A Distributed Shared Memory System Oriented to Volume Visualisation

Marcelo Knörich Zuffo
Roseli de Deus Lopes
Volnys Borges Bernal

[mkzuffo, roseli, volnys]@lsi.usp.br

Laboratório de Sistemas Integráveis
Departamento de Engenharia Eletrônica
Escola Politécnica da Universidade de São Paulo
Av. Prof. Luciano Gualberto, trav.3 n. 158, 05508-900 São Paulo, SP, Brazil

Abstract: In this paper we propose the use of the Distributed Shared Memory (DSM) paradigm for parallel volume visualisation. The goal is to offer a comprehensive and portable programming model that exploits the parallelism and data coherency commonly found in volume visualisation. Our approach is based on the development of a library layer that offers a simple and straightforward programming interface. This programming model allows users to implement volume visualisation programs that run on sequential and parallel computing environments. The library provides a contiguous shared memory space, synchronisation, task scheduling, data spatial partitioning and I/O services. In order to achieve better performance, users should tune their programs through re-configurable parameters such as loop scheduling, volume data access mode, block shape, and cache size. Preliminary results are presented based on a library implementation that runs over IEEE POSIX threads, Message Passing Interface (MPI) and UNIX interfaces on an MEIKO-CS2 Multicomputer with 10 processing elements.

Key words: Volume Visualisation, Parallel and Distributed Processing, Distributed Shared Memory.

1 Introduction

Currently, scientific visualisation is an essential tool for several knowledge fields, allowing professionals and scientists to enhance the analysis, understanding and learning of natural phenomena, through simulations and instrumentation that generate volumetric data. Volume Visualisation is concerned with the representation, manipulation and rendering of such volumetric data [2], and is an important set of techniques and algorithms for many scientific visualisation applications.

Parallel computing systems are becoming a common execution environment for the increasing computing demand of typical scientific applications. These systems are available as Shared Memory Parallel Computers (multiprocessors), Distributed Memory Parallel Computers (multicomputers) or clusters of workstations connected by high-speed networks.

We propose the use of the Distributed Share Memory (DSM) paradigm for implementing parallel volume visualisation programs for multicomputers and clusters of workstations.

The DSM paradigm offers a programming model that makes the physically distributed memory accessible to all processing elements. This paradigm offers a general and convenient programming model that enables simple data sharing through the uniform mechanism of reading and writing data structures in the distributed memory.

We have designed and implemented a DSM system oriented to volume visualisation applications. This system takes advantage of the strong data coherence and parallelism commonly found in many volume visualisation algorithms. The proposed DSM system is a library layer interface called PVV[1] (Parallel Volume Visualisation). The PVV library was designed for implicit and explicit parallel programming of volume visualisation supporting a wide range of volume rendering algorithms [4], 2D and 3D image processing algorithms.

2 Previous Work

Corrie and Mackerras [3] proposed the use of DSM for volume rendering, they implemented a volumetric ray-casting algorithm based on a read-only DSM for an AP1000 Multicomputer. The authors considered the worker-farm paradigm using an image-space task sub-division and to achieve good load balancing the image was subdivided in equal size squares. A speed-up of 108.2 for 127 processors was reported. The authors pointed out the importance of correct cache size to avoid page thrashing.

Some authors implemented volume rendering algorithms in shared memory parallel systems. Nieh and Levoy [8] implemented the ray-casting algorithm on the DASH shared memory parallel computer. The authors reported the advantages of programming shared memory systems and the need for good load balancing. Lacroute [5] implemented a Shear-warp algorithm on a Silicon Graphics Challenge shared memory parallel machine. A performance of 10 frames a second for a 256^3 volume was achieved using dynamic load balancing and a task partitioning strategy to avoid synchronisation events. The main conclusion of using shared memory parallel computers is that the communication costs and data redistribution do not dominate rendering time, and that cache locality requirements impose a limit on scalability in such parallel machines.

The performance of volume rendering methods in mesh-connected Multicomputers was studied by Neumann [7]. The author considered independently the parallel algorithms and the volume rendering methods, and pointed out how the parallel algorithm has a major impact on the communication requirements among processing elements and on the overall performance of the system. Neumann observed that the communication cost in many parallel volume rendering

---

[1]This project was funded by The European Commission, program Information Technologies for Developing Countries Contract ITDC-225, and Fundação para o Amparo à Pesquisa do Estado de São Paulo, FAPESP Contract 95-2747-6.

algorithms decreases as the data and system sizes grow, suggesting scalability of such algorithms for big datasets and massively parallel computers.

3 System Organisation and Design

To guarantee program portability, the PVV library offers a programming interface that is independent from the parallel system architecture. To guarantee system portability, across different parallel machines, the DSM sub-system was implemented in software, and the PVV library was implemented on top of parallel programming standards such as the Message Passing Interface (MPI), IEEE POSIX Threads (Pthreads) and Unix standard interfaces and system calls.

