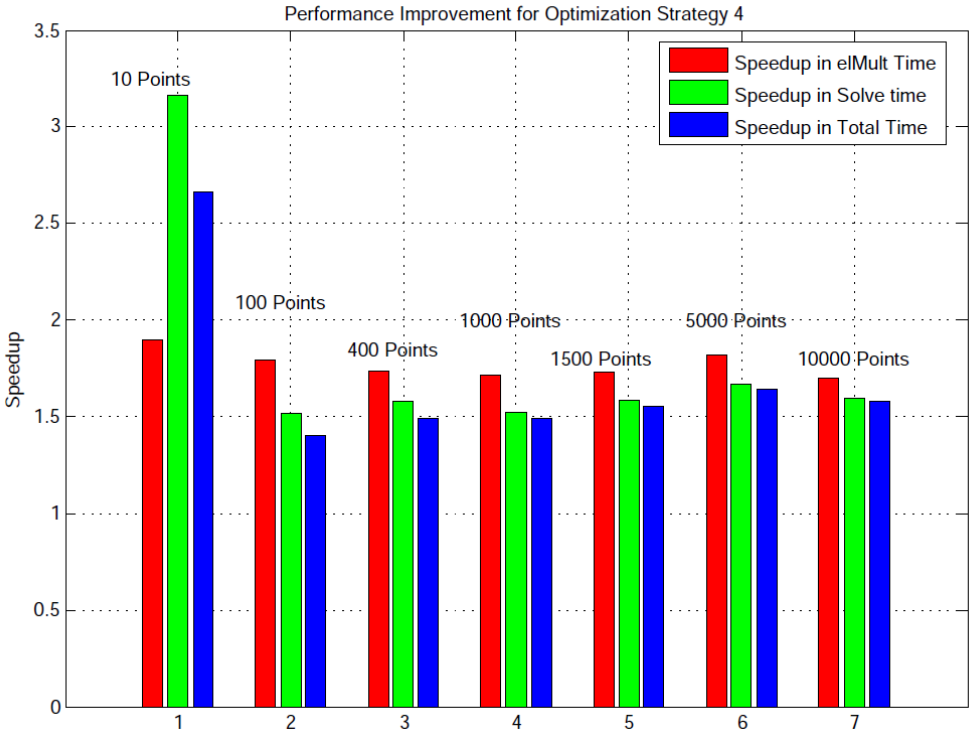


Figure 4.4: Performance Improvement by Adopting Optimization Strategy 4

### 4.1.5 Optimization Strategy 5:Explicitly prefetch arrays and bind it to the cache-line with Strategy 4

We have observed that compiler directed prefetching is not suitable for Dendro. However prefetching of arrays should improve performance. On top of all the optimizations, we analyzed the code to see when we should call the prefetch functions so that we can make sure that it is present exactly when it is needed. Although it is difficult to predict the exact times (or number of clock-cycles)at which to prefetch. The obvious question to ask here is

1. *When (and where) someone should call prefetching?*
2. *How to make sure that the prefetched cache-lines will not be removed before their use?*

#### When to call prefetching?

We have used the following instructions (supported in gcc 4.4.4) to prefetch the two array inArr (the array from which the values will be read) and the outArr (the array to which the values will be written to).

```
__builtin_prefetch (inArr,0, 3);
```

