Proceedings of EuroPVM/MPI 2002, Sep. 29 – Oct. 2, Linz, Austria, LNCS, Springer-Verlag.

Enhanced File Interoperability with Parallel MPI File-I/O in Image Processing

Douglas Antony Louis Piriyakumar[1], Paul Levi[1] and Rolf Rabenseifner[2]

[1] IPVR, Department of Computer Science, University of Stuttgart, Germany
piriyaku@informatik.uni-stuttgart.de
[2] HLRS, High Performance Computing Center, University of Stuttgart, Germany
www.hlrs.de/people/rabenseifner/,
rabenseifner@hlrs.de

Abstract. One of the crucial problems in image processing is *Image Matching*, i.e., to match two images, or in our case, to match a model with the given image. This problem being highly computation intensive, parallel processing is essential to obtain the solutions in time due to real world constraints. The Hausdorff method is used to locate human beings in images by matching the image with models and is parallelized with MPI. The images are usually stored in files with different formats. As most of the formats can be converted into ASCII file format containing integers, we have implemented 3 strategies namely, *Normal File Reading, Off-line Conversion and Run-time Conversion* for free format integer file reading and writing. The parallelization strategy is optimized so that I/O overheads are minimal. The relative performances with multiple processors are tabulated for all the cases and discussed. The results obtained demonstrate the efficiency of our strategies and the implementations will enhance the file interoperability which will be useful for image processing community to use parallel systems to meet the real time constraints.
Keywords. Supercomputer, Computational Models, File Interoperability, Parallel Algorithms, Image processing and Human recognition.

1 Introduction

The challenging image processing problems in the wide spectrum of defense operations to industrial applications demand run-time solutions which need enormous computing power aptly provided by supercomputers [1]. In most of the core applications problems in Robotics, Satellite Imagery and Medical Imaging, recognizing the crucial parts or items or structures in the given images continues to attract more attention. Thus, the major factor in these computer vision related fields is matching objects in images [2]. Currently, lot of interests are evinced on human motion based on model based approach, and temporal templates [3] with real-time constraints also [4]. A survey of the visual analysis of human movement is presented in [5].

In this paper, an efficient parallel algorithm is developed using the Hausdorff method to ascertain the presence of a human being in the image (or image sequence) by matching the images and the models. In any parallel system, the

proper choice of the parallel computing model is the paramount factor. MPI (message passing interface) is used here for its efficiency, portability and functionality [6]. However, due to the domain specific nature of the problem, the images usually stored in files, differ in formats considerably. This poses an impediment to the efficient implementation of the parallel algorithm despite parallel I/O implementations in MPI-2 [7]. As most of the file formats can be converted into ASCII file format in many systems, here we have implemented 3 strategies including one for free format integer file reading and writing. The algorithm is implemented on the supercomputer CRAY T3E with MPI model. A different parallelization method is formulated so that I/O timings are optimized. The time taken for I/O in all the cases are compared with analysis.

The following sections are organized as mentioned here. Section 2 introduces the real application of an image processing problem. The parallel algorithm for image matching is sketched in section 3. In section 4, the File interoperability in MPI is portrayed. The experimental results are analyzed in section 5. Section 6 concludes the paper.

2 Image Processing with Hausdorff Method

The Hausdorff distance [8] is defined as follows: Let the two given finite point sets be $A = a_1, a_2, ..., a_n$ and $B = b_1, b_2, ..., b_m$. Hausdorff Distance H(A,B) = max(h(A,B),h(B,A)) where $h(A, B) = max_{a \in A} min_{b \in B} EN(a - b)$ where EN is the Euclidean norm (just the distance between the two points).

Here, we have images which have to be analyzed and the models which depict the pattern to be recognized, in our case human beings. Using the general corner detecting algorithm (here SUSAN filter is used [9]), the corners of the images are found which serve as the point set B. Similarly, all the model images are filtered with the same SUSAN filter to find the corners and are stored as point set A. For this application, it is sufficient to find h(A,B). The main Hausdorff distance H(A,B) being a metric has to satisfy the symmetry property and that is why it is defined as maximum of the directed h(A,B) and h(B,A). The model is placed over all possible positions on the image and for each such position, h(A,B) is computed which is computational intensive. The model is found to be present in the image if h(model,image) is less than some predefined threshold. Against various models, the image is matched using the method.

