Medical Image Processing and Visualization on Heterogenous Clusters of Symmetric Multiprocessors using MPI and POSIX Threads

Christoph Giess, Achim Mayer, Harald Evers, Hans-Peter Meinzer
Deutsches Krebsforschungszentrum, Dept. Medical and Biological Informatics, Heidelberg, Germany

Abstract

In this paper we present the design and implementation of a parallel system for interactive segmentation and visualization of three-dimensional medical images which is distributed on a heterogenous cluster of workstations or personal computers. All image processing functions are multithreaded to use the advantages of symmetric multiprocessors. Its platform-independence is achieved using standardized libraries like MPI and POSIX Threads.

1 Introduction

Modern three-dimensional imaging systems like computer tomography (CT), magnetic resonance imaging (MRI) or 3D-ultrasound devices (US) are used increasingly in medical diagnostics. The information, about position, size and shape that is included in this data, bears important parameters to determine pathological changes. Surgical treatment planning and therapy control are further applications. Digital image processing should support medical personnel by selecting relevant information and editing these visually. Most of the existing applications work only on two-dimensional images. Recent image acquisition techniques enable increased resolution in the third dimension allowing real three-dimensional image processing operations. The compute power required rises considerably for three-dimensional algorithms. A constant resolution improvement will further enlarge this problem.

The presentation of processed images is possible in two ways: as a two-dimensional slice or as a three-dimensional reconstruction. The latter has proven to be helpful when assessing anatomical structures. It is faster to comprehend, understandable for laymen and can therefore improve the communication between doctor and patient as well as interdisciplinary collaboration.

The benefit utilizing image processing systems depends especially on the response times. A wide acceptance of such a system will only be achieved if it offers an interactive usage. Another important factor is the cost depending on hard- and software requirements, education and technical support. The design aim is to provide medical supply of high quality at a low expenses. Using existing hard- and software reduce the cost significantly.

The use of a parallel system is often required to fulfill time-critical aspects. To our opinion, clusters of workstations or personal computers are the most suitable platforms for clinical use. The increasing use of symmetric multiprocessors (SMPs) in local networks offers extra computing power without further increasing the communication overhead between hosts.

In the field of image processing and visualization many algorithms have been developed for distributed memory architectures. Most of them are only considered in isolation, without a real application, limited to 2D or bound to a special hardware, in most cases MPPs.

This paper presents an approach for a distributed parallel image processing system covering important segmentation and visualization algorithms for three-dimensional medical data, based on a heterogenous network of SMPs.

2 Material

In the clinical field the existing hardware is in most cases limited to personal computers and workstations. SMPs offer a good price-performance ratio for computionally intensive applications, like image processing. In the near future they will be added to the existing hardware. Ethernet and Token Ring are the commonly used network architectures today slowly changing to Fast Ethernet and ATM.

The computing model combining these different configurations can be described as a cluster of symmetric multiprocessors. We post no requirements regarding the number of hosts in a cluster or number of CPUs per computer. That means, a standalone single-processor machine also fits into this model. The model has to support heterogenous systems with any networking infrastructure.

Only platform-independent standards are used to fulfill these requirements. The program was implemented in ANSI-C which offers a high performance and is availible on all platforms.

The de facto communication standard for distributed computing is MPI. Currently it has some disadvantages compared to other message passing libraries, particulary PVM, however it is being promoted by industry and research. Neither the MPI-1.1 nor MPI-2.0 standard requires a heterogeneous cooperation [1], that is why vendor implementations should be avoided for our purposes. The preferred solution is therefore a platform-independent implementation like MPICH from the Argonne National Laboratory, which we are using in release 1.1.0, or the LAM distribution from the Ohio Supercomputer Center. Both offer additional support for SMPs besides their support for MPPs. Multiple processes exchange messages on SMPs using shared memory.

The thread model is more appropriate for shared memory systems. It provides a higher performance through avoiding message passing and consumes less resources, in particulary main memory. The latter is necessary to minimize or prevent swapping when handling large data sets. With POSIX Threads (IEEE 1003.1c) there is also a platform independent standard.

This resulted in a two-level model. On every available host just one process is running. The processes communicate via MPI. Any host starts as many threads as it has processors, using shared memory for control.