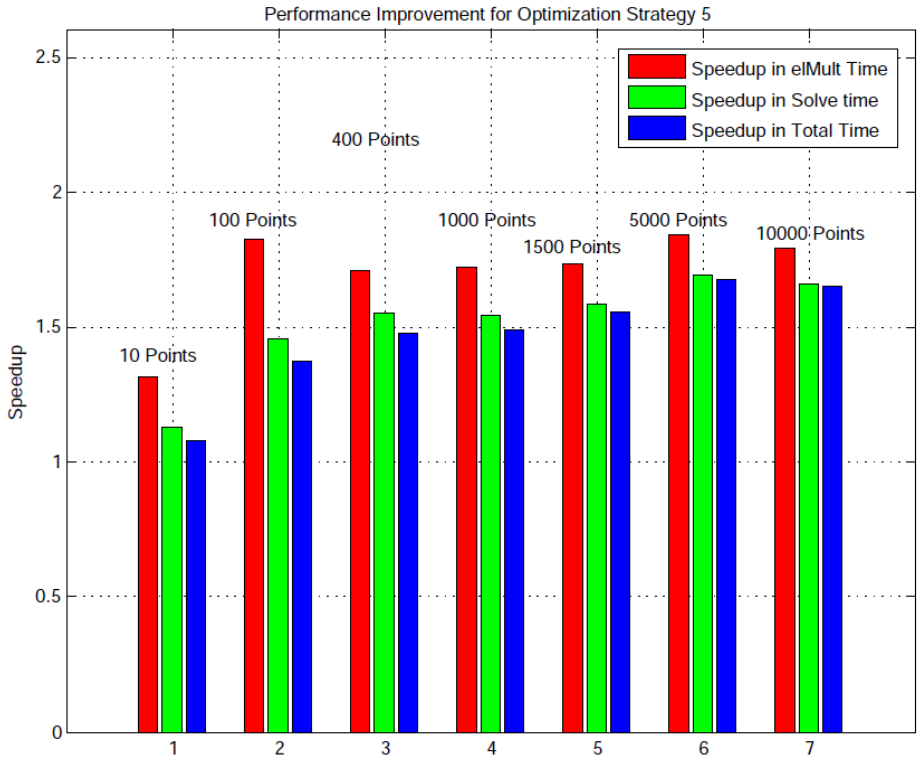


Figure 4.5: Performance Improvement by Adopting Optimization Strategy 5

```
__builtin_prefetch (outArr,1, 3);
```

The memory prefetch time is almost equal to the time required to fetch a particular block from memory to cache-line and is approximately 50 to 100 cycles. Thus, we call `__builtin_prefetch()` around 100 cycles before memory is used.

**How to preserve the prefetched data?**

`__builtin_prefetch()` is used to prefetch the inArr (for reading) and the outArr arrays (for writing to) from memory. We used 3 as the third parameter to `__builtin_prefetch()` to denote that the data has a high degree of temporal locality and should be left in all levels of cache if possible.

**Performance Improvement for Optimization Strategy 5**

The performance improvement has been plotted in Figure 4.5. We achieved 72% speedup relative to -O3 in `elMult` time and more than 48.8% speedup in Solve time for number of points = 1000. Also, we achieved more than 79% speedup in `elMult` time and more than 65% speedup in overall execution time for 10000 points. One crucial point to note here is that optimization strategy 5 provides better performance as compared to that achieved by optimization strategy 4. It will be important to compare the impact of the