3 Parallel Algorithm for Image Matching

3.1 Parallelization

For this particular problem, there can be at least three methods to parallelize. One way is to take each image by a processor and match with all models. The other way is to take one by one all images by all processors and divide the set of models equally among the processors. The third way is to divide each time one image by the number of processors and match that portion of image with

all models. In the last model, overlapping is essential to get the proper solution. The second model is preferable in the situation while tracking a person is the major factor. In this paper, the second method is implemented. A good insight to various strategies of parallelization can be found in [10], [11] and [12].

3.2 Outline of Parallel Program

Let r be the number of images, q be the number of models and p be the number of processors.

```
MPI_Comm_rank(MPI_COMM_WORLD,&j);
MPI_Comm_size(MPI_COMM_WORLD,&p);
for each image i=1..r do
{ MPI_File_open(...,MPI_COMM_WORLD,...);
  MPI_File_read(...);  /*all processors read i^th image */
  MPI_File_close(...);
  for each model k = j, j+p, j+2*p, .. q do
    { MPI_File_open(...,MPI_COMM_SELF,...);
      MPI_File_read(...);  /*each processor j reads only the corresponding model k */
      MPI_File_close(...);
      h = Hausdorff distance of the image i matched with model k for all positions.
      h_partial_min = min ( h, h_partial_min);
      whenever the h(k,i) < threshold, this position is notified.
      /* if required the best matching model is also computed depending upon
      the minimum threshold. */
    }
  MPI_Allreduce(h_partial_min,&h_min,1,MPI_FLOAT,MPI_MIN,MPI_COMM_WORLD);
}
```

4 Parallel I/O and File Interoperability in MPI-2

As the vital focus of this paper is not on parallel processing, the I/O operations especially file related operations are investigated. The significant optimizations required for efficiency can only be implemented if the parallel I/O system provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files [7]. Parallel reading of the same image or model into the memory of several MPI processes can be implemented with the MPI_File_read_all. This collective routine enables the MPI library to optimize reading and broadcasting the file information into the memories of all processes. In image processing, there exists also a huge number of different formats to store the image data in files. The standard image processing software gives the options of a proprietary format or a standard ASCII format. Because most of the formats can be converted into ASCII file format in many systems, and to circumvent problems with the 64-bit internal integer format on the Cray T3E, we have chosen to use only an ASCII format. Therefore, it is mandatory to implement the conversion of ASCII file (mostly representing integers being pixel

coordinates and gray values) so that file interoperability in MPI can be used effectively for image processing. As the sizes of the files increase obviously the I/O overheads also increase. In image processing, there will be always many files both for images and models. Hence, it is not only the sizes of the images, but also the number of them is a matter of concern for I/O overheads.

The file interoperability means to read and write the information previously written or read respectively to a file not just as bits of data, but the actual information the bits represent. File interoperability has three aspects namely, 1. transferring the bits, 2. converting different file structures and 3. converting between different machine representations. The third being the concern here, the multiple data representations and the inability to read integer data stored in an ASCII file which is needed for image processing are explained in the following subsection.

4.1 Data Representations

MPI-2 defines the following three data representation, 1. *native*, 2. *internal* and 3. *external32* [7]. In *native* representation, the data is stored in a file exactly as it is in memory. In *external32* format, also a binary data representation is used. Obviously, it is impossible to use these formats directly to read integer data from ASCII files. The *internal* representation cannot be used for data exchange between MPI programs and other non-MPI programs that have provided the image data, because the internal representation may be chosen arbitrarily by the implementer of the MPI library. MPI-2 has standardized also a method to use *user-defined* data representation. Here, the user can combine the parallel I/O capabilities with own byte-to-data conversion routines. The major constraint is that the representation of a given data type must have a well-defined number of bytes. As the number of digits of integers in an ASCII file vary (and each integer may end either with a blank or an end-of-line character), user-defined representation also cannot help reading integers efficiently from ASCII files.

