Performance Evaluation and Enhancement of Dendro

Jayanta Mukherjee* and William D. Gropp†
Department of Computer Science, University of Illinois at Urbana-Champaign
201 North Goodwin Avenue, Urbana, IL 61801-2302, USA
Email: {*mukherj4, †wgropp}@illinois.edu

***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.**

I. Introduction

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.

A. Dendro Structure

The objective of this work is to figure out the performance bottlenecks that affects the speedup and scalability of the applications. 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.

B. Organization of Dendro: Functional Blocks

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 [4].
4) **Solve** : This is the actual solver which calculates the displacements for the non-hanging nodes using matrix-free method [5].

II. 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.

A. 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 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

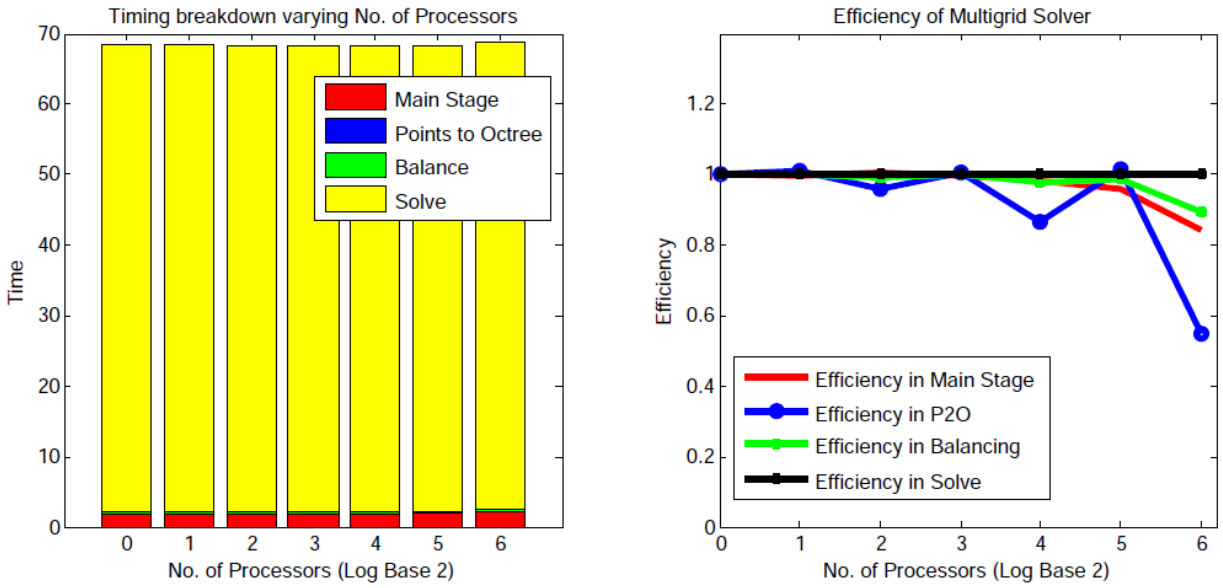

Fig. 1. Scalability Study of the Multigrid Solver of Dendro

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 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 VII-A of the Appendix.

B. Experiment:2: Performance Statistics of Dendro

We have used the PETSC profiler output to analyze the performance statistics. By looking at Figure 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 2.

a) : 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, 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**