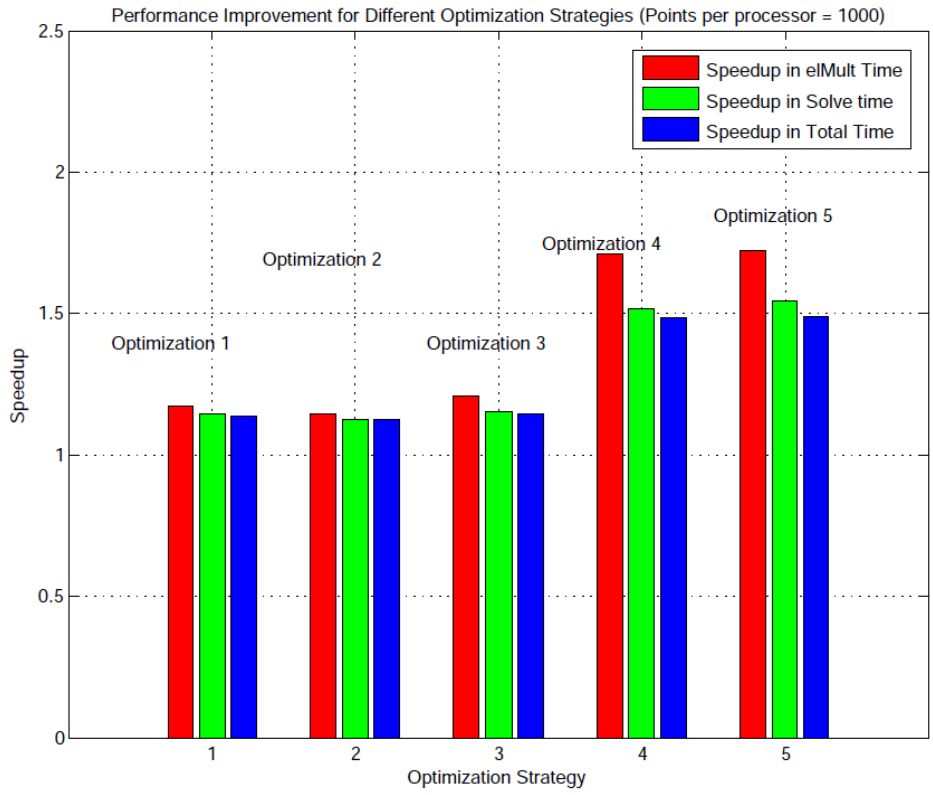


Figure 4.6: Comparing Performance Improvement by Different Optimization Strategies

different optimization strategies.

## 4.2 Comparing Different Optimization Strategies

In this section, we compared the performance improvement possible for number of local points to a processor equal to 1000. In Figure 4.6, we plotted the performance improvement data for the Dendro elasticity solver `newElasSolver`. We observed that it is possible (by strategy 5) to achieve more than 32% additional speedup over the best-possible compiler assisted speedup (as described in Optimization strategy 2). Using optimization strategy 5, we have achieved a 1.7236 times speedup for `elMult` for problem size which is 58% of the estimated available speedup as described in Section 3.4.

# Chapter 5

# Conclusions and Future Work

From the performance statistics, it was clear that the Solve part of the code takes most of the time (around 98%), out of which the matrix-vector multiplication takes around 95% of the total execution time. The Balancing and Points to octree creation takes insignificant time compared to Solve. By improving the time required for matrix-vector multiplication, the performance has been greatly improved. The discrepancy between theoretically achievable speedup and the speedup achieved by optimizing the code may be due to cache-behavior or communication patterns (use of blocking/non-blocking calls, collective communications) which impacts the performance. For number of local points per processor = 10000, optimization strategy 5 gives more than 79% speedup which contributes to an overall 65% speedup in the execution time. We achieved 58% of the estimated speedup for local number of points = 1000. To improve performance further and to achieve better performance for strong scalability, a more detailed analysis of the communication patterns of Dendro must be done. The analysis on computation and communication will help figure out the bottlenecks to achieve better scalability of the code. Incorporating parallel I/O improve the scalability of Dendro.

In Dendro, the octants of the octree needs to be sorted frequently. Improving the performance by adopting a new sorting technique may improve performance. For balancing the octree, an ordering based on Hilbert Curves is preferred to the Morton ordering [20]. So, performance can be improved by using Hilbert Curve ordering as it assures better the data-locality.

# Appendix A

## A.1 Software Release and Project Website

The default Dendro v-1.0 was compatible with petsc-2.3.3. We modified Dendro to make it compatible with petsc-3.x versions and the latest releases (say petsc-3.1-p2). The source code (Dendro-1.0.3.1) and patches have been posted on our website (https://netfiles.uiuc.edu/mukherj4/www/) in the Softwares (https://netfiles.uiuc.edu/mukherj4/www/software.html) section.

### A.1.1 Dendro-1.0.3.1

Users can download the source code od the updated package (Dendro-1.0.3.1) or you can download the patch for Dendro v1.0 (from the Dendro website http://www.cc.gatech.edu/csela/dendro/html/index.html) and apply the patches. For installing and using Dendro and **Petsc**, **MPICH 2** should be installed on your system.

| **Dendro Version** | **Download Url** | **Description** |
|---:|---|---|
| Dendro v1.0.3.1 | `$SOFTWARE/Dendro-1.0.3.1.tar.gz` | Compatible with PETSC-3.x |
| Patch for Dendro v1.0 | `$SOFTWARE/patch-Dendro-1.0.3.1.txt` | Using diff -Naru |
| Patch for Dendro v1.0 | `$SOFTWARE/patch2-Dendro-1.0.3.1.txt` | Using diff -rau |

**Note:** `$SOFTWARE = https://netfiles.uiuc.edu/mukherj4/www/softwares` as shown in the Table above.

Please contact us at mukherj4@illinois.edu or at wgropp@illinois.edu if you have any questions or comments.

