Recognising and Avoiding Thrashing in Dynamic Load Balancing

Frank Stangenberg

EPCC–SS94–04

Recognising and Avoiding Thrashing in Dynamic Load Balancing

Frank Stangenberg

EPCC–SS94–04

September 1994

Abstract

This project investigates the phenomenon of thrashing, which can appear during dynamic load balancing. The dynamic load balancing functions of PUL-SM are used to explore configurations that can lead to thrashing. Thrashin in two dimensional meshes can be detected by visualising the mesh decomposition after each load balancing stage, but this technique is inefficient, time consuming and cannot easily be extended to three dimensional meshes. A simple algorithm to detect some kinds of thrashing without visualisation is described. Using this thrashing detection mechanism, techniques which successfully avoid many kinds of thrashing, have been developed. This report presents the results achieved by these thrashing techniques.

1 Introduction

Irregular meshes are often used for modelling the real world to calculate problems like air flow or fluid dynamics. Normally these meshes are very large (existing production meshes range from 200000 elements[1] to well in access of 3 million) and because of their size the calculation can be very time consuming. By distributing irregular meshes across a network of processes[2], the calculation can be speeded up. PUL-SM is a library which provides functions for working with static irregular meshes in parallel. The facilities provided by PUL-SM include dynamic load balancing functions, which redistribute the computational load more evenly among the processes during the calculation. "Static" in this context means that the structure of the mesh does not change. (e.g. there is no mesh refinement or derefinement during the calculation). During the load balancing process elements are possibly moved to other processes to make the load more even. If movement of elements is performed without an improvement in the load balance, then *thrashing* has appeared.

The goal of this project is to find a mechanism for detecting thrashing without use of visualisation, and to avoid thrashing if it appears.

This paper starts with a short description of irregular meshes (Section 2.1) as well as dynamic load balancing (Section 2.2) and the dynamic load balancing algorithms used in PUL-SM (Section 2.3). Section 3 introduces thrashing and describes this phenomenon in more detail. Section 4 describes the tools used at EPCC for the visualisation of irregular and regular meshes. The detection of thrashing is descibed in Section 6 and a method of avoiding thrashing is presented in Section 7. Section 8 presents the results in detecting and avoiding thrashing and finally Section 9 offers some conclusions.

2 Background

2.1 Irregular Meshes

An irregular mesh is a 2D or 3D collection of connected elements, covering an area of interest, such as the air around the wing of an aeroplane. *Nodes* and *edges* are the basic components of irregular meshes (as well as of regular meshes). Nodes are points in a 2D or 3D space and connected by edges to form the irregular mesh. The edges connect to form closed polygons. In a 2D mesh these polygons are the elements of the mesh. In a 3D mesh, these polygons are the faces of polyhedra. Each polyhedron is an element of the 3D mesh. For irregular meshes no underlying regular structure is required. They may consist of different kinds of polygons (polyhedra in 3D) but usually they are used with a single type of polygon (triangles or hexagons) or polyhedron (tetrahedra). Figure 1 shows examples of irregular meshes in 2D.

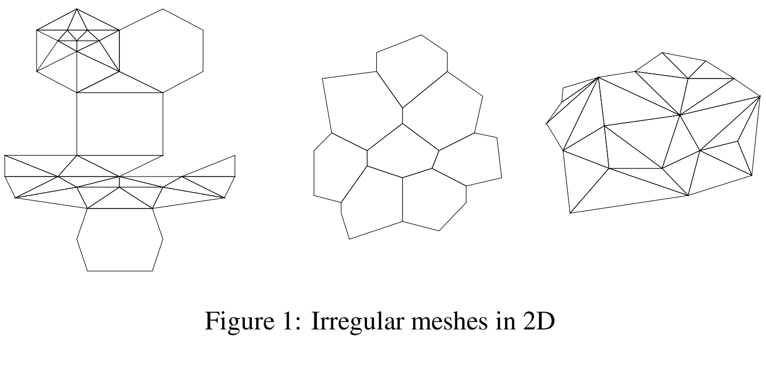


Figure 1: Irregular meshes in 2D

A load is given to every element in an irregular mesh depending on the amount of work to be done for that element. This load may change during the calculation. To improve the performance of computations

[1] In PUL-SM, elements and entities equivilant

[2] Throughout this paper "network of processes" means the logical arrangement of processes rather than a hardware configuration

done using such irregular meshes they can be distributed over a network of processes. Figure 2 shows an example of a distributed mesh.

