

Distributed Image Processing Framework in Parallel Processing

Umer Iqbal Wani

Assistant Professor, Department of Computer Science and Engineering, Lovely Professional University, Phagwara, Punjab, India

Bhanu Talwar

Assistant Professor, Department of Computer Science and Engineering, Lovely Professional University, Phagwara, Punjab, India

ABSTRACT

The imaging system that provides high content throughput, apply algorithms of ever incrementing complexity and number to streams of large 2D and 3D image data sets. These imaging systems are intrinsically parallel, applying same complex and time-consuming algorithms to huge set of images and that's why can aid from parallel execution. Parallel programming is totally different from serial programming in the main three important areas: how the parallel execution is created and supported, the need for communication and synchronization between the cooperating processes and how the failures are handled. If a computation work is divided between the independent subtasks, each subtask is allotted to multiple processors processing concurrently, so that the reduction in time period is achieved, It is the essence of High Performance Computing (HPC). In the coming sections, we give a brief review of models, patterns and framework that can guide in managing the complexity of parallel programming, then the detail of our framework, test cases and end results.

INTRODUCTION

In today's era, mostly all the computers are working on the concept of Parallel Programming. All of them support parallelism and perform multiple tasks by using pipelining concept. The division for parallelism results in the need for communication between the subtasks so that they can distribute the work, collect end results and manage dependencies between them. Various Parallel Processing Models are described as follows.

PREVIOUS WORK

A. Parallel Programming Models.

The models bound from low level to high level abstractions that majorly deal with efficient execution against productivity and portability [1]. OpenMP is the real standard low-level model for shared memory parallel programming. At the same level, we have Intel Threading Building Blocks which provides C++ template library for parallel programming on multicore processors. MPI is a message passing interface which is a standardized and portable message passing system to function on a wide variety of parallel computing architecture[2]. At the higher level of abstraction, we impose message passing semantics on the shared memory system, example of this is Actors Model and shared memory semantics is implemented on the distributed memory systems as represented in Partitioned Global Address Space (PGAS) languages [4]. To go even on the higher level of abstraction, we have functional programming but it has very limited success in HPC. Along with these all level of abstractions, we have hybrid models which are also becoming very popular with each passing day. The most important framework that we used is an SPMD variant built on MPI, targeting MIMD architectures [6]. To incorporate the hybrid models, just use the plugin functionality.

B. Parallel Programming Patterns and Frameworks

A pattern surrounds the detail of a problem occurring in some domain, a context which is suitable to it and a prototypical solution that is known and have wide acceptance among the domain experts and various parallel programming pattern languages and certain classification is proposed. We have a technique which combines the concept of components and patterns which is an object-oriented reuse technique- Software Frameworks [3]. The difference between patterns and this technique is that this technique is more concrete, provides code that accomplishes the core structure of an application, usually as abstract classes and defines the interaction between these classes or the control flow of the program. We can categorize the frameworks according to their level of abstraction. At the high level, they protect the programmer from the difficulties of parallel programming, but by the same token can remove the access to low level machines specifying tuning efficiencies [5]. Now we are going to review the general sample of frameworks that separate parallel processing from application domain code. Algorithmic Skeleton Frameworks is high level parallel programming model which is a general method for parallel programming. Skeleton enforce extra constraints that provides user with appropriate and right solutions and discourage inappropriate implementations. There are some other general approaches which includes framework that

generates applications from parameterized design pattern templates, which expose different design patterns that are parameterized with different plug in components and other similar template based frameworks such as ‘Selective Embedded, Just in Time Specialization’(SEJIT) who creates new modules specialized for particular computation[10][11]. There are other more frameworks that enforce a special parallel programming pattern, for example- MW which is a C++ framework that enforces parallel computations using the omnipresent master-worker model in applications [8].

We can have more specific frameworks for domain or application in parallel programming which provides an optimized core parallel processing infrastructure and support libraries. To increase reusability and maintainability, we have an approach known as parallelizable patterns in image processing libraries [9]. In this library, a single API is issued regardless of the fact that we have sequential or parallelized implementation. In addition to this we have data and task parallelism in which the former one uses algorithm skeletons for image operators and latter one composition into an image application task graph. In the next section, we describe a lightweight framework that have the extensible plugin of parallel patterns and domain code, and also allows for simple extension of the framework itself [7].

