Parallelization of Dynamic Workload Prediction tool for Trace Driven Simulation

Kaustubh Shukla
Electrical and Computer Engineering
University of Florida
kaustubh.shukla@ufl.edu

Devansh Gulati
Electrical and Computer Engineering
University of Florida
d.gulati@ufl.edu

Kshitij Raj
Electrical and Computer Engineering
University of Florida
kshitijraj@ufl.edu

***Abstract*—This project implements a parallel version of a Dynamic Workload prediction tool which is used for trace driven simulations. Trace driven simulations help in analyzing workloads and therefore it is important to parallelize it. The sequential implementation of this tool does not scale well when this process is expanded for higher workloads to get an accurate trace. In order to efficiently analyze which parallel programming paradigm is suitable for implementing the parallel model of this tool through task distribution, we have implemented the parallel implementation of this tool in MPI, MPI+OpenMP and MPI+CUDA. The input files for this tool implementation are- a rank csv file consisting of all the element entries and another csv file consisting the location of each particle corresponding to each element in the whole rank space. MPI programming model acts as a root for workload distribution among different processors. Since the Dynamic workload Prediction tool is an embarrassingly parallel application, MPI effectively divides the input file across processors and acts as a good medium for parsing the input files and extracting the required rank to particle count and particle movement trace csv file. The results obtained in MPI are found to provide a great reduction in execution times over 32 processors and 50 files. This result is even further improved by using MPI+OpenMP with 4 OpenMP threads for each node. MPI+Cuda on the other hand is found to be not as effective owing to its huge communication overhead which takes up the significant portion of the execution time. This project is implemented on University of Florida's HiperGator network.**

Keywords*—*Dynamic Workload, HiperGator, Hybrid MPI, Trace Driven Simulation

I. Introduction

Dynamic workload prediction tool is used with trace-driven simulations. Trace driven simulation is important because it helps us analyze the workload based on the basis of particle count. Workload per processor depends on the number of particles and if the particles per processor is dynamic then the particle flow information varies among processor and varies at each timestep. Therefore, it is important for us to extract trace of the particles. The goal is to parallelize workload prediction tool on multiple programming paradigms for different machine and applications configurations. A trace data for a specific problem can be given as input along with applications and machine processor count parameters. The basic assumption is that we can generate synthetic traces for larger problem sizes on the basis of the original traces generated by the movement of particles in smaller problem sizes.

Given the serial implementation of the problem, this project aims to implement the tool using parallel programming models like MPI, OpenMP and CUDA so that the particle workload predictor will see dramatic speedup in simulating the problem. Using trace mapping and element to rank input files, we are able to track the particle movement across the processors at a particular timestep. In addition to tracking the particle count, we also track the communication workload by tracking the particle movement across processors at a given timestep. This is achieved by analyzing particle-element mapping. The first input file is a particle trace csv which is of the following format: *gpid1, gpid2, gpid3, gEid, Px, Py, Pz, Ts* where gpid is the unique combination of a particle id triplet, gEid is the element id which corresponds to the element the particle is in while Px,Py,Pz correspond to the particle cordinates followed by the timestep at this trace was observed. The second input file is an Element to Rank mapping csv which is of the following format: *Rank,gEid1,gEid2,gEid3,.....gEidN*. This file signifies that a set of element ids or elements resides in a "Rank". Therefore the hieriarchy of the problem can be analyzed as a tile of ranks which consists of a set of elements which further consists of particles. This is shown in Fig. 1. From these two input files, the following output csv files are extracted:
Rank, no. of particles–File 1
Source Rank, Destination Rank, no. of particles, Ts– File 2
Here Source Rank and Destination rank refer to the source and destination rank ids when a particle moves between ranks, the Ts refers to the timestep of analysis and the no of particles refer to the number of particles that have moved. Therefore this tool generates a particle trace for each timestep and gives an insight into particle count and particle movement.
The need for a parallel implementation of this tool arises from the fact that this whole trace file production process is embarrassingly parallel, therefore we can achieve significant speedup. The need for trace driven simulations is increasing rapidly as the complexity and performance of computer systems increase. Parallelizing the particle workload tool will decrease the time it takes a serial code to generate the same set of results and help developers design efficient algorithms. The following section of this article are as follows. Section II describes related works in this field. Section III describes our program flow and methodologies used in parallelizing the workload prediction tool. Section IV describes the results obtained for various methodologies while Section V concludes the paper.