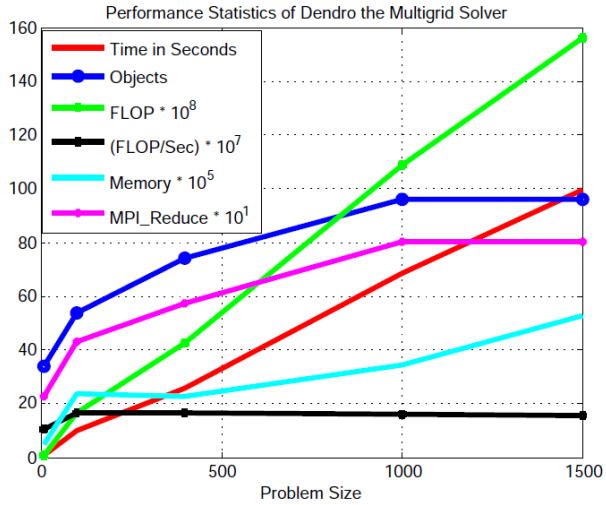

Fig. 2. Performance Statistics of the Multigrid Solver of Dendro

- MatMult
- elMultFinest

2) Vector functions
- VecDot
- VecAXPY
- VecAYPX

3) Krylov Solver
- KSPSolve
- PCApply

b) : We have plotted the Performance Statistics for Matrix Multiplication routines in Figure 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

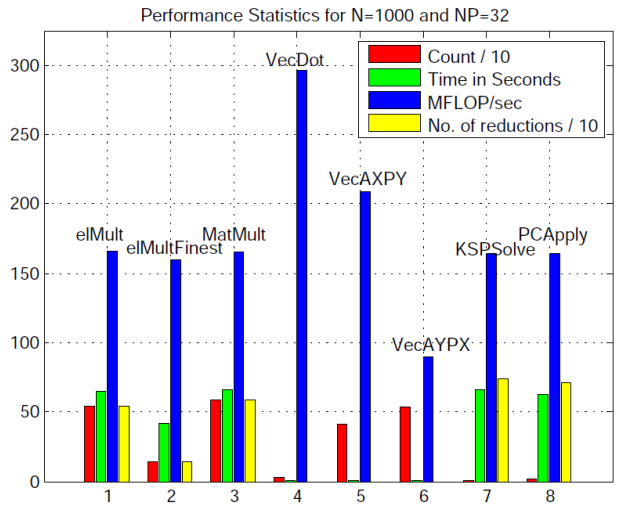

Fig. 3. Breakdown of the timing and performance of the Multigrid Solver of Dendro