THE PROPOSED FRAMEWORK: DFRAME

It is basically designed as a distributed image processing framework. Its task is to distinguish the orthogonal parallel processing from image processing and guards the application developers from the parallelizing aspect of development [12]. This isolation helps the programmers to divert their all concentration on the core parallel programming element, being able to develop plugin models implementing parallel patterns that application developers can use. It also allows for the making of parallelized entities, such as the parallelized pipelines consists of individual image processing pipelines that runs and works concurrently in the parallel environment by DFrame.

The features of framework that includes parallel processing are:

- A modular design that supports the plugin of models implementing the parallel patterns and for plugin imaging libraries associated with plugin models.
- Point to Point and collective Master-Worker models are supplied, that facilitate adaptive distribution of processing entities based on efficient load balancing.

- Instrumentation to monitor the performance of parallel programs, and provide feedback on operation and efficiency.
- Configurable debugging capabilities to help detect deadlocks, synchronization issues, memory leaks and other programming errors.

Also, important features for imaging are also provided:

- An image filtering library, representing dynamically distributable entities within the framework.
- An image IO library for loading and storing tiff (Tag image file format) and stk (Stick Figure Files) images.
- An imaging library specific to distributable ray tracing algorithms and ancillary infrastructure.

The various abstractions are provided at various levels by DFrame. At the lower level of DFrame, to manage the plugins of models implementing parallel patterns and application modules that use those modules, it is equipped with an infrastructure to perform all these tasks. Also it provides basis for MPI point to point and collective communication processes, with the deconstruction packing, sending, receiving, unpacking and reconstruction of objects being performed transparently by the framework. It also provides its users with the client interface enabling clients to interact with DFrame [10]. To make sure that parallel resources are used in efficient manner, Models are implemented that plug into and use the Dframe services to arrange the distribution of data and execution of application code. By using an API defined by the model, an application code module can only communicates with its associated model. Apart from application code, a model then uses DFrame's communication infrastructure to effect parallel execution and interaction. Its process information includes information about the model or model group which should run or not, along with the information about the target application code. A model is first loaded and run, then it uses the DFrame infrastructure to load and exercise the exact application code. To allow reusability and extensibility of different parallel programming models, it is incorporated with the module and dynamic 'plugin' design [4]. The latest feature of Dframe is that it can automatically adjusts itself according to the parameters of model and also selects the model that is most appropriate so that we can optimize the efficiency.

At the topmost level, multiple task specifications can be composed to create a pipeline of parallel entities. A priori knowledge is automatically inserted by the user at the time of

composition of pipeline, so that user can be shielded from the problems of further intervention. The most important aspect is that the framework can create multiple instance of same processing entity when demanded. So to enforce and manage concurrency in system, we add parallel processing instances of data streams or data partitions. To manage data flow, we use temporal and special partitioning strategy where former is used to process multiple images concurrently and latter is used to divide a single image into multiple volumes that are also processed alternatively. Figure 1 shows the DFrame used to create parallel tasks.

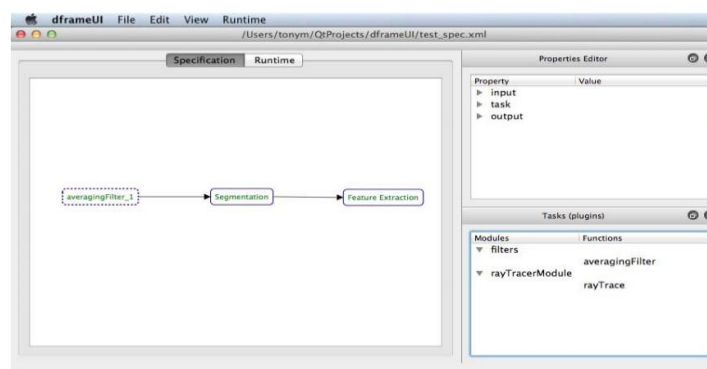


Figure 1: The DFrame UI prototype, built using QT4

EXPERIMENTAL SETUP

The following tests use custom benchmarking on non-standardized 3-D images from cell data research, to make the proficient use of framework.