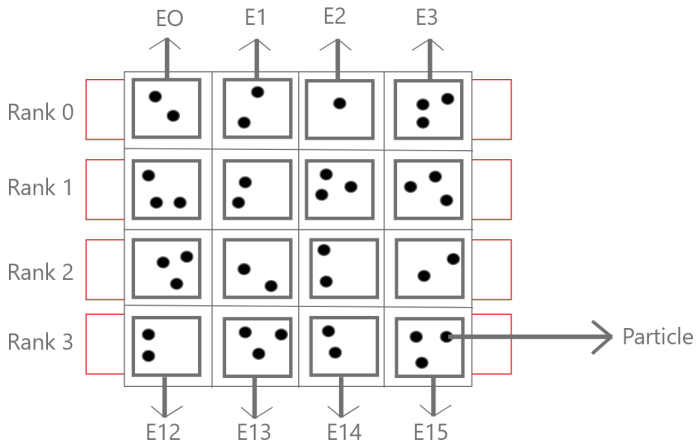

Fig. 1: Rank, Element, Particle Hierarchy

II. Related Works

This project is based on [1]. The tool does its implementation in a serial manner while we have extended it to parallel implementation. There has been a lot of work done in relation to particle in a cell algorithm and their implementations using high performance computing. With the emergence of new architectures every few years application developers are often hesitant in developing codes with the fear of their code becoming archaic after the development of new architectures. To cope with these changes Victor.K.Decyk et.al [2] have come up with a set of new adaptable particle in cell algorithms that gives assurance to developers by finding some key features in the development of these new aforementioned architectures. The first observation made was that the architectures in future will generally be hierarchical with the lowest level of hierarchy being dominated by some tightly coupled SIMD cores which will be used for evolution in lock step, the next higher level of hierarchy will be that of loosely coupled SIMD cores with slower memory and final the highest level of hierarchy will be that accumulation of these lower levels with message passing. The adaptive algorithm generally deals with plasma physics applications which involves three points, 1) The deposition of particles on a grid, 2) Solving a differential equation to find the electric and magnetic fields on the grid, 3) Finding out the acceleration of particles on the grid by solving newtons equations. Traditionally the codes have been made parallel by keeping particles and their interacting fields in the same nodes however only coarse-grained decomposition could be achieved by this. Paulett.C.Lewier [3] of the California Institute of Technology also came up with general concurrent algorithm of a particle for a particle (GCPIC) and has been the golden standard in cell simulation code. In this algorithm the physical domain of the particle in simulation was decomposed into a set of sub domains and these sub domains were then distributed among the different nodes on a machine. The most critical bottleneck of the codes involving the upgradation of the locations and the speeds of the particles were hundred percent efficient when increased with the increase in then number of cores used for the computation of the resources showing an extremely strong scaling. The push times and the times to deposit forming the most intensive areas of operation were also seen to evolve at a 94-97 percent efficiency. The authors of [4] carry out these modifications done on Plasma physics particle in cell algorithms. The simulations were carried out on a JPL Mark III Hybercube parallel computer system. Dynamic workload prediction is also key part of our project and finds many applications especially in the development of multistaged multi core chips. In the case of dynamic workload prediction several advances have been made to allow for the development of low powered, highly reliable chips as theorized by Monir et.al [5] . The approaches to the workload prediction have been historically reactive or proactive. The approach of an SVM (Support Vector Machine) which instead of reducing the training error minimizes the generalized performance has been adapted to workload prediction problem and utilized to Dynamic Voltage scaling.

III. Programming Methodologies

Our implementation of the parallel version of the original JAVA based tool is spanned across four programming methodologies namely, Sequential C++, MPI, MPI+OpenMP and MPI+Cuda. This section explains the flow of how these methodologies parallelize the tool and the next section analyzes the results obtained.