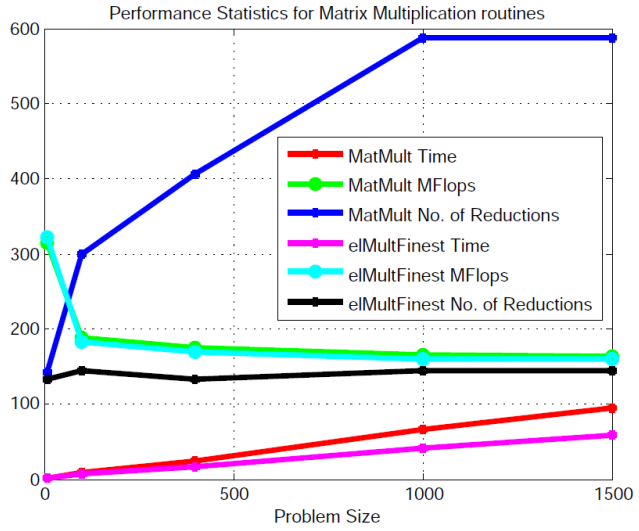

Fig. 4. Performance Statistics for Matrix functions

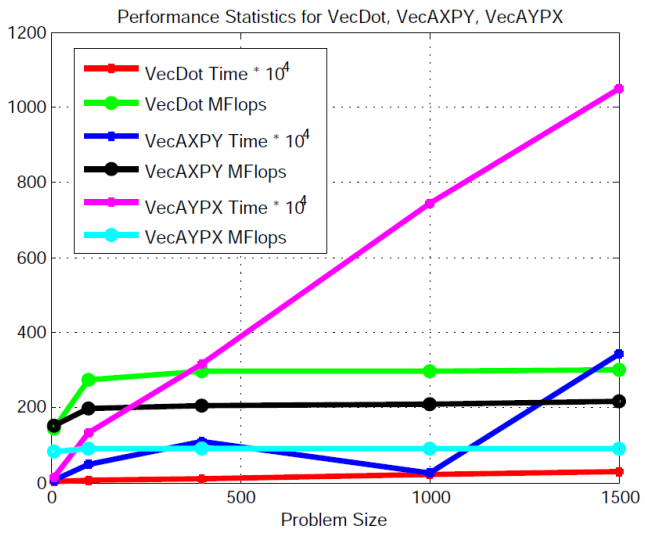

Fig. 5. Performance Statistics for Vector functions

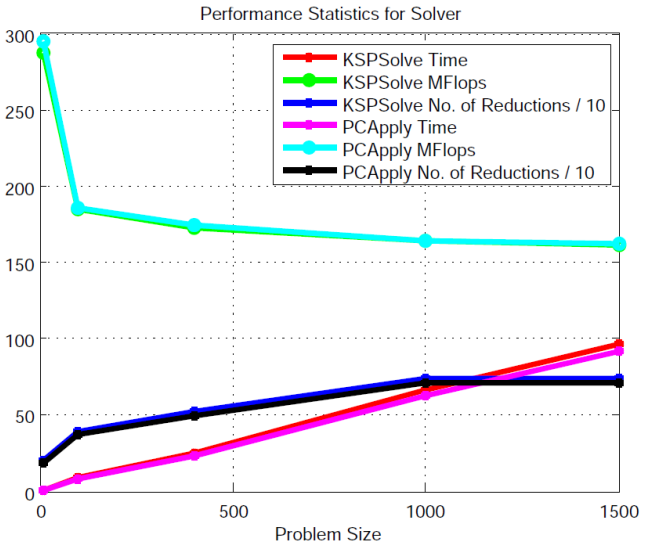

Fig. 6. Performance Statistics for Krylov Solver

`elMultFinest` is a part of the `MatMult` operation being performed. In Figure 5, the performance statistics for Vector operations has been performed with the required scaling to fit the data in the plot.

Figure 6 contains the performance statistics for the `KSPSolve` and `PCApply` routine.

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

C. 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 7, the number of times the Creations/Destructions for Matrix (`Mat`) has been performed with the required memory for performing that.

c) : We have plotted in Figure 8, the number of times the Creations/Destructions for Vector (`Vec`) has been performed with the required memory for performing that.

d) : From the Figure 8, it is shown that as expected, the memory requirement for performing Vec grows linearly with the problem-size. In Figure 9, we have plotted the count and memory requirement for performing the Krylov Solve.