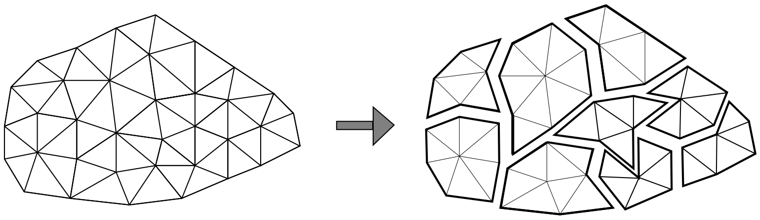


Figure 2: Distribution of an Irregular mesh

Each process holds at least one element of the mesh. The load on each process is the sum of the loads of its elements. During the calculation the loads of the elements may change, possibly leading to a load imbalance. Maintaining good performance in this case requires that a load balancing step be performed dynamically during the calculation.

2.2 Dynamic Load Balancing

When irregular meshes are initially distributed to a process network, an attempt is made to distribute the load as evenly as possible. During the calculation the load of the elements may change, leading to possible change of the load on different processes, probably resulting in a load imbalance. Dynamic load balancing tries to redistribute the work more evenly among the processes. In this context "dynamic" means that the distribution of the irregular mesh is redefined during the calculation. There are two main approaches for dynamic load balancing:

1. Global dynamic load balancing (see Section 2.2.1)
2. Local dynamic load balancing (see Section 2.2.2)

2.2.1 Global Dynamic Load Balancing

For global dynamic load balancing a special process is needed which totally redefines the distribution of the irregular mesh if necessary. This process has to have knowledge about the whole irregular mesh and the load of each element. To get this information and to perform the redistribution, a lot of communication is required. Because of this large amount of communication, this strategy is only efficient when the load imbalance is very bad, and only feasible when the mesh is not too large for a single process to hold information about it. Global dynamic load balancing is not a scaleable solution.

2.2.2 Local Dynamic Load Balancing

Local dynamic load balancing strategies are based on a direct neighbour communication. Two different ways of dynamic load balancing with direct neighbours are possible:

- Load balancing with one neighbour at a time
- Load balancing with all neighbours at a time

For the first direct neighbour load balancing strategy the load between two processes is compared and possibly an element shift is performed. Either the choice of the neighbour to compare with is done randomly, or the process collects load information from all neighbours and then chooses the neighbour to compare with (normally the neighbour process which is mostly lightly loaded, because it has less work to do).

Load balancing with all neighbours at once means that the processes collect the load information from all their neighbours, probably shifting elements to one or more neighbours at once. This is the main dynamic load balancing strategy used by PUL-SM.

2.3 Dynamic Load Balancing in PUL-SM

In the main dynamic load balancing strategy used by PUL-SM, each process determines the load of every direct neighbour and compares this load with its own. Direct neighbours are those processes which share a common point or edge (or face in 3D). This leads to a number of elements being moved to processes which are lighter weighted. Figure 3 shows an example.

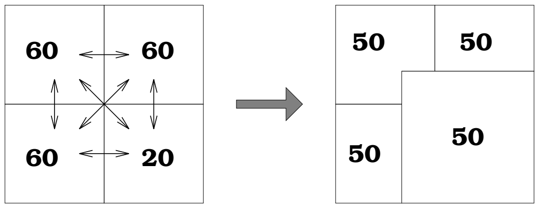


Figure 3: Load balancing strategy of PUL-SM

In the first step every process determines the load of its direct neighbours. This is represented by the arrows in Figure 3. Using this information it calculates the number of elements to be shifted to one or more neighbours in the second step. Finally, the elements are moved.

In addition to the main dynamic load balancing algorithm used in PUL-SM, two variants of this dynamic load balancing are implemented, a *fast* balance and a *fine* balance. The algorithm used for fast dynamic load balancing attempts to minimise interprocess communication. At each load balancing step, when the load of two neighbours differs by a value greater than a threshold, given as a parameter to the functions, all the elements of the more heavily loaded process which border the lightly loaded process will be given away to that process. The user is allowed to change this threshold value. All other decisions are made by the load balancing algorithm itself.

The fine dynamic load balancing algorithm allows the application a greater degree of control over the number and choice of elements that are transfered during load balancing. An application defined function is used to determine how many elements should be transfered, given their relative loads. This algorithm requires a greater number of communications between neighbouring processes and will loosely synchronise the calling processes.