A. *Sequential-C++*

Sequential C++ can be considered as an MPI with just one processor. On surface the translation of JAVA code to C++ seems trivial, however we have introduced multiple data structure optimizations in our C++ implementation to make it more efficient. The C++ implementation involves two key functions- Evaluating and populating a data structure that stores Ranks corresponding to every Element id (gEid) as obtained from rank to element csv file and correlate that Rank to the gpid set in the second csv file with geID as the key. This leads to the production of two output files, namely the particle count for each rank and the movement trace of particles across ranks. Both of these files are obtained per timestep. A naive way of doing this would be to store the contents of the rank to element csv file directly into an array, however this whole procedure would lead to a quadratic time complexity (Number of ranks * Number of elements) when we try to parse this for generating the required output files. In our implementation, we have used a dictionary to implement the intermediate data structure that holds the rank and element information with geID as the key and rank as the value. This drops down the parsing time complexity to O(1) which is a significant speedup. A second optimisation is to implement a multi key multi value Hash table. This hash table consists of old rank, new rank as the key pair and movement-count,timestep as value pair. These values i.e old rank, new rank, particle movement count and timestep are also written in a csv file, which constitutes the movement trace. We added this intermediate hash table stage instead of directly writing into a file to make this code readily optimised if we want to add another stage of functionality that utilizes the movement trace as input. However this optimisation is severely limited by the number of file rows and parsing its columns which leads to a quadratic time complexity. Fortunately this is where parallel programming paradigms like MPI and hybrid MPI come into picture.

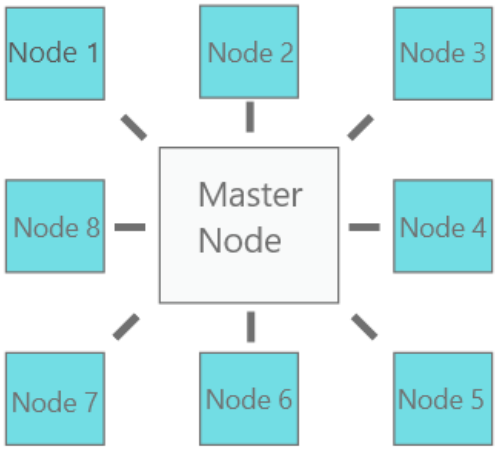

Fig. 2: MPI nodes

B. C++ MPI

Since this tool is an embarrassingly parallel application, there is great potential in parallelisation. We have achieved this by using MPI with C++ to distribute both the files and computation workload across processors. A master rank sends file information to worker ranks so that they can read the relevant file content from the memory. It is assumed that MPI ranks have all the files in a common shared memory. Therefore a memory overhead is attached with this implementation, however considering the workload size, this overhead is bound to be paid. Along with sending the file names, offsets of the shared file name array, and the number of files to be executed in each rank, the master rank is also involved in computation. Once the ranks have read the distributed and divided file contents, they work as usual to concurrently parse the file to evaluate the particle count and movement trace. Time constructs are also used to evaluate the time taken for the complete movement trace. The time complexity for each processor is reduced by N where N is number of ranks (ideally). So it goes down from O(M*8) to O((M*8)/N) where M*8 represents the M no of lines in the csv file with 8 entries each as mentioned in Section I. The distribution of workload in C++ MPI is illustrated in Fig 2. The example considers 8 nodes with the master nodes allocating tasks to the worker nodes.

C. Hybrid MPI

MPI in itself is very powerful to distribute the workload across processors and perform concurrent computations which lead to significant speedups. However to analyze the potential advantages of Hybrid MPIs, we have implemented the serial version of the code in two Hybrid MPI methodologies, namely MPI+OpenMP and MPI+CUDA. MPI+OpenMP is the multi-threaded flavour of MPI. There are two potentially concurrent regions in the serial code, the first one being the stage where the dictionary for rank to element relation with element as key and rank as value is evaluated and the second where this dictionary is used in conjunction with the parsed input file. We have used OpenMP threads to evaluate the Rank dictionary for each processor. MPI+OpenMP node layout is illustrated in Fig 3. We have extensively analyzed the results for the threads in the next section.