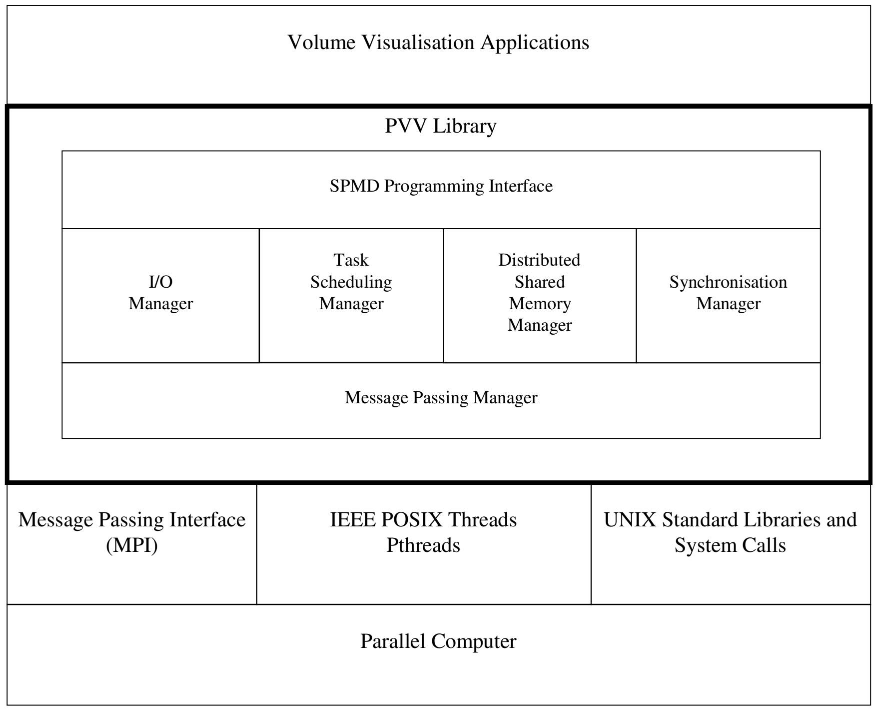


Figure 1- PVV Library Organisation

The PVV library offers the following functionality:

- Data representation and I/O;
- Transparent manipulation of shared data such as volumes, images and arrays[2];
- Explicit parallel programming constructions such as node identification and synchronisation;
- Implicit parallel programming constructions such as loop management and task scheduling control.

Figure 1 shows the PVV library organisation.

The I/O Manager is a thread responsible for the I/O calls in the PVV library. This manager supports the ***`PVV_read_volume`*** and ***`PVV_write_volume`*** calls. These calls guarantee I/O operations for a distributed volume among the processing elements.

The Task Scheduling Manager is responsible for task distribution among the processing elements. Section 4.4 describes the task scheduling strategies adopted.

[2] In the PVV library, there are functions and macros to represent and manipulate volumes (3D), images (2D) and arrays (1D) in the same fashion. To avoid repetition we will refer only to the functionality related with volumes.

The DSM manager is responsible for offering a transparent and contiguous addressing space on top of the physical distributed memory of the processing elements. The DSM is described in section 4.

The synchronisation manager is a thread responsible to synchronise activities among the processing elements. To perform explicit synchronisation PVV offers the call ***PVV_sync()***, Program 2 shows an example of such call.

The Message Passing Manager is a thread responsible for the managing of all messages arriving from the other processing elements. The incoming messages could be synchronisation requests, block migration, directory updates, I/O requests, and task assignments.

3.1 The Programming Interface

The programming interface was designed to ensure program portability across different MIMD parallel computers, including distributed memory systems, clusters of workstations and shared memory parallel systems.

The PVV library provides a single and shared address space for volumes (3D), images (2D) and arrays (1D), facilitating the programming effort avoiding for the user all problems related with parallel volume visualisation programming such as data partitioning and task scheduling.

The interface is based on an ***ANSI-C*** library. The programs run in parallel using the Single Program Multiple Data paradigm (SPMD), where the same program is loaded and executed in the different processing elements.

3.1.1 Implicit Parallelism

For many volume visualisation programs, such as ray-casting and 3D convolution filtering, users can take advantage of intrinsic volume visualisation parallelism and write implicit parallel programs with some PVV constructions.

```
#include <pvv.h>

int main(int argc, char *argv[])
{
PVV_volume_t       vol = NULL;
PVV_byte_t         voxel;

PVV_init(&argc, &argv); /* Init PVV Managers                      */

PVV_SET_SUBVOLUME_BLOCK( 8, 8, 8);
vol = PVV_create_volume( 256, 256, 256, PVV_BYTE_ID);

PVV_SET_VOXEL_INTERVAL  ( vol, 20, 20, 20, 100, 100, 100);
PVV_SET_SCHEDULING      ( vol, PVV_CONTIGUOUS);
PVV_FOR_EACH_VOXEL_BEGIN (vol)
      { /* Some processing that will be applied to the current voxel */
      .
      .
      .
      PVV_PUT_CURRENT_VOXEL( vol, voxel, PVV_byte_t);
      }
PVV_FOR_EACH_VOXEL_END

PVV_write_volume( vol, "filename", PWF);

PVV_finalize();          /* Finish PVV Managers                    */
}
```

Program 1 – Creating, Manipulating and Writing Volumes With Implicit Parallelism

Through the use of the SPMD programming paradigm, PVV programs can be executed sequentially and without any modification can be executed in parallel.

Program 1 shows a simple PVV program with implicit parallelism. This program creates a 256^3 volume subdivided in 8^3 sub-volumes. Some processing is done in the sub-volume defined by the *(20, 20, 20)* and *(100, 100, 100)* vertices, finally the volume is write to the disk. The loop ***PVV_FOR_EACH_VOXEL*** is executed in parallel with ***PVV_CONTIGUOUS*** scheduling. A better description of this PVV library functionality will be described in further sections.

3.1.2 Explicit Parallelism

In some particular cases, the parallelism of volume visualisation algorithms is not so straightforward, and these algorithms should be implemented in parallel using explicit SPMD parallel constructions. In this situation, the user should take care of the program execution flow, assigning explicitly the computation that will executed in parallel for each processing element, and taking care of the synchronisation among the processing elements.

Program 2 shows an example of explicit parallelism. In this program a 256^3 volume is created, processing element 0 will initialise the volume sequentially with 0, while the remaining processing elements will perform another tasks in parallel. In the end of the switch all processing elements are synchronised.

```
#include <pvv.h>

int main(int argc, char *argv[])
{
PVV_volume_t      vol = NULL;
PVV_byte_t        voxel;

PVV_init(&argc, &argv); /* Init PVV Managers                */

vol = PVV_create_volume(256, 256, 256, PVV_BYTE_ID);

switch(PVV_MY_NODE)
      {
      case 0: /* Init volume vol sequentially in Node 0     */
            for(k=0; k<vol->resK; k++)
              for(j=0; j<vol->resJ; j++)
                for(i=0; i<vol->resI; i++);
                  {
                  voxel = 0;
                  PVV_PUT_VOXEL(vol, i, j, k, voxel, PVV_byte_t);
                  }
      break;

      case 1: /* Do something in Node 1                     */
            .
            .
      break;
            .
            .
            }
PVV_sync();       /* Synchronise all nodes                  */

PVV_finalize();   /* Finish PVV Managers                    */
}
```

Program 2 – Explicit Parallel Programming

4 A DSM Oriented to Volume Visualisation

We adopted a fixed block size DSM; each block holds a group of voxels. The block has a spatial organisation in order to exploit coherence. The consistency model adopted is the sequential model.