## A.2 Experiment: Analyzing I/O Overheads

In Dendro, they use a serial code called `splitPoints` to split the points and store them in different files. They assign names for files make it ready for the parallel processing. The runScal code takes those file and

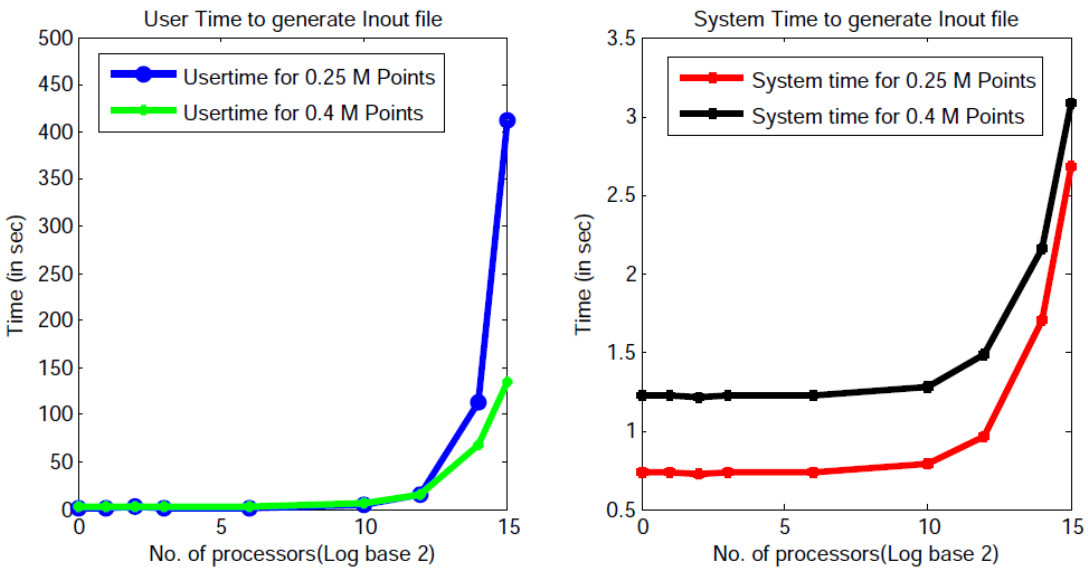


Figure A.1: Time to generate input files as the number of processors are getting increased

generate files with points required for the balancing operation on octrees. So, for doing a parallel balancing, the points need to be stored in files corresponding to the rank of each process. This provides scaling (strong) to the the Balancing and Points to Octree creation. From Figure A.1, it is evident that as the number of processors is getting increased, the time to generate the input files is also increasing. To see the timing requirements for number of processors less than or equal to 1024 ($2^{10}$), we have plotted Figure A.2 to focus on the time required to split files for lesser number of processors. From the experiments we found that for an input size per processor (number of local points) equal to 1000 and number of processors equal to 1024, the `splitPoints` takes around 5 seconds to read generate the (binary and text) files. As we have seen from the elasticity solver `newElasSolver` of Dendro takes around 77 seconds to solve and less than a second to do convert the points to octree and balance them. An I/O time of 5 seconds is a significant overhead. So, the weak-scalability will be hugely affected by the current I/O procedure. Not only, that, the files are not auto-removed after use. So, after every run one needs to clean them.

## A.3 Parallel I/O

In the Experiment A.2, the timing mentioned is for reading one binary file and writing to one binary and one text file per processor. Outputting the text file is not a part of the Dendro `splitPoints` program. Here, we compared our implementation with the Dendro provided `splitPoints` program. We used `MPI_File_open` to open the file by all the processes, then, provide an appropriate view (location in the file from which individual process has to read) by using `MPI_File_set_view` for all the processes. We used the collective