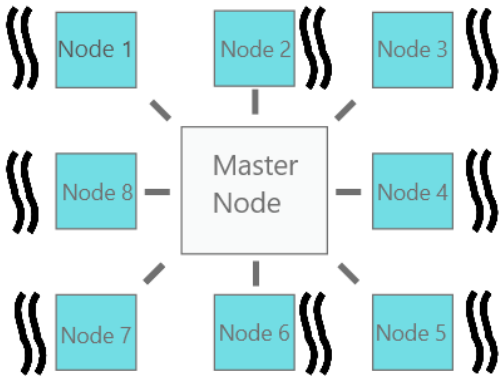

Fig. 3: OpenMPI nodes

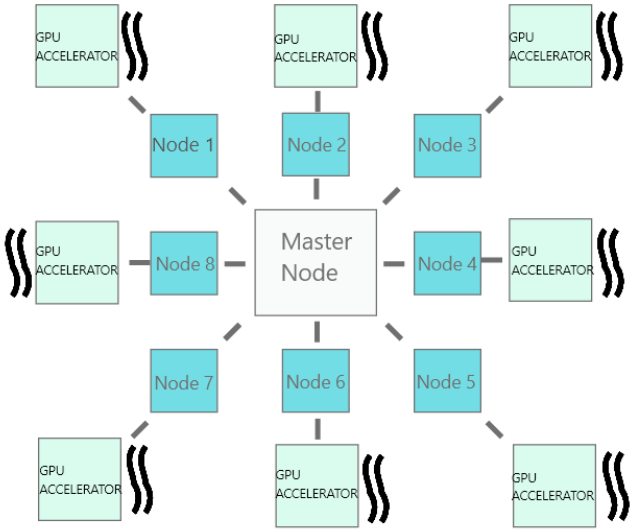

Fig. 4: MPI+CUDA nodes

MPI+CUDA is used for the same section of the code i.e to accelerate the Element, Rank dictionary creation. However as opposed to OpenMP, CUDA will have significant communication overheads. The file transfer between device and host would have the larger share for execution time than the computation in the CUDA kernel, as this tool isn't computationally intensive. Fig 4 showcases the MPI+CUDA layout. In both the figures, the black colored strings represent threads.

IV. Results and Analysis

The target of this paper is to showcase the techniques of parallelizing the Dynamic Workload prediction tool. We are provided two input files in the csv format, A particle id trace and an element id to rank relation, as specified in Section I. Our target was to decompose the functionality of the prespecified JAVA workload prediction tool into multiple

chunks/units, that can be implemented concurrently to achieve speedup. As mentioned in Section III, we have implemented this task of parallel computing by implementing the following programming paradigms- MPI, MPI+OpenMP, MPI+CUDA. Standalone implementations of OpenMP and CUDA were also implemented for a target function so that it can be smoothly expanded to the MPI paradigm. The target function had two possible candidates- The Element,Rank Dictionary function or the movement detection function. Due to the file I/O overhead of the latter, Element,Rank Dictionary function was chosen to run over multiple threads in the OpenMP environment. Since this Element,Rank dictionary is involved in over 65k iterations and was the part of every MPI processor, an analysis of the concurrent multithreaded execution of this section is necessary. A similar rhetoric is used for acceleration of this Hash function using CUDA. The hash function is deployed on a GPU as a kernel for each host node. On surface this may seem advantageous but it incurs huge communication costs, as will be seen later in this section. All of these different methodologies were implemented on the University of Florida HiperGator provided by UFRC [6]. The detailed analysis of the results obtained by each programming paradigm is illustrated in following sections.

A. *Serial C++*

A serial implementation of the tool in JAVA was already provided to us. The JAVA tool however was not compatible to be evolved into a parallel programming model. Therefore the JAVA code was translated into C++ code. As explained in Section III, numerous optimisations were made in the C++ code to reduce the time complexity. One such optimisation was to create a dictionary for the Element,Rank key value pair. This reduced the searching time complexity to O(1) from the original complexity of O(N*e) where N is the no of ranks and e is the no of elements in each rank. Fig 5 showcases the values obtained after scaling the problem size i.e the input file size vs the execution time. As expected for a serial implementation with no file decomposition, the execution time should linearly increase with an increase in problem size. However thats the ideal case, in a practical scenario, there are a lot of factors that account in the overall execution time as the problem size increases. For instance in our Fig 5 graph, the increase in execution time is expected till Input=46, however there is a dramatic shootup in the execution time for Input=50. This could be because of many reasons. One possible reason can be that, for Input size 50, it reaches its maximum cache capacity and it has to carry out cache block eviction or page replacement (in case of TLB), therefore we see the rise in execution time. The other possible reason could be that the HiperGator memory resources were heavily congested which led to this delay. Overall our analysis concludes that memory related overheads have caused this shootup.

B. *MPI*

As mentioned in the previous section, the serial model doesn't scale very well for a larger problem size as seen by the shootup for 50 input size. Since this is an embarrassingly parallel application, it would be very inefficient for us to not exploit this property and stick by the sequential model. This is the entire motivation behind this paper, for applications that