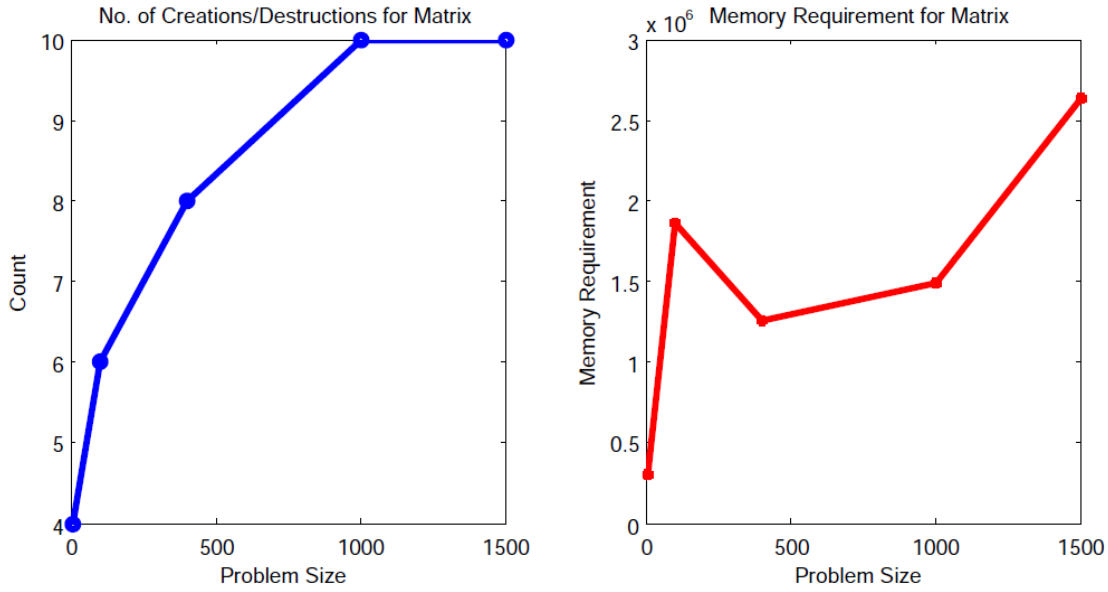

Fig. 7. Memory Requirements for performing Mat

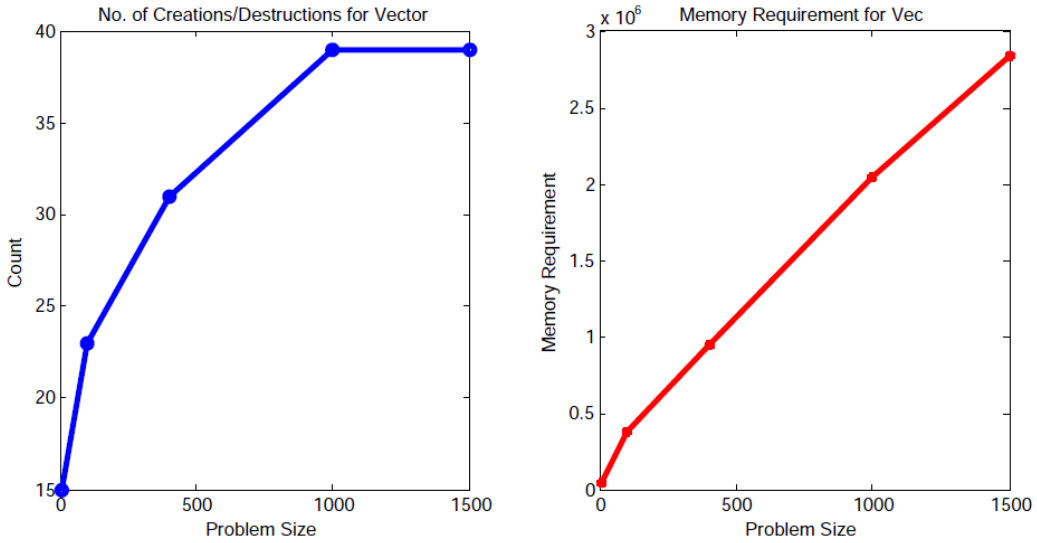

Fig. 8. Memory Requirements for performing Vec

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

III. 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

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

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

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

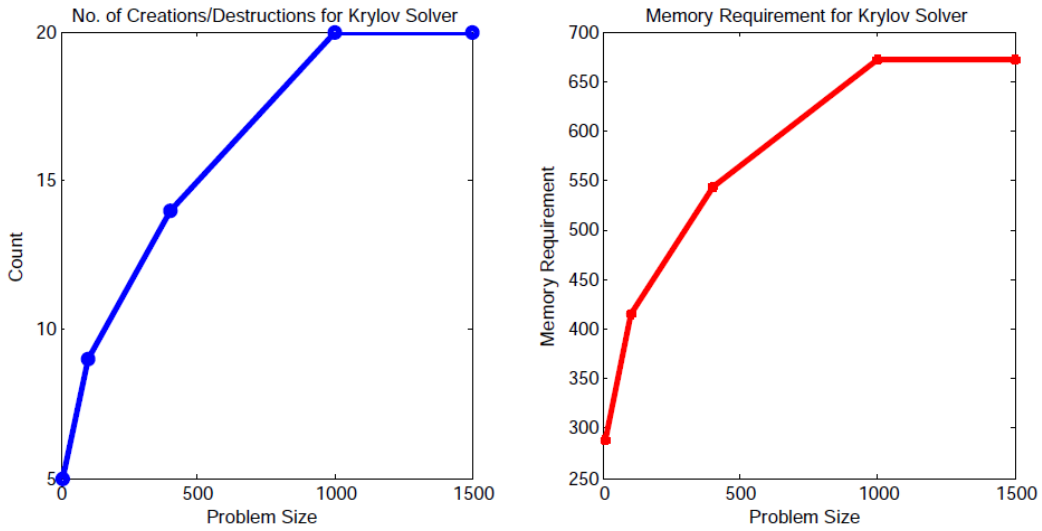

Fig. 9. Memory Requirements for performing Krylov Solver

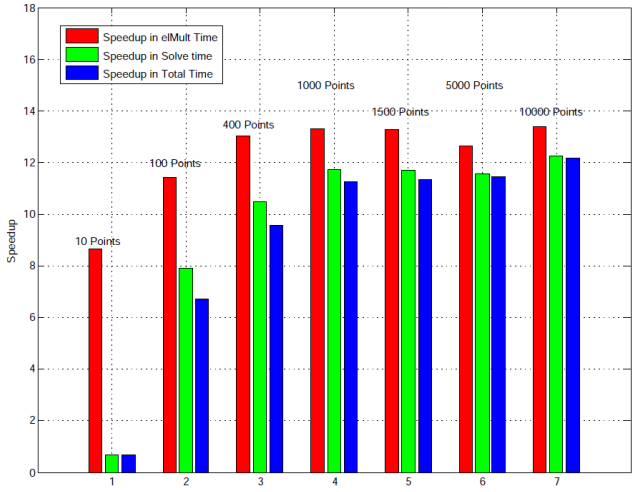

Fig. 10. Performance Improvement by using -O3 over default Dendro Elasticity Solver (`newElasSolver`)