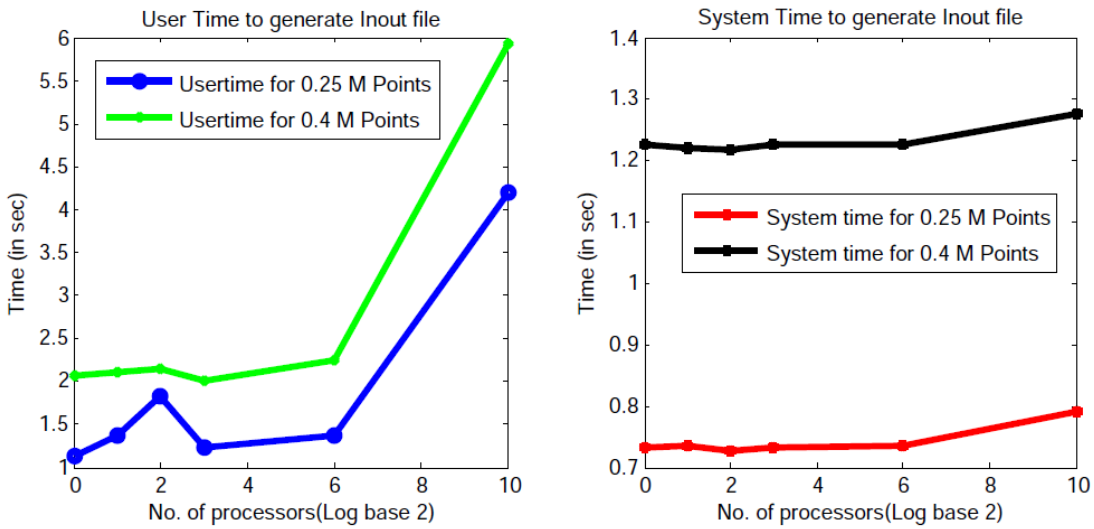


Figure A.2: System time and User time to generate input files for different process

read (`MPI_File_read_at_all`) to facilitate concurrent reading of the file. In Figure A.3, we compared the `splitPoints` program with a parallel I/O based application that uses two different approaches to provide atomic access to the file:

1. Using `MPI_File_set_atomicity()`

2. Using `MPI_File_sync()`

In Figure A.3, we compare the read-time for the parallel I/O based splitPoint with the total time (read and write time) of the serial application, because if someone is adopting parallel I/O, they do not need to write it to any files to be read by some process later. So, once it is read, the process is all set to proceed. Parallel I/O will not only save the time or improve scalability, but also, it helps to keep the disk clean by not creating so many files. Also, it helps to avoid ambiguity between different prefixes assigned to the generated file (by splitPoint) which will be later used by some process.

## A.4 Changes to make Dendro Compatible with PETSC 3.1

**Change the Makefile:**

```
include $PETSC_DIR/$PETSC_ARCH/conf/petscvariables
include $PETSC_DIR/conf/variables
```

Include 0 as the fourth parameter in the method `KSPSetConvergenceTest()` as shown below:

```
ierr = KSPSetConvergenceTest(damg[0]→ksp, KSPSkipConverged, PETSC_NULL,0);
```