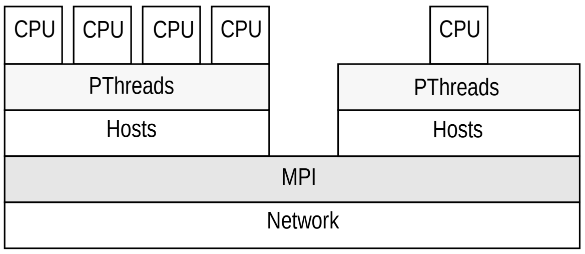

Figure 1: The implemented two-level communication and computation model

3 Sequential Algorithms

The following section gives a short description of all currently implemented functions in their sequential form. They cover most of the functionality needed for medical image processing and can be easily adapted to further requirements.

3.1 Histogramming

The output consists of an array $H[0..k\text{-}1]$ such that $H[i]$ is equal to the number of pixels in the image with gray value i. Here k is the number of possible grayvalues (in most medical images 2^{12}) [2].

3.2 Level-Window

A window is defined through its center and width. All pixels in the image with a gray value i, center - width $\leq i \leq$ center + width, are set to a new gray value.

3.3 Convolution and Morphology

The convolution is performed by shifting a mask over the image. The new gray value of the pixel under the center of the mask is calculated by summing up all the products of the mask values with their underlying image values.

Morphological operators like erosion and dilation also use surrounding voxels for computing the new image. All gray values under the mask are ordered by their value. The n-th value is chosen as the new gray value of the pixel under the center of the mask. The number n (between 1 and the number of mask elements) depends on the respective operation.

3.4 Connected Component Labeling

The connected component labeling works on binary images where all pixels with gray level 0 are assumed to be background and those with gray level 1 are foreground objects. A connected component is a maximal collection of pixels such that a path exists between any pair of pixels in the component. Each component should get its unique label [2].

3.5 Region Growing

Starting from a seed point in the image all neighbor pixels are examined. They belong to the region if they fulfill the given membership criterion. The new region border becomes the starting point for the next growing step. If no new region points are found in a step the whole object is identified.

3.6 Raytracing

A main requirement for image processing applications in the medical field is the visualization of three-dimensional image data in a easy to understand 2D representation. The Heidelberg Raytracing Model [3] is an application-oriented visualization model that has been developed for the use in the medical field. It is based on image-order traversals where the ray is shot from the eye point through each screen pixel (raycasting). In contrast to other applications the Heidelberg Raytracing Model uses two light sources: the first one at 45 degrees to the left of the observer, and the other one which serves for the illumination of the parts of the scenery left in the dark by the first one, in the observer's eye. With this model there are shadows created on the surface of the object which is an important depth clue for viewers evaluating the generated images.

4 Methods

On the process level the program uses the Master/Worker Model. The master process performs all I/O operations, controls the worker processes and executes system global computations in some algorithms. In the current implementation it also provides a text based user interface. In the future this will be replaced by a server interface enabling multiple applications to use the provided functionality. The optimal performance is achieved by running each worker process on a single host.

The same model is used on the thread level where each working thread is bound to one processor. The master thread is responsible for the worker control and performs process global computations. The master *thread* has to share one processor with a working thread. The master *process* can either be executed on the same host as a worker process or on its own. Thereby it is also possible to run the program on a standalone computer. In the following the number of processes will be denoted as N, where process 0 is the master and processes 1 to *N*-1 are the workers.

4.1 Volume Partitioning

A major requirement for efficient parallel algorithms on distributed memory systems is finding a decomposition that minimizes the communication between processes. The latter is especially important if slow networks are used.
Domain decomposition provides the greatest flexibility and scalability in parallel image processing. To apply this method the image is split in several parts that are sent to individual processes.

The most common approach for dividing three-dimensional volumes consists of minimizing the surfaces of the subblocks. An optimal partitioning exists if the number of worker processes is a power of two. The efficiency of this decomposition on MPPs was shown in [6]. Heterogenous clusters impose a loadbalancing problem. A priori, every host has a different computional speed and in addition its workload can vary during runtime. It is only partly possible to adapt the subblock size to the respective performance with this decomposition model.