4.1 *The Block Management Protocol*

Each shared data structure such as a volume, image or array can be partitioned in a set of regular shape blocks. The block size and shape can be a user-defined property of a data structure.

The block management protocol is based on the Single Reader/Single Writer Algorithm (SRSW) [1]. The blocks are distributed among the processing elements. Each processing element is responsible to manage all access requests related to the local blocks from other processing elements.

When a shared data structure is created, it is partitioned in blocks that are distributed among the processing elements in the **Permanent Block List**. Upon request the blocks can migrate to another processing element and are stored in the **Temporary Block List**. This kind of DSM protocol is called **hot potato** [1]. Information about the current block location is held in the **Block Directory**. We choose this protocol due to the considerable amount of volumetric data to be manipulated and the little block sharing observed in many parallel volume visualisation algorithms.

For read/write volumes, only one valid copy of the block exists at any time, and the block migrates through the processing elements. For read-only volumes many copies of a block could be replicated among the processing elements. By default all volumes are read/write. The PVV library allows users to any time change the volume access mode to read-only or read-write.

The Temporary Block List acts as a cache and the replacement block policy is the Not Recently Used (NRU) algorithm. The advantage of the NRU algorithm is its smaller computational cost compared with the Least Recently Used (LRU) algorithm. The Temporary Block List size, in blocks, is a user-defined property of a shared data structure and its correct size definition for different algorithms is important to avoid block thrashing. The DSM granularity is the block.

Figure 2 shows all possible block transactions among processing elements. Which are the following 5 situations:

- **Situation A:** A processing element (requester) asks for a block from its block server (provider). The block migrates to the Temporary List (TL). The Owner Block Directory is updated;
- **Situation B:** A processing element replaces a dirty block in its Temporary List, this block is writes-back (migrates) to the Permanent List (PL) of the Provider. The Provider Block Directory is updated;
- **Situation C:** A processing element replaces a clean block in its Temporary List, this block is frees. The Provider Block Directory is updated;
- **Situation D:** A processing element requests a block from its Provider. The Provider redirects the requisition to another processing element that owns the block. The block migrates. The Provider Block Directory is updated;
- **Situation E:** A processing element requests a block to its Provider. The Provider redirects the requisition to an Owner that did not own anymore the block (False Owner), the False Owner redirect the requisition back to the Provider, that redirects the requisition to another processing element that owns the block. The block migrates. The Provider Block Directory is updated.

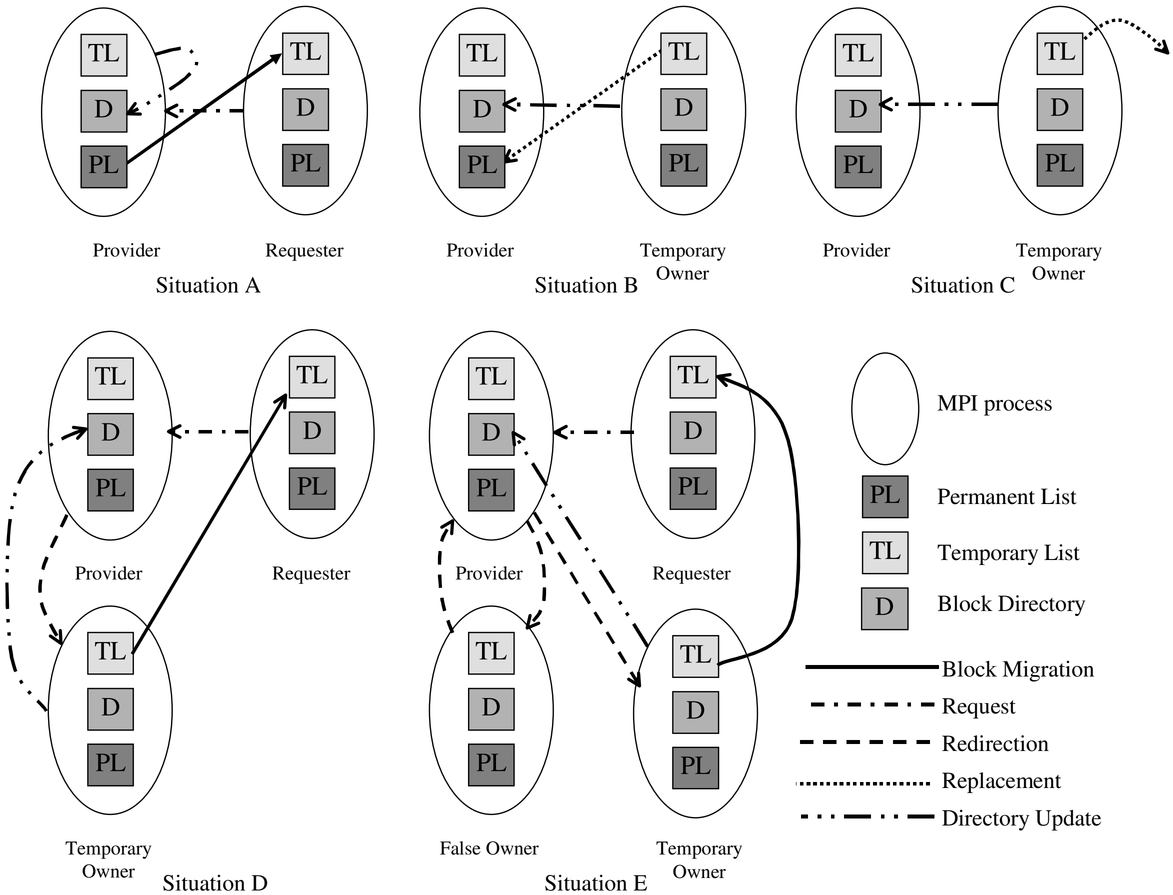


Figure 2 – Block Transaction Map

4.2 Block Access and Block States

To provide transparent data access to shared data structures the PVV library offers the ***PVV_PUT_VOXEL*** and ***PVV_GET_VOXEL*** constructions[3], these constructions are built on top of a more general constructions ***PVV_PUT_DATA*** and ***PVV_GET_DATA***.. To enhance performance these constructions are implemented with macros and they require as arguments, the volume pointer, the voxel position *(i, j, k)* in the volume, a variable to be write or read and the variable type to do proper alignment.