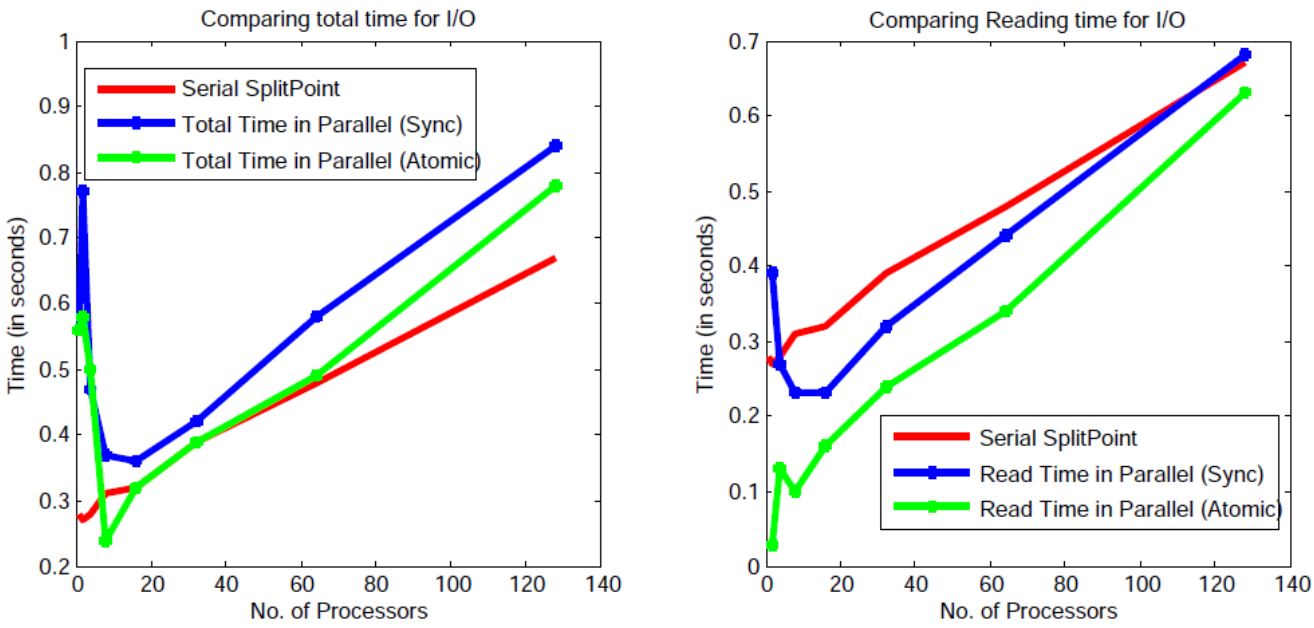


Figure A.3: Timing for Parallel I/O for Dendro

Positive-definite systems need to be taken care of as follows:

```
PCFactorSetShiftType(ipc,MAT_SHIFT_POSITIVE_DEFINITE)
```

In `$DENDRO_DIR/src/omg/omg.C`, we have modified the following functions

1. `PC_KSP_Shell_SetUp()`
   - For PETSC-2.3.3

     PetscErrorCode PC_KSP_Shell_SetUp(void* ctx)
   - For PETSC-3.1-p2

     PetscErrorCode PC_KSP_Shell_SetUp(PC pc)
2. `PC_KSP_Shell_Apply()`
   - For PETSC-2.3.3

     PetscErrorCode PC_KSP_Shell_Apply(void* ctx, Vec rhs, Vec sol)
   - For PETSC-3.1-p2

     PetscErrorCode PC_KSP_Shell_Apply(PC pc, Vec rhs, Vec sol)
3. `PC_KSP_Shell_Destroy()`

- For PETSC-2.3.3

  PetscErrorCode PC_KSP_Shell_Destroy(void* ctx)

- For PETSC-3.1-p2

  PetscErrorCode PC_KSP_Shell_Destroy(PC pc)

For all the above methods instead of passing context, get the context using `PCShellGetContext(pc,&ctx)` as shown below.

```
ierr = PCShellGetContext(pc,&ctx);
```

- In the following files

  `$DENDRO_DIR/examples/src/drivers/tstRipple.C`, `$DENDRO_DIR/examples/src/drivers/runScal.C`, `$DENDRO_DIR/examples/src/drivers/testConAndBal.C`, `$DENDRO_DIR/examples/src/drivers/rippleBal.C`, `$DENDRO_DIR/examples/src/drivers/testConAndBal.C` and `$DENDRO_DIR/examples/src/drivers/justBal.C`

made the some changes similar to the following

- For PETSC-2.3.3

```
int stages[5];
PetscLogStageRegister(&stages[0],"P2O.");
PetscLogStageRegister(&stages[1],"Bal");
PetscLogStageRegister(&stages[2],"Solve");
PetscLogStageRegister(&stages[3],"ODACreate");
PetscLogStageRegister(&stages[4],"MatVec");
```

- For PETSC-3.1-p2

```
PetscLogStage stages[5];
PetscLogStageRegister("P2O",&stages[0]);
PetscLogStageRegister("Bal", &stages[1]);
PetscLogStageRegister("Solve", &stages[2]);
PetscLogStageRegister("ODACreate", &stages[3]);
PetscLogStageRegister("MatVec", &stages[4]);
```

and included "petscsys.h" and "petsclog.h" to all the files mentioned above.

In `$DENDRO_DIR/examples/src/drivers/tstMatVec.C` we made the some changes similar to the following

- For PETSC-2.3.3

```
PetscLogEventRegister(&Jac1DiagEvent,
"ODAmatDiag",PETSC_VIEWER_COOKIE);
PetscLogEventRegister(&Jac1MultEvent,
"ODAmatMult",PETSC_VIEWER_COOKIE);
```

- For PETSC-3.1-p2

```
PetscLogEventRegister("ODAmatDiag",
PETSC_VIEWER_COOKIE, &Jac1DiagEvent);
PetscLogEventRegister("ODAmatMult",
PETSC_VIEWER_COOKIE, &Jac1MultEvent );
```

and included "petscsys.h" and "petsclog.h" instead of using petsc.h from petsc-2.3.3.