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 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.

IV. Performance Improvements

Now from the above analysis 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 IV-E1.

From, Experiment VII-A, 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 VII-B, we have explained the details about the performance improvements possible with relative merit and

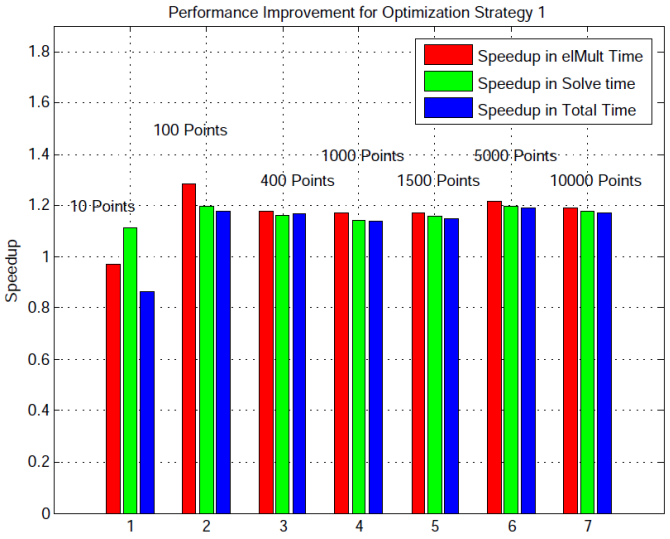

Fig. 11. Performance Improvement by Adopting Optimization Strategy 1

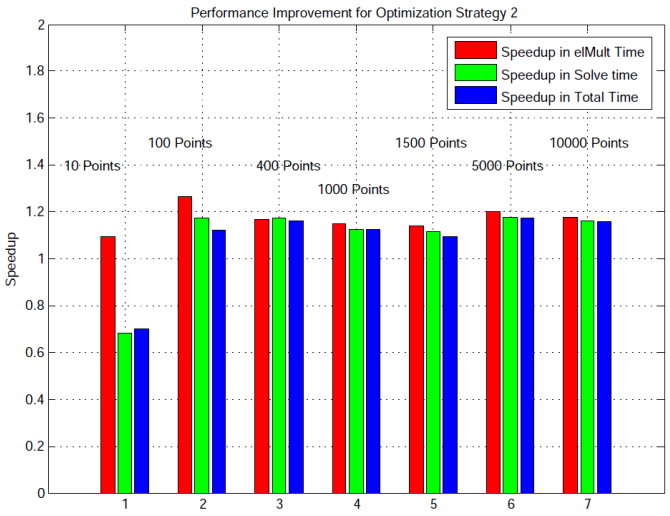

Fig. 12. Performance Improvement by Adopting Optimization Strategy 2

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 VII-C.