PVV_block_replacement frees a block from the Temporary List (TL) when this block is clean, ***PVV_block_writeback*** writes back a block in its processing element provider when a block is dirty.

The block status diagram is showed in Figure 3.

There are only three states, the contiguous line represents state transitions among the same processing element. The dashed line represents state transitions among different processing elements, in this case the block migrates with its current state.

[3] There are the same constructions for images: ***PVV_PUT_PIXEL*** and ***PVV_GET_PIXEL,*** and for arrays: ***PVV_PUT_AXEL*** and ***PVV_GET_AXEL***.

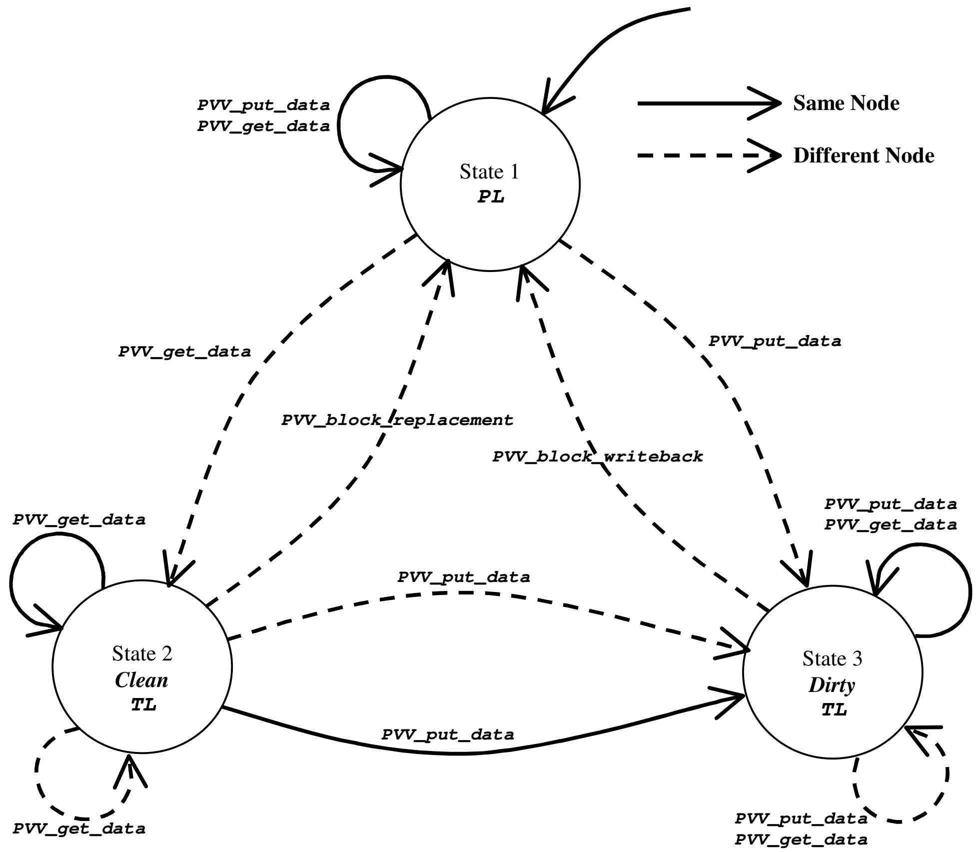


Figure 3 – Block Status Diagram

4.3 Data Locality and Partitioning

Data locality is an inherent property of a considerable set of volume visualisation algorithms. In order to take advantage of this property, the PVV library offers a mechanism to maintain data locality by the spatial partitioning of the volumetric data in blocks. The blocks could be slices, columns and cubes (Table 1).

This approach has many advantages, some of them are:

- Due to the DSM management protocol and the spatial dependency of some volume visualisation algorithms (example ray-casting), an optimal block shape can be selected in order to enhance data coherence;
- Due to the interconnection network characteristics that change for each different parallel system, an optimal block size can be selected to minimise network latency and contention;
- Due to the memory hierarchy, an optimal block size can be chosen to fit the best primary and secondary cache performance.

Sub-volume shape	Sub-volume resolution (voxels)	Voxel size (bytes)	Block size (bytes)
Cube	16x16x16	1	4096
Column	256x4x4	1	4096
Slice	64x64x1	1	4096

Table 1 – Examples of Block Size and Shape

4.3.1 Table Oriented Voxel Addressing

We propose a mechanism for regular decomposition of Cartesian volumes by the use of addressing tables. The advantages of addressing tables are:

- Fixed addressing cost for different block shapes;
- Same access interface for different block shapes;
- Faster method, when compared with arithmetic addressing evaluation;
- Allows fast incremental addressing;

For a volume, with resolution *(resI, resJ, resK),* we would like to evaluate a given voxel position *(i, j, k)* address ***d***. The addressing cost is fixed and defined by:

$$d(i, j, k) = I[i] + J[j] + K[k] \tag{1}$$

We will subdivide the regular Cartesian volume into a set of regular and homogeneous blocks (sub-volumes). We can arbitrarily choose the block resolution $(resI_L, resJ_L, resK_L)$. The volume resolution in terms of sub-volumes $(resI_H, resJ_H, resK_H)$ is given by:

$$\begin{aligned} resI_H &= \left\lceil \frac{resI}{resI_L} \right\rceil \\ resJ_H &= \left\lceil \frac{resJ}{resJ_L} \right\rceil \\ resK_H &= \left\lceil \frac{resK}{resK_L} \right\rceil \end{aligned} \tag{2}$$