4.2 Reading Integer Data from ASCII File with MPI I/O

The former constraints force the implementation of the following strategies:

Normal File Reading with fscanf In this first strategy, the files are read using normal file reading command *fscanf* instead of MPI for the sake of comparison with MPI file I/O operations. It may be recalled that there is no need for conversion as fscanf can directly read the integers from the files.

Off-line Conversion In this second strategy, the ASCII file is converted into a native file by a separate program. This gives the facility to convert the required ASCII file off-line which enables the image processing program to read the native file without any difficulty. To achieve heterogeneity, MPI external 32 data representation can be used instead of the native format.

Runtime Conversion In this third strategy, the entire ASCII file is read into a large buffer of type CHAR, and then individually by reading every character

ranks=0	1	2	3	4	5	6	7	Remarks
R(0,0)	R(1,1)	R(2,2)	R(3,3)	R(0,4)	R(1,5)	R(2,6)	R(3,7)	read first images&models
c(0,0)	c(1,1)	c(2,2)	c(3,3)	c(0,4)	c(1,5)	c(2,6)	c(3,7)	
r(1)	r(2)	r(3)	r(4)	r(5)	r(6)	r(7)	r(0)	exchange models
c(0,1)	c(1,2)	c(2,3)	c(3,4)	c(0,5)	c(1,6)	c(2,7)	c(3,0)	
r(2)	r(3)	r(4)	r(5)	r(6)	r(7)	r(0)	r(1)	
c(0,2)	c(1,3)	c(2,4)	c(3,5)	c(0,6)	c(1,7)	c(2,0)	c(3,1)	
r(3)	r(4)	r(5)	r(6)	r(7)	r(0)	r(1)	r(2)	
c(0,3)	c(1,4)	c(2,5)	c(3,6)	c(0,7)	c(1,0)	c(2,1)	c(3,2)	
R(8)	R(9)	R10)	R(11)	R(12)	R(13)	R(14)	R(15)	read next models
c(0,8)	c(1,9)	c(2,10)	c(3,11)	c(0,12)	c(1,13)	c(2,14)	c(3,15)	
r(9)	r(10)	r(11)	r(12)	r(13)	r(14)	r(15)	r(8)	
c(0,9)	c(1,10)	c(2,11)	c(3,12)	c(0,13)	c(1,14)	c(2,15)	c(3,16)	
...	...	...	...	...	...	...	...	
R(4,q-1)	R(5,q-2)	R(6,q-3)	R(7,q-4)	R(4,q-5)	R(5,q-6)	R(6,q-7)	R(7,q-8)	read next image & models (with reverse sequence, q = # of models)
...	...	...	...	...	...	...	...	

Table 1. Parallelization scheme of I/O and computation.

till it is terminated either by a blank or by an end-of-line character, the same is converted into an integer at run-time. In fact, the original file remains as ASCII file and is still used. The conversion can be stored as a native file for further use, if need be. It may be recalled the ASCII to Integer conversion function is very easy to implement which is also system independent.

4.3 Optimizing the Parallel I/O

The image data usage pattern has two chances for optimization: (a) all image data must be reused (and probably reloaded) for comparing with several models, and (b) all models must be reused (and probably reloaded) for comparing with several images. In the sequential version of the software, each image is loaded once and all models are loaded again for comparing with each image. By reversing the sequence of models for each even image number, at least the latest models can be cached in memory. In the first parallel version loading of the images can be optimized with collective reading into all processes.

If more than one image can be analyzed in parallel, i.e., if one can accept an additional delay for the analysis of an image because not all available processors are used for analyzing and because the start of the analysis is delayed until a set of images is available, then the parallelization can be optimized according to the scheme in Table 1. The scheme shows the analysis of 4 images in parallel on 8 processors. R(i,k) denotes reading of the image i and model k, R(k) is only reading of model k, r(k) is receiving of model k with point-to-point communication from the right neighbor (sending is omitted in the figure), and c(i,k) denotes the computation of the Hausdorff distance for image i and model k.