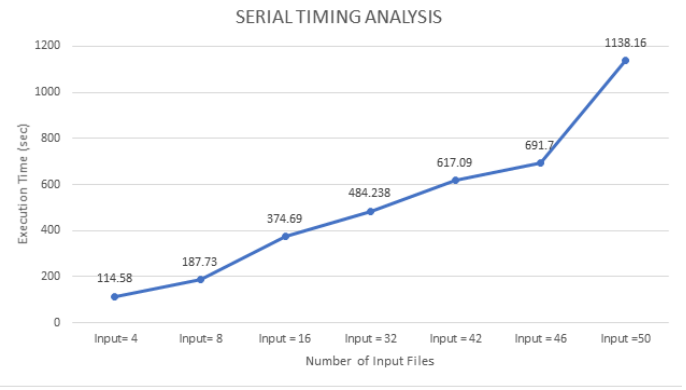

Fig. 5: Serial execution graph

are embarrassingly parallel, a concurrent implementation of the problem that distributes the workload across processors is necessary. In our MPI implementation we have provided the results as a graph in Fig 6. This graph shows a comprehensive plot of execution time vs problem size for varied amount of processors. It can be clearly observed that for processors 2 and 4, the execution time drops for each input size as compared to a single processor. This is owing to the input size distribution and concurrent execution of data. The same amount of work gets done in the lesser amount of time. Other processors also work as expected, with decreased execution time. An interesting case is observed for 8 processors, there is dramatic drop in the execution time for for file size 46. This again could be attributed to internal compiler optimisations in terms of cache/page replacements to ensure spatial and temporal locality of required data blocks. The parsing/evaluations involved for file size 46 were all localized spatially/temporally which lead to fall in execution time. The sharp decrease in congestion for memory/resource utilization over the Hipergator server is also the reason for this dip in execution time. An interesting observation that can be made here is that for both Sequential and MPI graphs, we observe blips in data, a sharp rise in the former while a sharp drop in the latter, both of them working at the extreme ends of their processing spectrum. These blips can often be attributed to the effects of interactions with organizational parameters of extended memory like hierarchy that affect artifactual communication and performance.

The results obtained after MPI processing are also analyzed for strong scaling and weak scaling. Strong scaling is plot in Fig 7 where the number of processors are increased for problem size of 50, i.e the maximum file size. As is theorized, the execution time falls for increase in processor since the same amount of work, lets say N is shared across P processors. Thus each processor does N/P of the work and therefore executes it at an earlier time as we keep increasing P (Work for each processor decreases). However in a realistic case the work distributed across processors is not always uniform. Therefore we see a non uniform descent in the graph instead of a linear descent. It is also observed that this descent is less prominent as we reach the upper limits of processors because of Amdahl's law. The upper limit of speed up or the lower limit of execution time is bounded by the serial part of the problem. The weak scaling of the graph is also shown in Fig 8 and analyzed for