The parallel implementation of the Heidelberg Raytracing Model using this decomposition model is described in [7]. It is based on the ray dataflow approach [4]. Some limitations compared to its sequential version exist. Only parallel projection is implemented and it omits the shadow casting second light source.

In [4] a raytracing model is introduced dividing the volume into small cells which are randomly distributed onto the existing processors. With this method it is possible to implement a renderer on a MPP using coherency to improve performance. However, with this model the communication cost for other image processing algorithms are enormous, thus preventing efficient and scalable implementations.

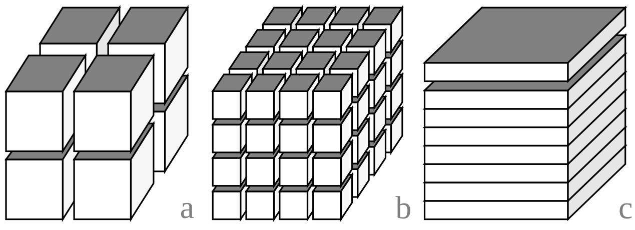

Figure 2: Distribution model for 8 worker processes: a) minimal surfaces b) small cells c) slabs

A third method consists in dividing the volume in *N*-1 slabs. The surface of these subblocks is at least the surface as in the minimal surface approach described above (for $N = 2^n$) but much less than the one with small cells. Other advantages are:

- every process has a maximum of two neighbors simplifying the program logic and reducing the number of connections
- slab exchange can be done directly without any copying because the volume lies linearly in main memory.

4.2 Volume Handling

The volume decomposition is primarily static. Limitations in communication bandwidth require to minimize the volume data transfer between the nodes. During the execution of a function only the border slices needed are transmitted. The results are kept on every host until they are explicitly requested.

To save the current image, the master process has to gather all volume parts from the workers. It is expensive to transfer the whole volume just for viewing intermediate results. The user has other possibilities instead. He can either choose a single volume slice or can calculate an image from any viewpoint using the Heidelberg Raytracing Model. Only two-dimensional images have to be transmitted in both cases.

The master process is responsible for all I/O operations. When loading a data set, the volume is divided into *N*-1 slabs. These slabs are scattered onto the slave processes. Every process maintains a structure which consists of the following information:

- number of existing workers
- own rank
- initial partition (slab owners)
- slice status
- own performance rating and number of processors

Every process gets the number of workers and its own rank by calling MPI_Comm_size() and MPI_Comm_rank() (MPI library functions). The initial partition is a table containing the first and last locally available slice for every worker. This information is received from the master process together with the relative performance and the number of processors in the host. A mutex semaphore is applied to protect the structure from access by different threads at the same time.

The status of every volume slice is also maintained by each process. After initialization only two states exist: DVS_NULL for slices with a different owner and DVS_ORG for locally available slices. If a thread requests a slice its state is set to DVS_REQ. This prevents other threads from requesting the same slice again. After receiving a slice, its state changes to DVS_GOT. Currently this functionality is used only by the raycaster. The other implemented algorithms modify the image, therefore all exchanged slices are invalid when calling the next function.

4.3 Load Balancing

Existing networks often consist of computers with different performance. This depends on the processor, the architecture and the current workload. To get a minimal response time, it is necessary to implement a convenient load balancing model. This is in most cases combined with transferring data between two processes which should be minimized by the initial decomposition.

Our implementation offers two possibilities for load balancing. The first is a simple configuration file like the hostfile in PVM. It contains the relative computational speed for every host compared with other hosts (for one CPU) and the number of CPUs per host. The potential performance can be calculated by multiplying both values. The initial volume slab for every process is computed by the complete volume multiplied with the ratio of its performance to the performance of the whole system. The number of processors every host owns is used to determine the number of worker threads which should be started. Both can be manually changed during runtime. The changes become valid when loading the next volume.

Changing the task size of an algorithm during runtime is worthwhile only if the communication plus the computation cost on the other hosts are less then the local computation cost. That seems to be possible only using networks faster then a 10Mbit Ethernet. Nevertheless, load balancing is implemented in the raycaster. In this case the task size is defined by the number of scanlines every host has to compute. The viewplane decomposition is distributed the same way as the volume described in the last section. This is not changed during calculation of an image but between two of them. The work every process gets for computing the next images is based on the response time of the current one. That means, the first process finishing its work has to do more work in the next image while the last process gets less. A possible volume redistribution could also be based on this information.