# References


[1] R. S. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, "Dendro manual," 2009. [Online]. Available: http://www.cc.gatech.edu/csela/dendro/Manual.pdf

[2] W. D. Gropp and E. Lusk, *Installation Guide for MPICH, a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996, aNL-96/5.

[3] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "Petsc users manual," Argonne National Laboratory, Tech. Rep. ANL-95/11 - Revision 3.0.0, 2008.

[4] "Image registration." [Online]. Available: http://en.wikipedia.org/wiki/Image_registration

[5] H. Sundar, R. S. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM J. Sci. Comput.*, vol. 30, no. 5, pp. 2675–2708, 2008.

[6] S. Balay, K. Buschelman, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Petsc tutorial: Numerical software libraries for the scalable solution of pdes," 2000. [Online]. Available: acts.nersc.gov/events/Workshop2000/slides/Gropp.pdf

[7] M. A. Marsan, G. Balbo, and G. Conte, *Performance models of multiprocessor systems*. Cambridge, MA, USA: MIT Press, 1987.

[8] M. K. Vernon, E. D. Lazowska, and J. Zahorjan, "An accurate and efficient performance analysis technique for multiprocessor snooping cache-consistency protocols," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 308–315, 1988.

[9] G. S. Brodal, "Cache-oblivious algorithms and data structures," in *Algorithm Theory - SWAT 2004*, ser. Lecture Notes in Computer Science, T. Hagerup and J. Katajainen, Eds. Springer Berlin / Heidelberg, 2004, vol. 3111, pp. 3–13.

[10] L. G. Brown, "A survey of image registration techniques," *ACM Computing Surveys*, vol. 24, pp. 325–376, 1992.

[11] H. Schumacher, S. Heldmann, E. Haber, and B. Fischer, "Iterative reconstruction of spect images using adaptive multi-level refinement," in *Bildverarbeitung fr die Medizin 2008*, ser. Informatik aktuell, W. Brauer, T. Tolxdorff, J. Braun, T. M. Deserno, A. Horsch, H. Handels, and H.-P. Meinzer, Eds. Springer Berlin Heidelberg, 2008, pp. 318–322.

[12] E. Haber, S. Heldmann, and J. Modersitzki, "An octree method for parametric image registration," 2006.

[13] E. Haber, S. Heldmann, and J. Modersitzki, "Adaptive mesh refinement for non-parametric image registration," 2007.

[14] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "Logp: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '93. New York, NY, USA: ACM, 1993, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/155332.155333

[15] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the logp model - one step closer towards a realistic model for parallel computation," 1995.

[16] T. Hoefler, T. Schneider, and A. Lumsdaine, "Loggp in theory and practice - an in-depth analysis of modern interconnection networks and benchmarking methods for collective operations," *Simulation Modelling Practice and Theory*, vol. 17, no. 9, pp. 1511 – 1521, 2009, advances in System Performance Modelling, Analysis and Enhancement.

[17] D. H. Bailey, R. Lucas, P. Hovland, B. Norris, K. Yelick, D. Gunter, B. de Supinski, D. Quinlan, P. Worley, J. Vetter, P. Roth, J. Mellor-Crummey, A. Snavely, J. Hollingsworth, D. Reed, R. Fowler, Y. Zhang, M. Hall, J. Chame, J. Dongarra, and S. Moore, "Performance engineering: Understanding and improving the performance of large-scale codes," *CT Watch Quarterly*, vol. 3, no. 4, pp. 18–23, 2007.

[18] H. Shan, E. Strohmaier, J. Qiang, D. H. Bailey, and K. Yelick, "Performance modeling and optimization of a high energy colliding beam simulation code," *Proceedings of SC2006*, vol. LBNL-60180, Nov 2006.

[19] J. Mukherjee and S. Raha, "Power-aware speed-up for multithreaded numerical linear algebraic solvers on chip multicore processors," *Scalable Computing: Practice and Experience*, vol. 10, no. 2, pp. 217–228, 2009.

[20] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, "Dynamic octree load balancing using space-filling curves," Williams College Department of Computer Science, Tech. Rep. CS-03-01, 2003.