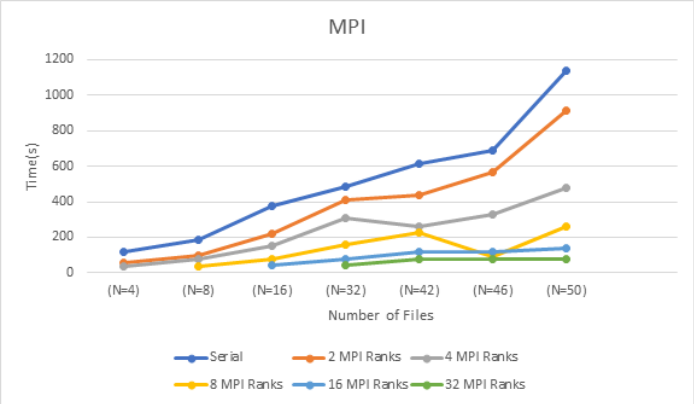

Fig. 6: MPI execution graph for 2-32 processors

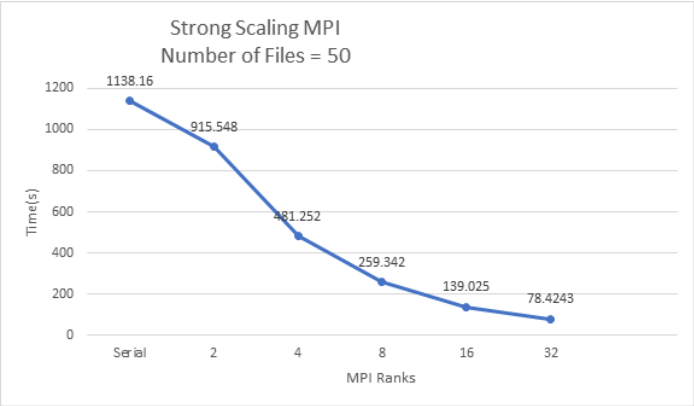

Fig. 7: MPI Strong Scaling

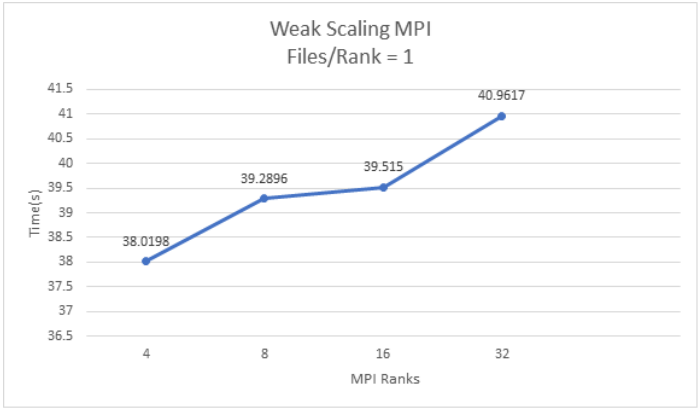

Fig. 8: MPI Weak Scaling

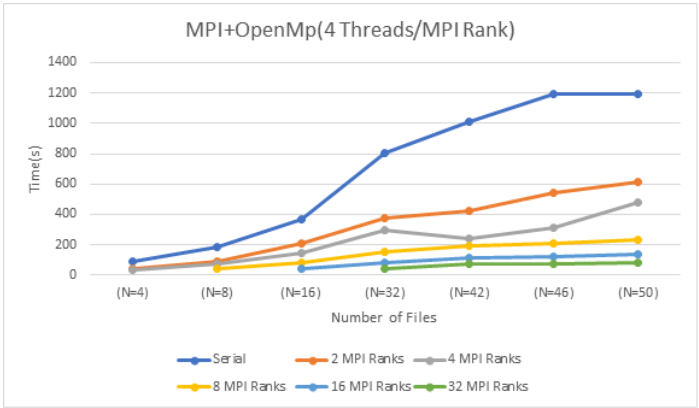

Fig. 9: MPI+OpenMP execution graph for 2-32 processors

a Files/Processor ratio of 1. For this ratio, ideally the graph should be a horizontal straight line as the computing power of each processor would be theoretically completely utilized (Gustafson's law), however in a practical case even though the ratio is 1, with increasing file size, the memory overheads also increase. For a larger file size in a processor, the file distribution overhead, the file reading overhead and execution overhead would also rise even if the ratio is one. Therefore there is an increase in the execution time for higher input size in weak scaling.

C. MPI + OpenMP

One advantage of embarrassingly parallel applications is that there is greater chance that if we keep breaking it down and dividing it, we get even more smaller units of embarrassingly parallel applications. This same philosophy is used in this paper for implementing MPI+OpenMP. The hash function which implements Element,Rank dictionary is implemented in a multithreaded fashion using OpenMP. Owing to a multithreaded manner of execution, the time complexity of the original iteration which was O(N*E) where N is the number of ranks and E is the elements per rank, drops downto O(N*E/P) where P is the number of OpenMP threads. On the Hipergator Node this is specified either using environment variable or using the -cpus-per-task flag. Fig 9 specifies the plot obtained for MPI+OpenMP. The execution time for various input sizes is observed for multiple processors at OMP-NUM-THREADS-4. An interesting observation can be made out for MPI+OpenMP, the shootups observed in execution times for larger input sizes in MPI are saturated in MPI+OpenMP. This is due to the fact that the OpenMP threads with their concurrent execution compensate for any memory access overheads for processing large input files. Other than the saturation at the upper limit, similar blips in the graph are observed for MPI+Openmp and MPI, owing to the similar memory overheads of extended memory hierarchy.

Thr results obtained after MPI+OpenMP are also analyzed for strong and weak scaling. Strong scaling is shown in Fig 10 and weak scaling is shown in Fig 11. The plot obtained is quite similar to the one obtained for MPI. In case of MPI+OpenMP the sharp drop is observed from serial to MPI+OpenMP implementation, this can be attributed by the fact that the serial block of code has movement trace files distributed across processors while the element,rank dictionary concurrently evaluated with OpenMP threads for each rank. Both these factors lead to a high speedup and therefore a steep drop in execution time. However once the MPI+OpenMP stages begin for multiple processors, the upper limit speedup or the drop in execution time is limited by the serial parts of the code i.e Amdahl's law. The weak scaling for MPI+OpenMP is quite different from the one observed in MPI. In this case the graph is quite unifrom and without any blips. This is observed because even though