4.4 Interaction between POSIX Threads and MPI

Our goal was to develop a universal communication and control model which can be used to efficiently implement most of our frequently used image processing functions. A reason for preferring MPI over PVM is described in [5] as follows: „The desired interaction of MPI with threads is that concurrent threads be all allowed to execute MPI calls, and calls be reentrant; a blocking MPI call blocks only the invoking thread, allowing the scheduling of another thread.“

We performed some tests to review this statement. The behavior of all communication functions in MPICH 1.1.0 indicates that they are protected by a semaphore. Therefore, if in a process multiple threads call blocking MPI functions and wait for different messages, care has to be taken to avoid deadlocks. In this case only one receive operation is performed, blocking this thread. All other threads are also blocked but wait on the semaphore. They don’t get the chance to receive their messages until an opposite message for the only receiving thread arrives. Such a limitation is not acceptable for more complex systems.

A simple modification in the above model allows a correct receiving of all messages. Receiving is performed just in one thread. The other threads are waiting for their own semaphore, each. Depending on the incoming message the adequate semaphore is posted from the receiver. With this model it is not yet possible to send messages independently from receiving.

To enable sending, the blocking receive had to be replaced with a nonblocking test operation. To avoid a permanent polling if no message is available, the receiver indicates to the operating system that the thread may be suspended using the sched_yield() function.

```
do {
 sched_yield();
 sem_wait( semaphore );
 MPI_Iprobe( MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &flag, &stat );
 sem_post( semaphore );
} while( !flag );
MPI_Recv( something )
```

All MPI calls in multithreaded functions are protected by an additional semaphore avoiding problems with nonthread-safe implementations.

This solution is not totally satisfactory but usable with all algorithms. A maximum independence from communication tasks and working tasks can be obtained. In our implementation the usage will be limited to the raycaster where the number and kind of requests is not known before. The other image processing functions have a predictable communication pattern, so this solution is not needed.

5 Parallel Implementation

5.1 Histogramming

At first the locally available volume is divided into P_i slabs, where *P* is the number of processors in the *i*-th host. Every thread gets the address of its start- and endpoint as parameter so that they can independently calculate their histograms. After finishing their work, the histograms are merged locally in the first step. The resulting *N*-1 histogram parts are sent to the master process which computes the final result. A hierarchical merging has not been implemented yet.

5.2 Level Window

After receiving the parameters from the master process (minimal, maximal and new gray value) threads were spawned like in histogramming. Every thread performs its work independently.

5.3 Convolution and Morphology

The master process sends the convolution mask to be used to all workers. With this approach a user application can generate every desired mask and is not limited to implemented ones. Depending on the mask size, every process sends in a nonblocking way its border slices to its direct neighbors. The threads perform the convolution on slices.
A working pool is created with all slices that have not yet been processed excluding border slices. From this pool every thread gets its work. The main thread waits for border slices arriving from its neighbors. On arrival it starts a thread for processing these slices. Finally it waits for all threads to terminate. The time needed for communication is hidden by processing the available data first.
The morphological operators are implemented in the same way as the convolution.

5.4 Connected Components

The local volume is divided into P_i slabs. The first step is to perform a run length encoding of the binary image. Every thread labels its subvolume independently, starting with the first one. The computation of the global labels is processed in two steps. Every thread compares its labelled border slices with the local neighbor slices. The detected equivalences are stored in a table local to every thread. By posting a semaphore they signalize the master thread the end of this task. The master thread analyses the equivalence tables and generates preliminary labels. Every process sends the border slices (with the preliminary labels) to its neighbor processes. Now the second equivalence detection is performed. The resulting equivalence tables are sent to the master process where they are analysed and the final labels are computed. These are sent back to all worker processes to perform a last relabeling.