The voxel address *(i, j, k)* is decomposed in the High Address (i_H, j_H, k_H) that corresponds to the block index in the volume and the Low Address (i_L, j_L, k_L) that corresponds to the voxel index in the block. The voxel address could be evaluated by:

$$\begin{aligned} i &= i_L + i_H \cdot resI_L \\ j &= j_L + j_H \cdot resJ_L \\ k &= k_L + k_H \cdot resK_L \end{aligned} \tag{3}$$

Each sub-volume has a regular size and shape defining a Block with B voxels:

$$B = resI_L \cdot resJ_L \cdot resK_L \tag{4}$$

We can evaluate the address by:

$$\begin{aligned} d(i, j, k) = {} & i_L + j_L \cdot resI_L + k_L \cdot resI_L \cdot resJ_L + \\ & (i_H + j_H \cdot resI_H + k_H \cdot resI_H \cdot resJ_H) \cdot B \end{aligned} \tag{5}$$

We can rewrite it as tables:

$$\begin{aligned} I[i] &= i_L + B \cdot i_H \\ J[j] &= j_L \cdot resI_L + B \cdot j_H \cdot resI_H \\ K[k] &= k_L \cdot resI_L \cdot resJ_L + B \cdot k_H \cdot resI_H \cdot resJ_H \end{aligned} \tag{6}$$

We can observe that the total addressing cost could be reduced from 9 multiplications and 3 sums to 3 table accesses and 2 sums, this cost is fixed and independent from the block size and shape.

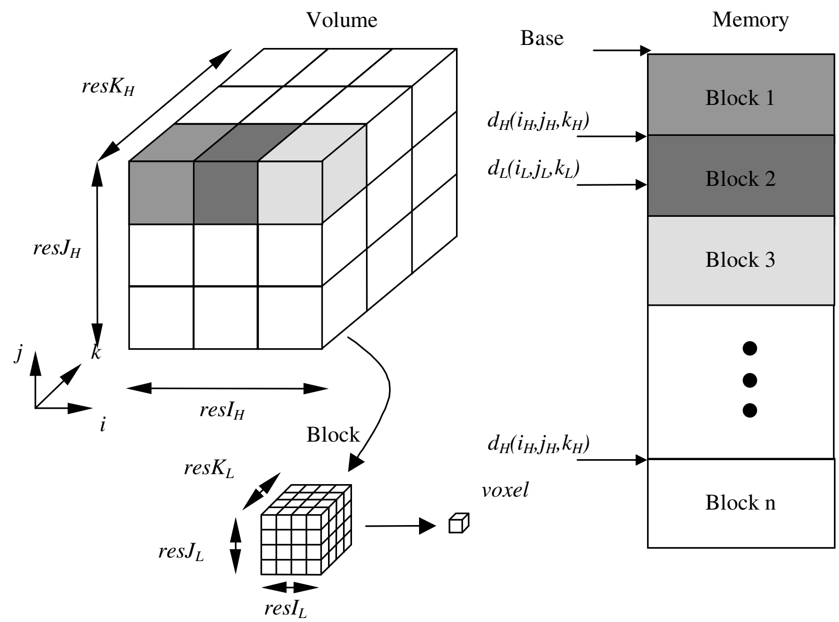


Figure 4 Regular block addressing

4.4 Parallel Loops and Task Scheduling

In many volume visualisation algorithms, there is quite often the need for some independent processing in all voxels in a given voxel interval from a volume. To do that in parallel users can write loops that can be parallelized using standard parallel processing techniques. The PVV library offers the construction ***PVV_FOR_EACH_VOXEL*** to encapsulate this kind of common need.

PVV_FOR_EACH_VOXEL is similar to the ***FORALL*** construction in Fortran 90. Program 1, shows an example of the use of the ***PVV_FOR_EACH_VOXEL*** construction.

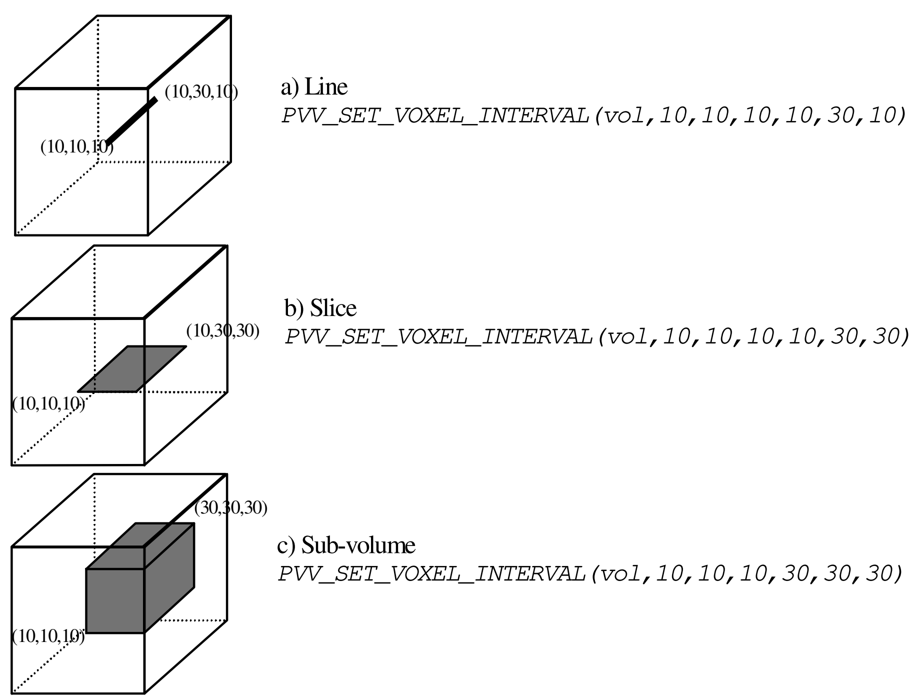


Figure 5 – ***PVV_SET_VOXEL_INTERVAL*** examples

The ***PVV_FOR_EACH_VOXEL*** construction defines a piece of program that should be applied for each voxel in a given interval, this piece of program is called **Virtual Voxel Machine (VVM).** The Virtual Voxel Machine has the following properties:

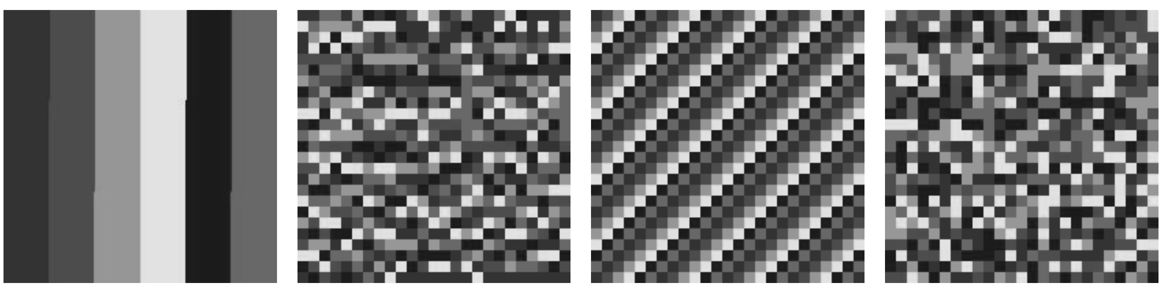
- One VVM is assigned to each voxel. In the ***PVV_FOR_EACH_VOXEL*** loop the current voxel is identified by the Current Voxel Position CVA register. Since the current voxel position is implicitly defined in a ***PVV_FOR_EACH_VOXEL*** loop, the special constructions ***PVV_PUT_CURRENT_VOXEL*** and ***PVV_GET_CURRENT_VOXEL*** are used to perform read/write operations respectively;
- VVMs executions are independent from each other; VVMs are grouped in **tasks**, and to enhance coherence and locality, the tasks have the same block size and shape.

The VVMs executions could be set for a given interval inside a volume. The voxel interval (line, slice or cube) is defined by the ***PVV_SET_VOXEL_INTERVAL*** construction, if the interval is not defined the program will be applied to all volume voxels, examples of such construction are shown in

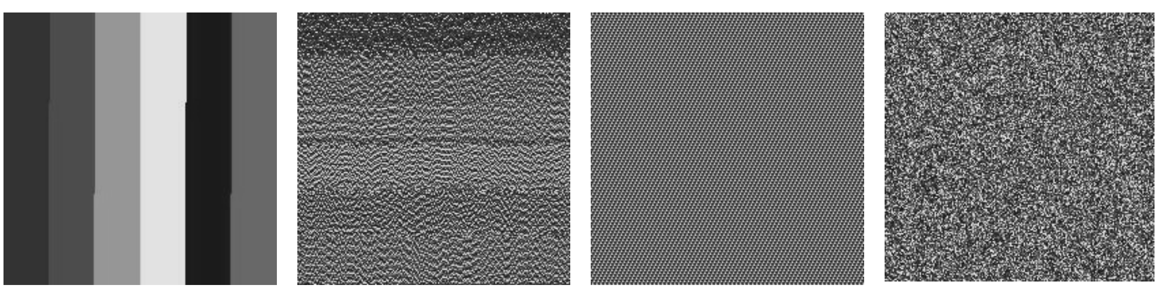
- Figure 5.

Strategies for task scheduling fall into three main classes: Static, Dynamic and Random. The scheduling strategies supported by PVV, showed in Figure 6, are:

- ***PVV_CONTIGUOUS:*** In this static scheduling strategy tasks are grouped in chunks that are equally distributed among the processing elements;
- ***PVV_INTERLEAVED:*** the tasks are uniformly distributed among the processing elements;
- ***PVV_DYNAMIC:*** the tasks are distributed upon requests from the processing elements;
- ***PVV_RANDOM:*** the tasks are randomly distributed among the processing elements.



a) Contiguous 8x8 b) Dynamic 8x8 c) Interleaved 8x8 d) Random 8x8



a) Contiguous 1x1 b) Dynamic 1x1 c) Interleaved 1x1 d) Random 1x1

Figure 6 – Scheduling strategies supported by PVV

Figure 6 shows the supported scheduling strategies maps considering 6 processing elements with different colours. We used two block sizes 1x1 pixels and 8x8 pixels

4.5 Tuning for Performance

The use of DSM systems decrease a lot the user programming effort for the implementation of parallel volume visualisation algorithms, however tuning is an essential step to achieve good performance in different parallel systems.

The proposed DSM system has some tuneable parameters that should be configured for each particular algorithm and parallel system.

The tuneable parameters are: **Block Size, Block Size, Task Scheduling, Temporary List Size** and **Data Structure Access Mode.**

The block size parameter is a very important tuneable parameter. The block size is proportional to the task granularity and DSM granularity. Since it affects directly the communications costs and load balancing. We can consider that the communication cost in distributed memory parallel systems is proportional to the interconnection network latency and throughput. Latency is a constant communication cost for different message sizes, while throughput is almost linear proportional to the block size. For small block sizes the latency is dominant in the communication cost, for big block sizes the network contention is dominant in the communication cost. Since the task granularity is defined also by the block size, small block sizes offer a better load balancing among the processing elements. The block size should be a trade-off among latency-related costs, load balancing and network contention communication costs.

The Block shape is a tuneable parameter directly related with the data locality. Since the majority of volume visualisation algorithms have strong spatial coherency, a good choice of block shape can enhance performance by the increase of local accesses, avoiding block trashing and taking advantage of the memory hierarchy. The block size and shape is a property of each data structure supported by the PVV DSM and it can be set by the ***`PVV_SET_SUBVOLUME_BLOCK()`*** construction (Program 1).

The task scheduling parameter is related with load balancing among the processing elements and data locality. The scheduling strategy is set by the ***`PVV_SET_SCHEDULING()`*** construction.