A. 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 11, 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.

B. 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 the `-fprefetch-loop-arrays` flag to achieve better performance. In Figure 12, 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.

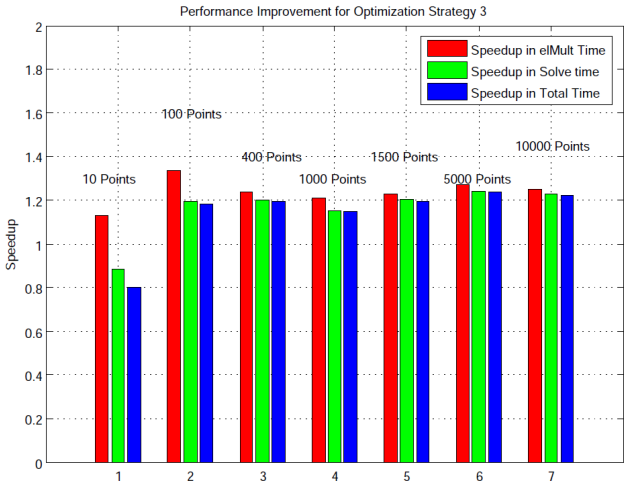

Fig. 13. Performance Improvement by Adopting Optimization Strategy 3

C. 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 13, 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 complier is already doing the unrolling, this optimization is only important from the point of view of code rearrangement

Fig. 14. Performance Improvement by Adopting Optimization Strategy 4

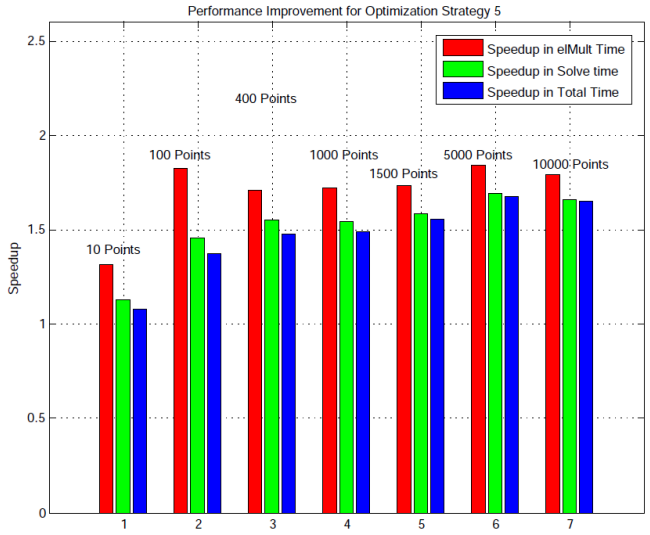

Fig. 15. Performance Improvement by Adopting Optimization Strategy 5

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.

D. 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 14, 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.

E. 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?*

1) 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);
__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.

2) 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.

3) Performance Improvement for Optimization Strategy 5: The performance improvement has been plotted in Figure 15. 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 different optimization strategies.

F. 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 16, we plotted the performance improvement data for the Dendro elasticity solver `newElasSolver`. We observed that it is possible (by strategy 5) to achieve more

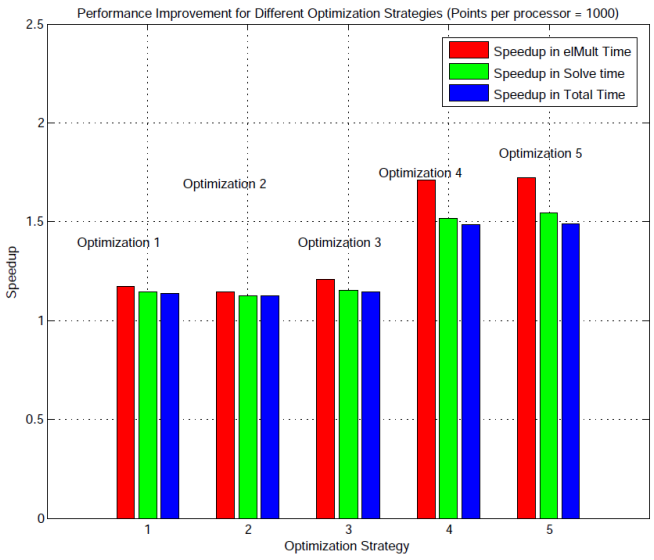

Fig. 16. Comparing Performance Improvement by Different Optimization Strategies

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 III.