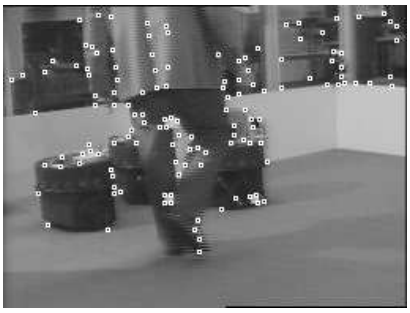

Fig. 1. A sample Image

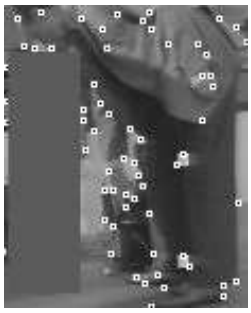

Fig. 2. A sample Model

Looking at the scheme, note that reading the image into several processors at the same time (e.g., image 0 into processes 0 and 4) can be still optimized with collective reading (MPI_File_read_all) that internally should optimize this operation by reading once from disk and broadcasting the image data to the processes. Reading several images and models at the same time can be accelerated by the use of striped file-systems. The scheme is also optimized for a cluster of shared memory nodes (SMPs). The vertical bar between rank 3 and 4 may donate such a boundary between SMPs. One can see on each node, that only one model is received from another node (and another model is sent) while exchanging all models.

5 Results and Analysis

For the purpose of illustration, four sample images (one shown in Fig. 1) and four models (one shown in Fig. 2) are considered. The algorithm is tested with 1, 2 and 4 processors on the Cray T3E-900 at HLRS. As the interest of the paper is on I/O, the I/O timings per process are tabulated in Table 2 for 4 images and 4 models. The timing is done with MPI_Wtime(). Before starting each I/O timing, a barrier is done to prohibit that any synchronization time is assessed as I/O time. Although the I/O requires only a small part of the total execution time in the current version of the program, it is expected that on faster processing systems and with better optimization of the Hausdorff algorithm, I/O will be a relevant factor of execution speed. In the *original* parallelization, each image is read by all processes (which may be optimized by the MPI library), and for each image, each process reads only a subset of the models according to the numbers of processors. In the *optimized* parallelization, each image is read by only one process, and for each set of images analyzed in parallel, each model is read only once and then transferred with message passing to the other processes. Table 3 shows the accumulated number of reading an image or model file or transferring a model for our test case with 4 images and 4 models.

We started our experiments with normal reading with fscanf. Our original parallelization resulted in a larger I/O time because each image had to be read

No.	*Parallelization*	*File Op*	*Conversion*	*I/O Entities*	*1 proc*	*2 proc*	*4 proc*
1	Original	fscanf	On-line	integers	0.126 s	0.130 s	0.142 s
2	Original	MPI	On-line	characters	7.087 s	6.173 s	6.563 s
3	Original	MPI	On-line	whole file	0.157 s	0.196 s	0.234 s
4	Original	MPI	Off-line	3*int, 2*array	0.189 s	0.182 s	0.195 s
5	Optimized	MPI	On-line	whole file	0.163 s	0.071 s	0.040 s
6	Optimized	fscanf	On-line	integers	0.129 s	0.068 s	0.036 s

Table 2. I/O time per process: wall clock time per process to handle the reading of 4 images and 4 models, including repeated reading or message exchanges of the model.

Parallelization	accumulated number of images + models read with		
	1 process	*2 processes*	*4 processes*
Original	4*1 + 16 + 0	4*2 + 16 + 0	4*4 + 16 + 0
Optimized	4*1 + 16 + 0	4*1 + 8 + 8	4*1 + 4 + 12

Table 3. Each entry shows the accumulated number of *images read* + *models read* + *models exchanged* by all processes with the different parallelization schemes, e.g., 4*2+16+0 means, that 4 times 2 identical images, and 16 models are read, and 0 models are exchanged by message transfer.