A. Test on applying a 2D averaging filter to slices of one 3D image, with distribution of the image slice

In this test, a master module loads a 3D image in the system and allocates the image slice to each worker process, which processes the image concurrently and returns back the updated and processed image slice to the master. Then master will recompose the slices into the resultant 3D image and then stores into the disk. The following graph shows the time and speedup terms, in which image is sliced into 21 volumes. The master process does the work of reading and writing the image to the disk. Additional overhead time is consumed in distributing each slice to the worker system and collecting resultant slices from the workers. From the graphs it is provident that the results are very encouraging. If we have only master as the one process in the system, then the master process does all the working. Otherwise, master will only allocate and collects the results, will not perform the image processing part.

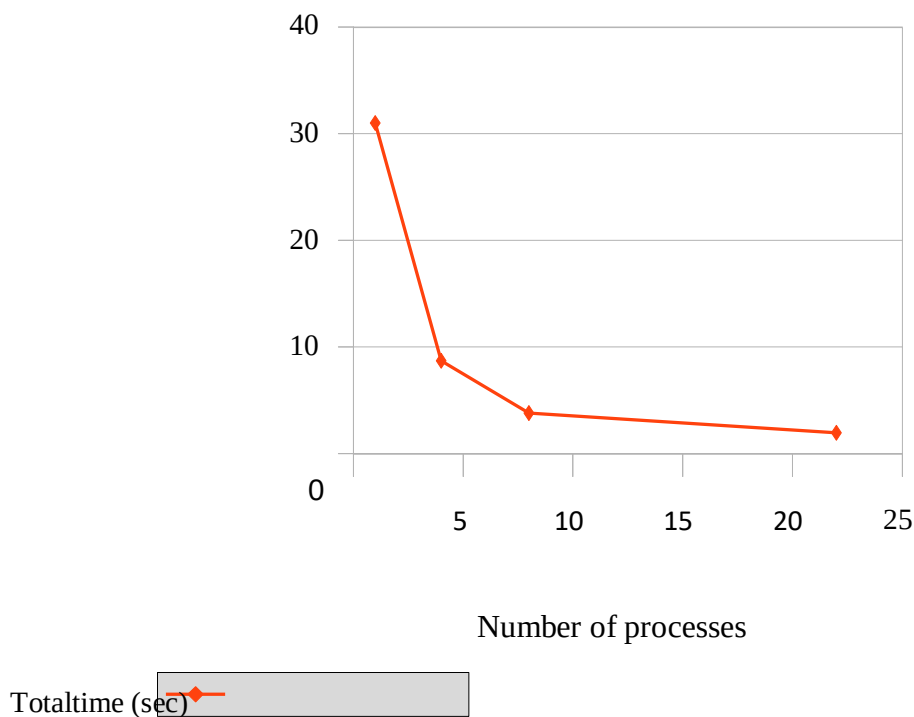
In our sample, 3D image contains 21 slices, so for the following configuration our system is tested:

4 processes: 1 master + 3 workers having 7 subtasks each

8 processes: 1 master + 7 workers having 3 subtasks each

22 processes: 1 master + 21 workers having 1 subtask each

We have also applied a ‘scatter-gather’ master worker model to this test case, in which all the processors are engrossed in MPI scatter and gather collective operations to distribute image slices and return results. In this model, master all does the processing part resulting in the need of one less processor. Almost the same results were found, also additional feature is the ease with which one model can be swapped with another. This model works well if the each processor have identical computational effort, as each process receives work and returns results as a collective operations across all processes.