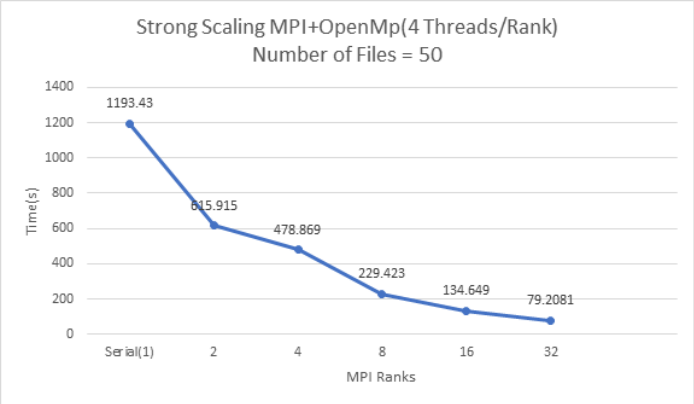

Fig. 10: MPI+OpenMP Strong scaling

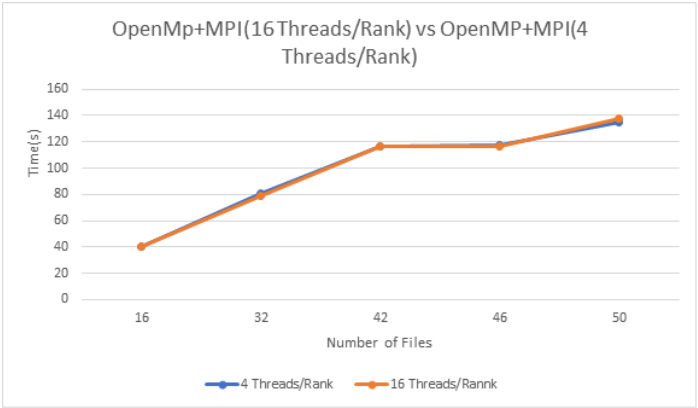

Fig. 12: MPI+OpenMP Thread/Rank analysis

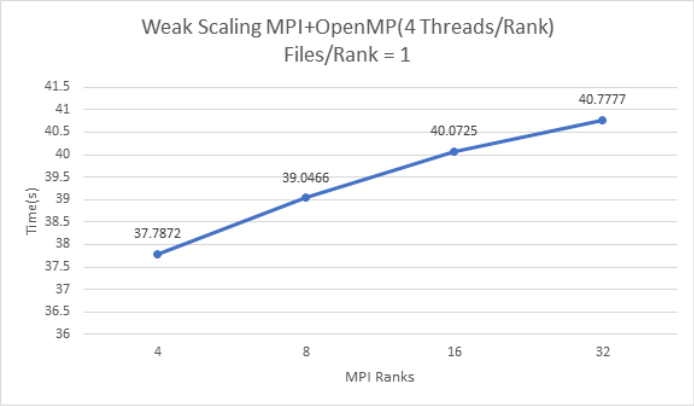

Fig. 11: MPI+OpenMP Weak scaling

the rise in file distribution overhead, file access overhead and file parsing overhead rises with a rise in file size, the increase observed in execution time is smooth because a multithreaded environment compensates for any overheads incurred due to non uniform distribution of file or memory access overheads. This is not the case in MPI only weak scaling.

Another interesting observation is made in Fig 12 for MPI+OpenMP, a plot of thread/rank=1 for 4 and 16 OpenMP threads is for varying number of files. The thread/rank ratio of 1 signifies that the both the number of ranks and number of threads will be scaled by the same amount to better compare it with thread and rank combination of other processor. It is generally expected that for the same amount of work 16/16 thread and rank combination would have smaller execution time. However it is observed that execution times for both the thread/rank combination overlap for different input sizes. This is because even though the threads increase, the ranks have also increased to 16 and more number of processors leads to higher communication overhead, because even though there is no communication between processors during execution, there is an initial communication of file descriptors from master node to worker nodes and there is higher contention for file reads from the shared memory. So the speedup obtained over 4/4 rank thread combination is neutralized due to the mentioned initialization overheads.