on each processor again. In the second experiment we parallelized this reading and substituted each fscanf by MPI-2 file reading. Because reading of ASCII integers is not available in MPI-2, we have chosen reading characters. Normally each integer is expressed only with a few characters, therefore, the expected additional overhead was not expected very high. But the measurements have shown that this solution was 46 times slower than the original code. The MPI-2 I/O library on the Cray T3E could not be used in a similar way as fscanf() or the getc() can be used. To overcome the high latency of the MPI I/O routines, reading the whole file with one (experiment No. 3) or only a few (No. 4) MPI operations was implemented. But there is still no benefit from parallelizing the I/O. The I/O time per process grows with the number of processes and the accumulated I/O time with 4 processors is therefore 4–6 times more than with one processor. In the last two experiments, the parallelization was optimized to reduce the number of reading of each image and model. This method achieves an optimal speedup for the I/O. But also with this optimization, the fscanf solution is about 10 %faster than the MPI I/O solution on 4 processes.

These experiments have shown that (a) MPI I/O can be used for ASCII files, (b) but only large chunks should be accessed due to large latencies of MPI I/O routines, and (c) optimizations that can be implemented by the applications should be preferred of optimizations that may be done inside the MPI library, (d) as long as many small or medium ASCII files should be accessed, it may be better to use standard I/O by many processes and classical message passing or broadcasting the information to all processes that need the same information, than using collective MPI I/O.

6 Conclusion

One of the computationally intensive image processing problem, *Image matching* which demands the solutions within real time constraints is investigated focusing the attention on MPI File Interoperability especially with ASCII files. Due to the domain specific nature of the problem, the images usually stored in files, differ in formats considerably. This poses an impediment to the efficient implementation of the parallel algorithm despite parallel I/O implementations in MPI-2. As most of the formats can be converted into ASCII file format in many systems, the three strategies namely, *Normal File Reading, Off-line Conversion and Run-time Conversion* for free format integer file reading and writing are implemented on Cray T3E with MPI-2. The modified parallelization presented in the paper produced better results comparing the I/O timings. The important conclusion of the paper is that the problem of file format conversion in image processing applications can be efficiently solved with the proper parallelization and MPI parallel I/O operations. In all the images, the accurate positions (to one pixel resolution) of the human beings with the corresponding best model are not only found correctly but also efficiently as the obtained results demonstrate.

References

1. Ian Foster and Carl Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, 1998.
2. Douglas A. L. Piriyakumar, and Paul Levi, "On the symmetries of regular repeated objects using graph theory based novel isomorphism", The 5th International Conference on PATTERN RECOGNITION and IMAGE ANALYSIS, 16 - 22 October, 2000, Samara, The Russian Federation.
3. Aaron F. Bobick and James W. Davis., "The Recognition of Human Movement using Temporal Templates", IEEE Trans. PAMI, vol.23, no.3, pp.257-267, March, 2001.
4. N. T. Siebel and S. J. Maybank, "Real-time tracking of Pedestrians and vehicles", IEEE International workshop PETS'2001.
5. D. M. Gavrila, "The Visual Analysis of Human Movement: A Survey", Computer Vision and Image Processing, vol. 73, no. 1, pp. 82-98, 1999.
6. William Gropp, Ewing Lusk and Anthony Skejellum., Using MPI, MIT press, 1995.
7. MPI-2, Special Issue, The International Journal of High Performance Computing Applications, vol. 12, no. 1/2, 1998.
8. Daniel Huttenlocher, Gregory Klanderman and William Rucklidge., "Comparing images using Hausdorff distance", Transaction on PAMI, vol. 15, no. 9, pp. 850-863, September, 1993.
9. S. Smith and J. Brady, "SUSAN - a new approach to low level image processing", Int. Journal of Computer Vision, vol. 23, no. 1, pp. 45-78, 1997.
10. Armin Baeumker and Wolfgang Dittrich., "Parallel algorithms for Image processing: Practical Algorithms with experiments", Technical report, Department of Mathematics and Computer Science, University of Paderborn, Germany, 1996.
11. J. F. JaJa, Introduction to Parallel Algorithms. Addison-Wesley, 1992.
12. Rolf Rabenseifner, Parallel Programming Workshop Course Material, Internal report 166, Computer Center, University of Stuttgart, 2001.
`http://www.hlrs.de/organization/par/par_prog_ws/`