The Data Structure Access Mode allows users to change the block management strategy for replication, when a data structure is read-only on a given program interval. The Data Structure Access Mode is set by the ***`PVV_SET_MODE()`*** construction.

5 Preliminarily Results

We implemented a version of our proposed DSM on a MEIKO-CS2 Multicomputer. The MEIKO-CS2 is based on a Fat-Tree interconnection network with 100Mbit/s bi-directional link per processing element. Each processing element is a Hypersparc 100MHz microprocessor with a 128 Mbytes of RAM. The machine used has 10 processing elements. The operating system is Solaris 2.3. We used the Argonne National Labs public domain MPI implementation, MPICH v1.0.13 available at http://www.mcs.anl.gov/ and the public domain IEEE POSIX Pthreads library available at: http://www.mit.edu:8001/people/proven/Pthreads.html.

We implemented and evaluated two volume visualisation programs: Volume Write/Read and Volume Ray-Casting Rendering.

5.1 Volume Write/Read Program

Volume Write/Read is a very simple program with two ***`PVV_FOR_EACH_VOXEL`*** loops. The first loop is responsible for writing a value in each voxel position, and the second loop is responsible for reading and checking the value in each voxel position. We used a 256^3 volume subdivided into 16^3 blocks. The scheduling strategy adopted was ***`PVV_CONTIGUOUS`***.

Figure 7 shows the Speed-up and Efficiency for 10 processing elements of the Read/Write program. We can observe that the Efficiency is near 50% for the Write loop. That happens because the block distribution does not match the scheduling, and block migration among the processing elements occurs. The Efficiency for the Read loop is almost 100% since all blocks are available in the local processing elements.

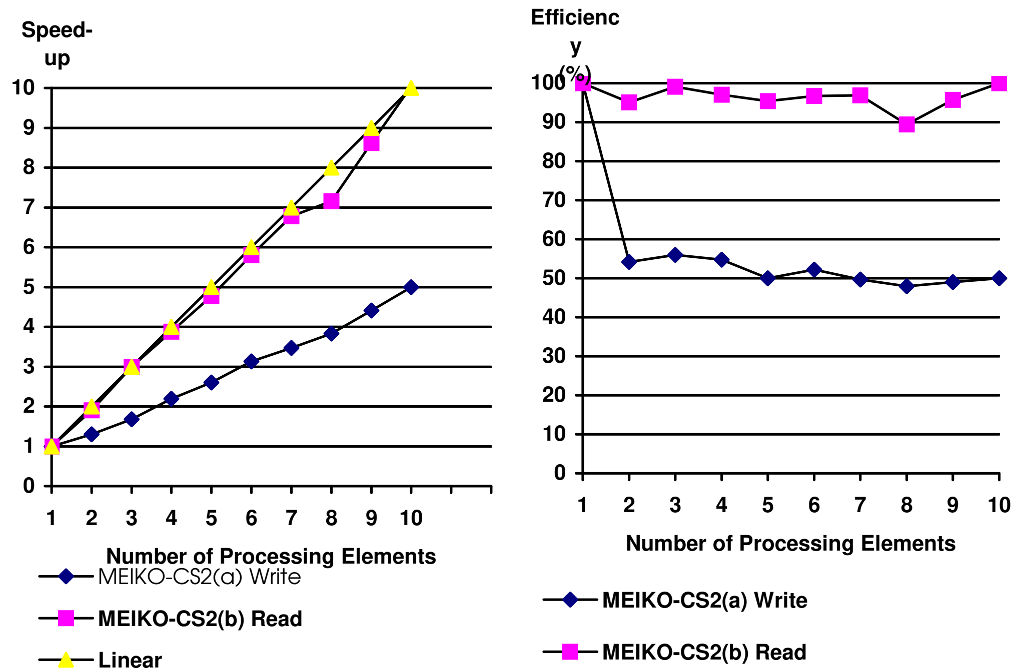


Figure 7 – Speed-up and efficiency for volume Write/Read Program

5.2 Volume Ray-Casting Program

We implemented the traditional ray-casting volume rendering algorithm. The volume resolution used was 256^3 and the block size was 16^3. The image resolution for the measurements is 512^2 subdivided in 32^2 blocks. The program has two main loops, the first loop ***PVV_FOR_EACH_VOXEL*** is used for the volume classification (RGBO mapping), and the second loop ***PVV_FOR_EACH_PIXEL*** is used for casting rays to the volume. The scheduling strategy for both loops is ***PVV_INTERLEAVED***.

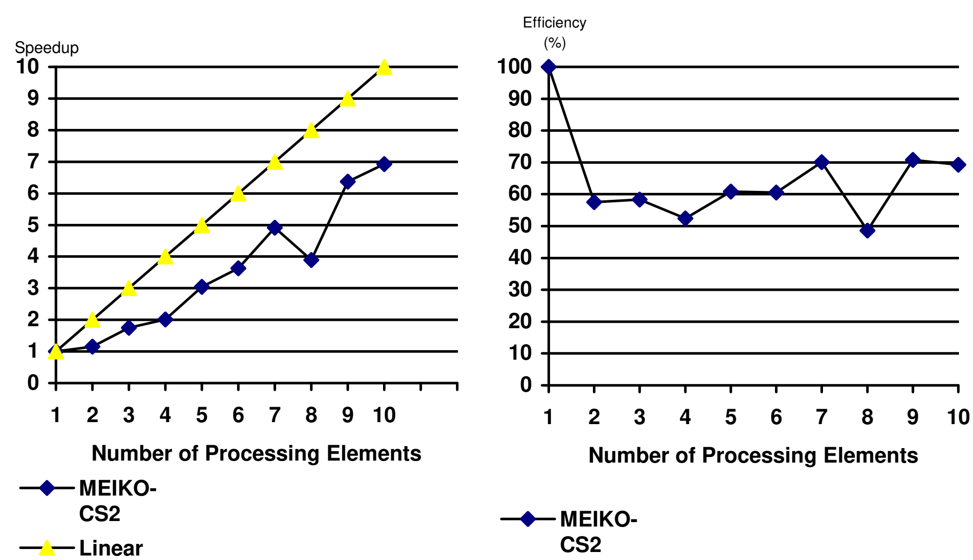


Figure 8 – Speed-up and Efficiency per Number of Processors for the Volume Ray-Casting Program

Figure 8 shows the speed-up and efficiency for the Volume Ray-Casting program. We can observe that the efficiency raised around 10% from 1 to 10 processing elements. Figure 9 shows the memory demand for each processing elements, in this case we keep the Temporary Block List size decreasing with the number of blocks. In Figure 9 we observe also the number of Page-faults per processing elements for the volume and for the image, an interesting observation is that the number of page-faults tends to decrease with the number of processing elements, suggesting scalability.