D. MPI + CUDA

CUDA is an attractive option to accelerate heavy computation intensive applications. In this scenario however there is no heavy computation task (example addition/multiplication), therefore using GPU to offload the Hash function seems like an overkill. However for purely analytical purposes, we have implemented a MPI+CUDA paradigm where for each node/rank, the processor acts like a host and offloads its hash function (Element,Rank) to a GPU. Each node is alloted 2 Tesla K80 GPUs on Hipergator and in this implementation each node utilizes one of them for 1024 threads. As expected, 63 percent of the time is utilized in copying the Rank to Element file as an array to the device, while 25 percent of the time is utilized in sending the Element,Rank key-value dictionary (hash table) back to the host from the device. Only 12 percent of the computation time is utilized in generating the Element to Rank key-value combination to populate the dictionary. Fig 13 showcases the plot obtained for MPI+CUDA implemention of 1024 threads in a GPU over 8 and 10 processors. We get a non uniform distribution of execution time and number of files in this graph for these two ranks. On the left half of the graph, 10 processors have a higher execution time than 8 processors for the same input size even though its expected that 10 processors would have a lower execution time. This anomaly is observed because in the lower end of the file there is not a lot of workload difference between 8 and 10 processors, but for 10 processors each node offloads the Element,Rank dictionary calculation to the GPU. Therefore the host to device and device to host memory copy overheads are higher for 10 nodes than it is for 8. At the same time at the right end of the graph, the execution time for 8 processors is higher than that of 10 processors, because the workload is better distributed in 10 processors than in 8 so 10 does the same amount of work quicker than 8 eventhough it has higher GPU to Host communication overheads.

V. Conclusion

To summarize, the goal of this was paper was to implement a parallel version of a Dynamic Workload prediction tool. Multiple programming methodologies like MPI, MPI+OpenMP and MPI+CUDA were used. The results obtained from each of the implementation were documented and analyzed. Overall the results obtained match the theoretical expectations. MPI

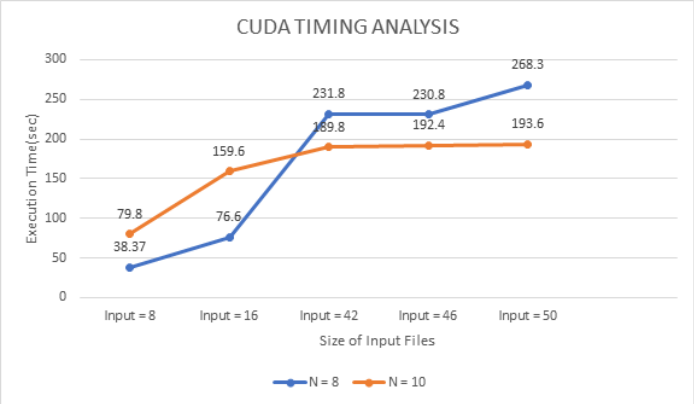

Fig. 13: MPI+CUDA analysis

and MPI+OpenMP are better options to parallelize this tool. Since this tool doesn't involve heavy computations, the overhead of data transfer from host to device for each node in MPI+CUDA makes CUDA an unfeasible option as compared to MPI and MPI+OpenMP. The implementation of this project was done on HiperGator at University of Florida as per the UFRC norms. The technical details of these norms were referred from UFRC wiki page. Hipergator allowed us access to 32 nodes and 32 tasks per nodes along with 2 Tesla K80 GPUs for each node. In future this work can be used expand to Ghost particle analysis.

VI. Acknowledgement

We would like to thank our course instructors Sai Chenna, Aravind Neelkanthan and Trokon Johnson for their support during this course and for this project and the guidance offered by our course professor Dr.Herman Lam.

References

[1] Sai Chenna, "A Report on Particle Workload prediction tool"

[2] Particle-in-Cell algorithms for emerging computer architectures Viktor K. Decyka, , Tajendra V. Singhb a Department of Physics and Astronomy, University of California, Los Angeles, CA 90095-1547, USA b Institute for Digital Research and Education, University of California, Los Angeles, CA 90095-1547, USA

[3] A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes PAULETT C. LIEWER Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California 91109 AND VIKTOR K.DECYK Physics Department, University of California at Los Angeles, Los Angeles, California 90024 Received June 23, 1988; revised December 14, 1988 Vol.58 Issue 8 Aug 2011.

[4] Workload Characterization and Prediction: A Pathway to Reliable Multicore Systems Monir Zaman, Ali Ahmadi and Yiorgos Makris Department of Electrical Engineering, The University of Texas at Dallas Richardson, TX 75080, USA

[5] Particle-in-cell simulations with charge-conserving current deposition on graphic processing units Xianglong Kong, Michael C. Huang, Chuang Ren, Viktor K. Decyk et.al

[6] https://help.rc.ufl.edu