5.5 Region Growing

Most region growing algorithms are inherently sequential. Our parallel approach is based on the connected components algorithm. The first step is to create a binary image depending on a chosen region growing criterion on which the connected component algorithm is performed. The only variation is an extra step in the calculation of the final labels. The process holding the seed point sends an additional message with its temporary label to the master process. Before sending the final labels back to the slave processes they are analysed if they belong to the same component like the seed point, if not the label changes to 0.

5.6 Raytracing

The master process sends all needed viewing parameters and a task to every worker process. The tasks are scanline numbers of the resulting image every process has to calculate. For the first image they are partitioned like the slabs in the volume. For parallel projection it is only necessary to exchange one border slice to be able to perform a trilinear interpolation. Moving around the volume in a circle is then possible without further data exchange. Perspective viewing needs a larger effort regarding communication. However after exchanging the data for computing the first image no other communication is needed when moving around the volume at the same distance. The disadvantage of this model is the huge communication effort to produce an image looking from the top or bottom. The whole volume has to be transferred to all processes to do this.

When receiving the first message for raytracing an image each worker process starts one thread. The main thread handles all incoming messages. These might be requests for slices, slices themselves, the parameter for the next image or the end message. Request messages are replied by sending the needed slices back. Arriving slices are received in the volume, signalizing their availability to the other thread. A signal is also sent if the next image has to be computed or the function has to be stopped. First the second thread computes all needed slices and requests them if they are not locally available. During the communication, for all pixels on the viewplane the entry and exit points are computed and, if not already done so, the worker threads are started. When receiving the signal that the needed volume has arrived, all worker threads that wait for a semaphore are resumed. They signalize their end to the starting thread which sends the results to the master process. The master process waits for all worker results to combine and save them.

6 Results

The raytracing time decreases from 10 seconds down to 3.5 seconds per frame on four different Pentium PCs (90, 100, 133 MHz, 1 and 2 CPUs). All other implemented algorithms produce similiar results. The use of more than six PCs does not lead to any additional speedup.

7 Conclusion

It is possible to combine distributed and shared memory hardware to calculate problems faster. The use of MPI and POSIX Threads allows interoperability in heterogenous environments. However, the interaction between these two libraries does not yet comply to all requirements.

With the proposed system we achieved the integration of several image processing and visualization algorithms into one parallelization scheme avoiding repeated redistribution of large volume data.The reduced processing time is important for the acceptance in clinical use and generally for interactive systems.

8 Acknowledgements

This research was performed at the Department of Medical and Biological Informatics at the German Cancer Research Center. The work was funded by the german BMBF (Projekt EVIMED FKZ 01 IR 406 C 3) and the Deutsche Forschungsgemeinschaft (SFB 414).

9 References

[1] Geist GA, Kohl JA, Papadopoulos PM: „PVM and MPI: a Comparison of Features”, Calculateurs Paralleles Vol. 8 No. 2, 1996.

[2] Bader DA, JáJá J: „Parallel Algorithms for Image Histogramming and Connected Components with an Experimental Study“, Technical Report CS-TR-338, Institute for Advanced Computer Studies, University of Maryland, 1994.

[3] Meinzer HP, Meetz K, Scheppelmann D, Engelmann U, Baur HJ: „The Heidelberg Ray Tracing Model“, IEEE Computer Graphics & Applications, Nov. 1991, pp. 34-43.

[4] Law A, Yagel R: „Exploiting Spatial, Ray, and Frame Coherency for Efficient Parallel Volume Rendering“, Proceedings of GRAPHICON’96, Vol. 2, Saint-Petersburg, Russia, July 1996, pp. 93-101.

[5] Snir M, Otto S, Huss-Lederman S, Walker DW, Dongarra J: „MPI: The Complete Reference“, The MIT Press, Cambridge, Massachusetts, London, England, 1996.

[6] Kochner B: „Datenparallele Beschleunigung ausgewählter Bildverarbeitungsalgorithmen auf MIMD-Rechnern“, Technical Report TR 88, Abt. Medizinische und Biologische Informatik, DKFZ Heidelberg, 1996.

[7] Niebsch R: „Erweiterung und datenparallele Beschleunigung des Heidelberger Raycasting-Verfahrens auf MIMD-Rechnern“, Technical Report TR 83, Abt. Medizinische und Biologische Informatik, DKFZ Heidelberg, 1996.