V. 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 will definitely help to scale the application. 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 [6]. So, performance can be improved by using Hilbert Curve ordering as it assures better the data-locality.

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] 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.

[5] 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

[6] 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.

VI. Acknowledgements

The project is funded by NSF Grant Number is 0849301. We would like to thank the developers of Dendro Rahul Sampath and Hari Sundar who helped us during the performance study by their constructive advices and suggestions. We would also like to thank the PETSC developers and support group for their valuable feedbacks. We would like to thank Michael Campbell and the Turing Support-team for their help.

VII. Appendix

A. 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 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 17, 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 18 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.

B. Parallel I/O

In the Experiment VII-A, 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 read (`MPI_File_read_at_all`) to facilitate concurrent reading of the file. In Figure 19, 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 19, 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.

C. 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);
```

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 `$DENDRO_DIR/examples/src/drivers/tstRipple.C`, `$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.

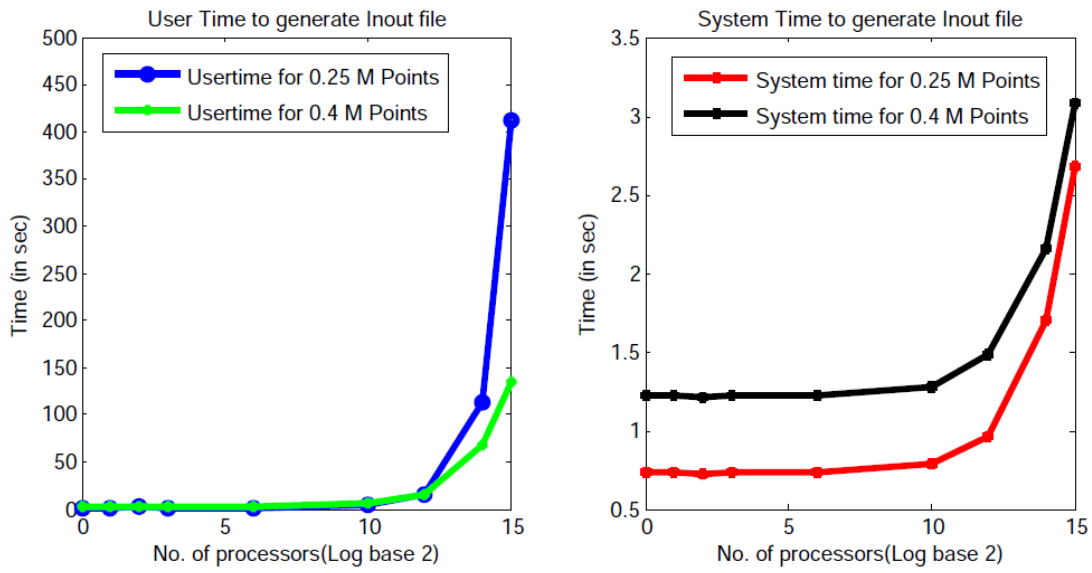

Fig. 17. Time to generate input files as the number of processors are getting increased

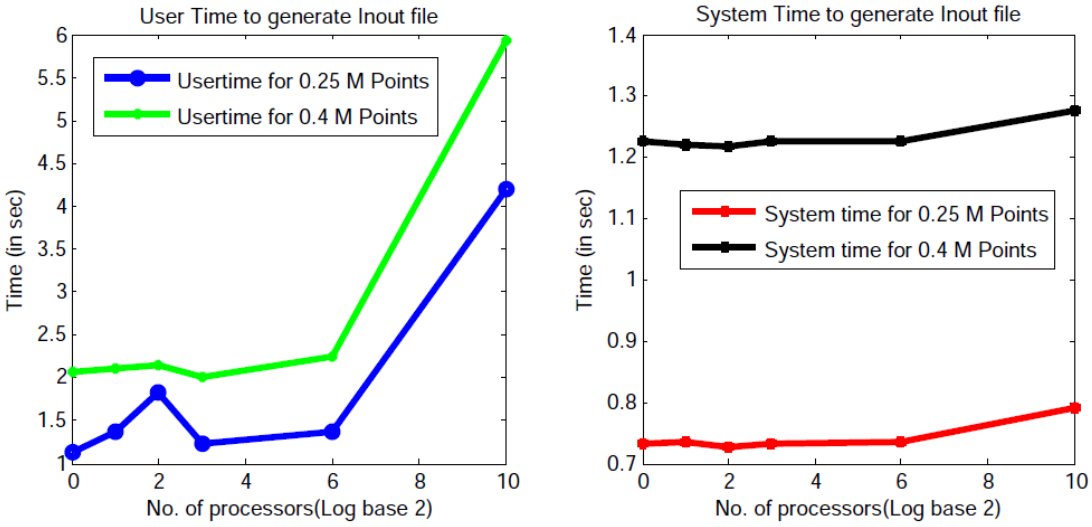

Fig. 18. System time and User time to generate input files for different process

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.

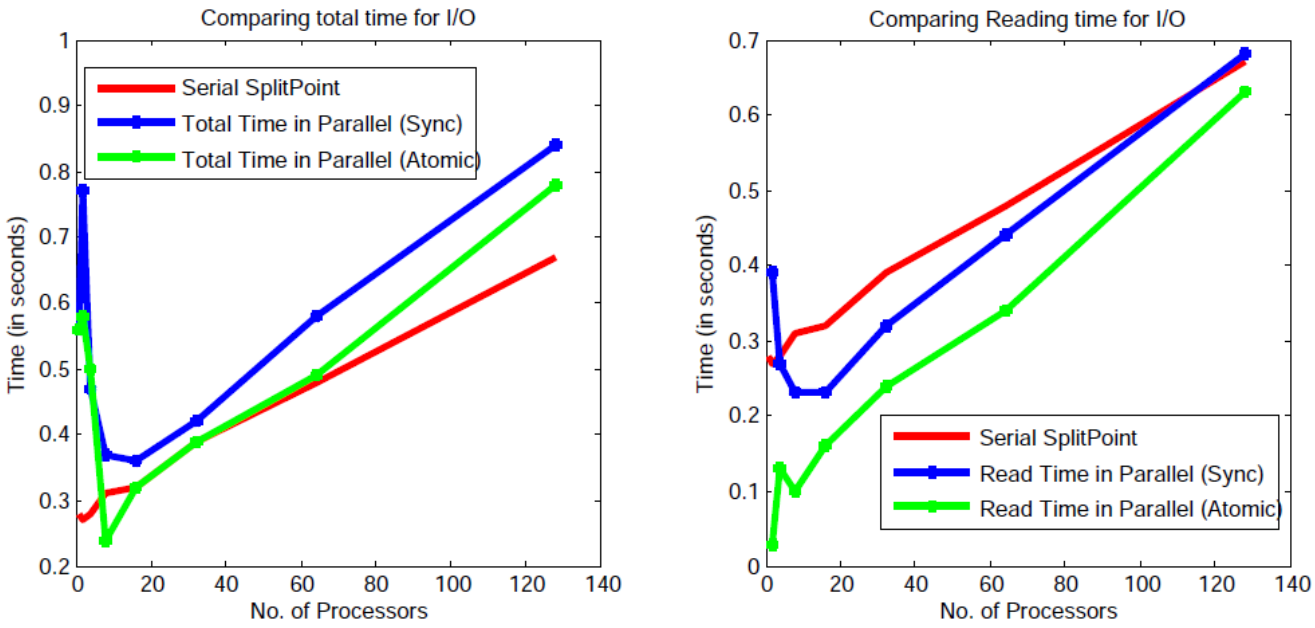

Fig. 19. Timing for Parallel I/O for Dendro