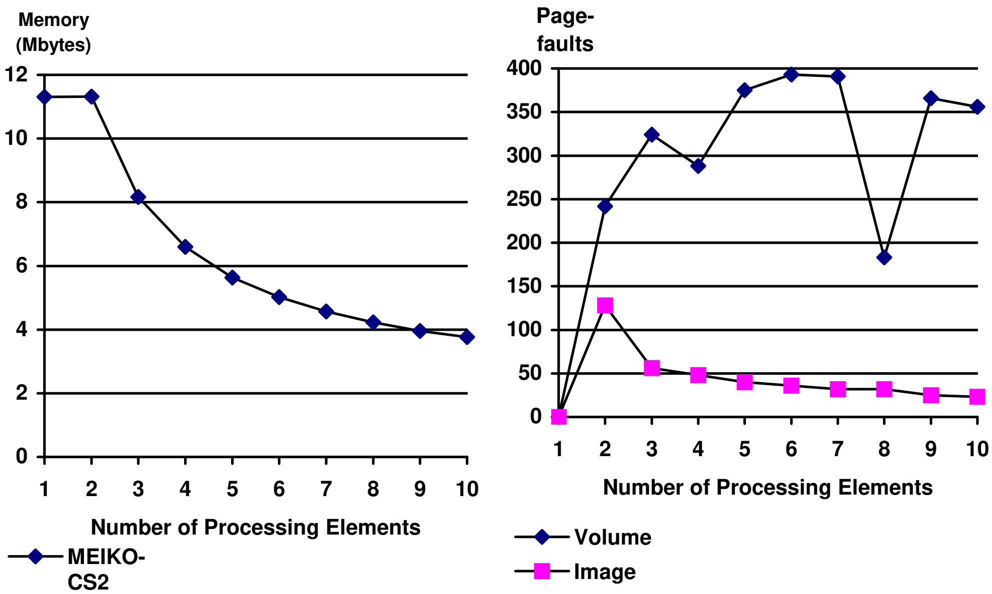


Figure 9 – Memory Demand and Page-faults per Processing Element for the Volume Ray-casting Program

6 Conclusion and Future Work

Many volume visualisation algorithms are inherently parallel. We developed the PVV library that facilitates the implementation of parallel volume visualisation programs using the DSM paradigm. The DSM was implemented considering portability issues among different parallel processing platforms and the natural data coherence of volume visualisation algorithms.

We implemented and evaluated two volume visualisation programs with the PVV library. The programs were developed with implicit parallelism and preliminarily results showed that in both cases the parallel performance scales with the number of processing elements. Another advantage is that the memory requirements in each processing element tend to decrease when the total number of processing elements increase, suggesting the use of the PVV library for processing huge volume datasets. Finally, one important conclusion is that the performance is strongly dependent from tuneable parameters.

Currently we have already versions of the PVV library running in operating systems such as LINUX, IRIX and DEC-OSF. We are implementing other volume visualisation algorithms such the shear-warp and region growing based 3D image processing tools to perform further detailed performance evaluation of the proposed system.

7 Acknowledgements

The authors would like to thanks Andrew Grant, from the University of Manchester, for their fruitful observations during the specification of the PVV library; Marcus Luchesse, from São Paulo University, for its performance measurements on the MEIKO-CS2; Fábio José Ayres, from São Paulo University, for his Ray-casting implementation in the PVV library; and Jecel Mattos Asumpção for the revision of this paper. Finally we would like to thanks Mr. Terry

Hewitt, director of the Manchester Visualisation Centre, and Prof. João Antonio Zuffo, Director of the Laboratório de Sistemas Integráveis, for their support on the developing of this project.

8 References

[1] J. Protic, M. Tomasevic, and V. Milutinovic. “Distributed Shared Memory: Concepts and Systems”, IEEE Parallel and Distributed Technology, summer 1996, pp. 63-79.

[2] A. Kaufman. “Volume Visualisation”, IEEE Computer Society Press Tutorial, A. Kaufman (Ed.), Los Alamitos CA, 1990.

[3] B. Corrie and P. Mackerras. “Parallel Volume Rendering and Data Coherence”. Proceedings of The 1993 IEEE Parallel Rendering Symposium, IEEE Computer Society Press, 1993, pp. 23-26.

[4] T. Elvins, “A Survey of Algorithms for Volume Visualisation”. Computer Graphics (26) 3, 192-201, 1992.

[5] P. Lacroute, “Real Time Volume Rendering on Shared Memory Multiprocessors Using The Shear Warp Factorization”, IEEE 1995 Parallel Rendering Symposium, p.15-22, IEEE Computer Society Press 1995.

[6] C. Montani, R. Perego, R. Scopigno . "Parallel Volume Visualisation on a Hypercube Architecture". Proceedings of the 1992 Workshop on Volume Visualisation, Boston 1992.

[7] U. Neumann, "Parallel Volume Rendering Algorithm Performance on Mesh-Connected Multicomputers", Proceedings of IEEE Parallel Rendering Symposium, 23-26, IEEE Computer Society Press 1993

[8] J. Nieh, M. Levoy, "Volume Rendering on Scalable Shared Memory MIMD Computers", proceedings of the 1992 Workshop on Volume Visualisation, Boston 1992.