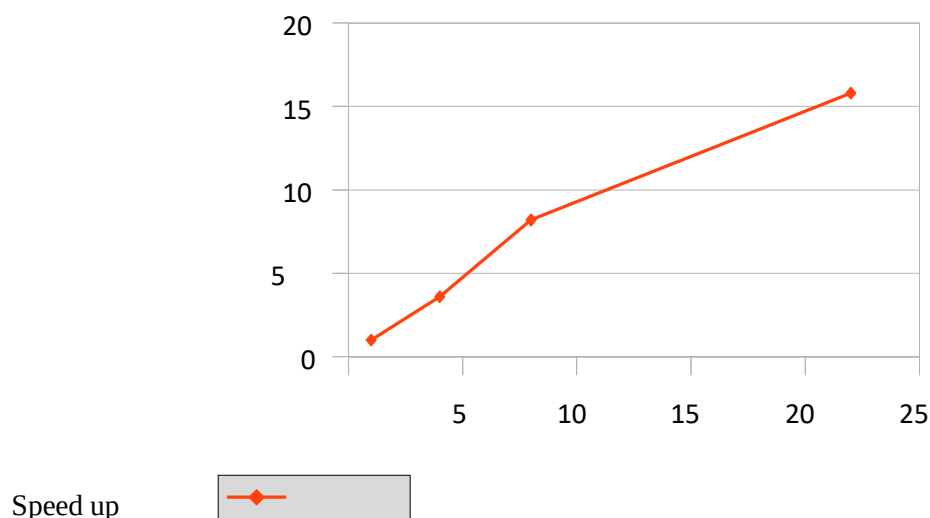


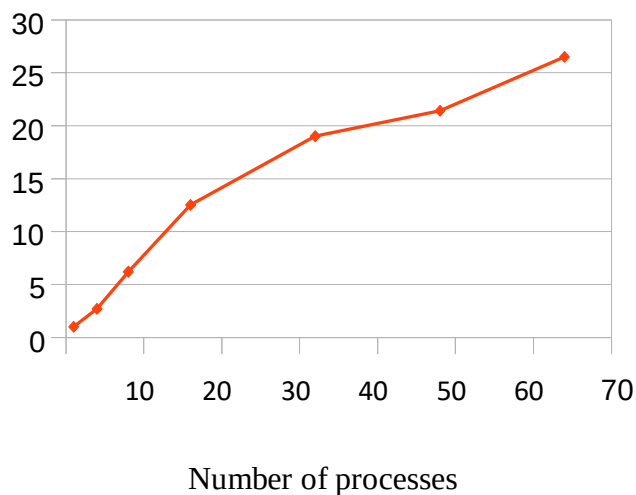
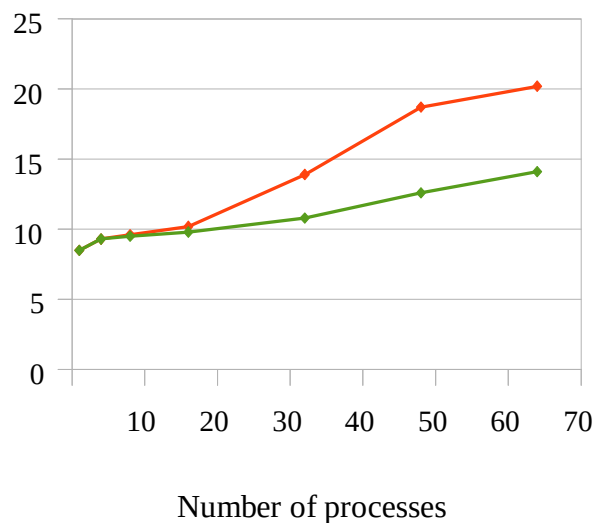
Figure 2: Performance with each worker processing one image slice of a 3D image

B. Tests on applying a 3D averaging filters to many 3D images, with distribution of image paths:

For this test case, we use the master worker model to test the degree increment in the speed that is attained in high content throughput scenario. To distribute the paths of a collection of 3D images one to every worker process, master module code is implemented. With the increment in the number of workers, the number of images processed also increments such that worker performs the function of read-execute-write on one 3D image. Each worker processes all voxels (value on a regular grid in three-dimensional space) of each distinct 3D image using 3x3x3 averaging filter, with all images being the same size. Ideally, with each addition of processor in the system will result in constant time as each processor process individual its own image. But, in reality we would experience a degradation in the performance of the system as the load on the storage system rises and also because of interprocess communication as process count is increased. Since each worker processes its own image then the total time recorded to perform the whole task is the ‘worst case’ time of the slowest processor to perform its task. It is also recorded as worst time because each processor will attempt to read its image at the same time, execution is performed by each worker at the same time and moreover write operation is also done by each worker at the same time, with some incremented variability.

The performance graph shows the time and speed characteristics of $n+1$ processors used to process n images. The speedup graph presents a view of attained performance. Not ideal, but these preliminary results are encouraging. As anticipated, the ‘worker execute time’ increases with increase in the processor count as increase in IO contention as now more images are

loaded for read and write operation. To investigate further, we linked in the MPE analysis tool. The analysis revealed that initialising the MPI tool for an increasing fraction of total overhead as processor counts increased. Also, significant time was spent at the barriers inserted into program. Some of them are added to allow reasoning of program flow and are otherwise redundant and will be removed to improve speedup. However, it is still an area of investigation that why processes can remain at barriers for significant period after all processes have arrived at the barrier. Other way, inter node communication is very small and did not alter performance anyway.



Speed up

CONCLUSION

The DFrame is designed to isolate the parallel infrastructure, models and patterns from application code, so that efficient parallel programmers can concentrate on developing

parallelising application code, while domain experts are protected from this aspect and concentrated over developing the application code. The higher level abstraction can impose a trade-off but the primary results are positive. This isolation of area of their concerns is also benefit when improving the system, where the efforts to improve the DFrame and models can be shared across all the associated module cycle, and the modular design will boost its use across many image processing projects requiring parallel processing capabilities. Performance analysis can help light up the characteristics and bounds of the system storage IO, interprocess communication, memory hierarchies and processors, and their interrelation and effect on the problem being solved and must be considered in the ‘shape’ of the problem being solved. Instrumentation within the DFrame is made available across models and modules, to aid performance analysis. Analysis of tests have revealed where model implementation can be improved and that it might be worth further research into using parallel IO and perhaps dedicating a subset of processors to storage IO and then to use internode communication to transfer data about the network. This must be balanced against the increased program complexity.

References

- [1] D. Bonachea and J. Jeong. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. Technical report, CS258 Parallel Computer Architecture Project, 2002.
- [2] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99,2004.
- [3] Antal A. Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: standard template adaptive parallel library. In Haber et al. [3].
- [4] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.
- [5] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High-Performance Computing Applications*, 21(3):291–312, August 2007.
- [6] J. A. Crawford and C. M. Mobarry. HRUNTING: a distributed shared memory system for the BEOWULF parallel workstation. In *Proceedings of the IEEE Aerospace Conference*, pages 337–344, Mar 1998.
- [7] Cray XE6. [online]. https://wickie.hlr.de/platforms/index.php/Cray_XE6.
- [8] Jeffrey Daily, Abhinav Vishnu, Bruce Palmer, and Hubertus van Dam. PGAS Models Using an MPI Runtime: Design Alternatives and Performance Evaluation. In *The*

International Conference for High Performance Computing, Network, Storage and Analysis. IEEE Computer Society, 2013.

[9] The DASH Project. [Online], 2014. <http://www.dash-project.org/>.

[10] James Dinan, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, and Rajeev Thakur. An implementation and evaluation of the MPI 3.0 one-sided communication interface. In Preprint ANL/MCSP4014-0113. IEEE Computer Society, 2013.

[11] James Dinan, Pavan Balaji, Jeff R. Hammond, Sriram Krishnamoorthy, and Vinod Tipparaju. Supporting the Global Arrays PGAS Model Using MPI One-Sided Communication. In IPDPS, pages 739–750. IEEE Computer Society, 2012.

[12] Karl Furlinger, Colin Glass, Jose Gracia, Andreas Knüpfer, Jie Tao, Denis Hünich, Kamran Idrees, Matthias Maiterth, Yousri Mhedheb, and Huan Zhou. DASH: Data Structures and Algorithms with Support for Hierarchical Locality. In Euro-Par Workshops, 2014.