This project uses *sm_fineBalance* and *sm_swFineBalance*, the fine dynamic load balancing functions provided by PUL-SM, because they allow the user to control the number of elements to be swapped (in the *transfer* function) and which elements to move.

PUL-SM will move the elements with the highest cost (not necessarily the elements with the highest load) first. Before one of these dynamic load balancing functions is called, the application has to choose which elements to move. It calculates a cost for each element, depending on the load this element has (for example, the application might give a low cost to highly loaded elements, so that lightly loaded elements are moved first). Investigation of thrashing in this project involved altering these paramters.

3 Thrashing

3.1 What does Thrashing mean?

Definition: Thrashing
The movement of elements by the process of dynamic load balancing without distributing the load more

evenly.

If thrashing appears, it is possible the process of load balancing will not end. So recognizing and avoiding thrashing is an important task for load balancing. There are two main kinds of thrashing:

1. local thrashing (see Section 3.2)
2. global thrashing (see Section 3.3)

3.2 Local Thrashing

Local thrashing happens between direct neighbours with nearly the same elements being shifted. As described before, elements are moved around between direct neighbours, without the load becoming more evenly distributed. The possible types of movement of elements are described in Sections 3.2.1 and 3.2.2.

3.2.1 Swapping

If two or more processors share common nodes and edges (faces in 3D) of the irregular mesh then during the process of dynamic load balancing direct neighbour swapping may arise. This means that if one process determines it is overloaded, it starts a process of load balancing in which one or more elements are shifted to one or more direct neighbours. If very highly loaded elements are shifted or too many elements are shifted, this might cause the neighbour to be overloaded, possibly resulting in it sending elements back to the initiator, etc., as Figure 4 shows. Figure 5 shows another example of direct neighbour

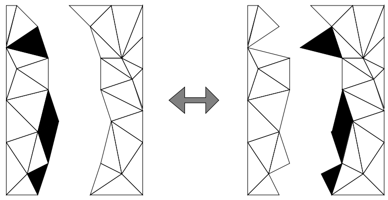


Figure 4: Direct neighbour swap between two neighbours

swapping in which the process directly in the middle of the grid is more lightly loaded than the rest. This process receives elements from all its surrounding neighbours and becomes overloaded. The next time the load balancer is called, this process detects that it is overloaded in comparison to every direct neighbour. Elements are then moved away and possibly the initial state is reached again, etc.

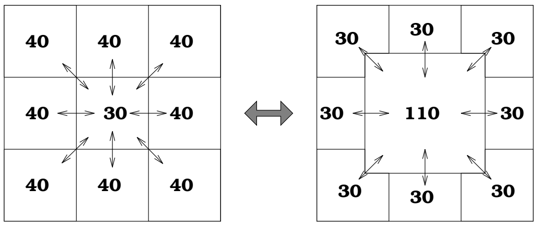


Figure 5: Direct neighbour swapping (pumping)

3.2.2 Cyclic Movement

If three or more neighbours share one common node in 2D (see Figure 6 for details) and one element

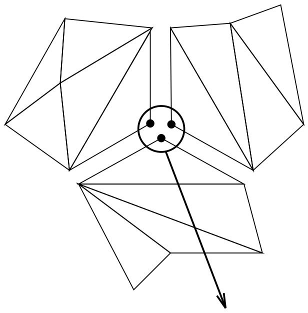


Figure 6: Three submeshes sharing a common point

which is incident to this point has a very high load, then this element may be shifted in a cycle as shown in Figure 7. Because of the high load of this element they do not reach a state of being evenly loaded.

This case may also happen in 3D, where the neighbours could share an edge, or a single node.

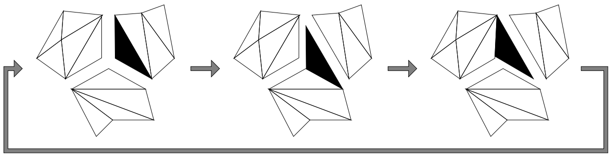


Figure 7: Cyclic shifting of one element

3.3 Global Thrashing

During dynamic load balancing it is possible that the load between direct neighbours becomes more even, but the global load imbalance of the whole application is as bad as before or worse. This is called “global thrashing”.

The same phenomena of movement as described in the former section may appear involving processes which are not necessarily direct neighbours. Other movements of elements are also possible. The different kinds of movement are explained in more detail below.

3.3.1 Global Swapping

Global swapping involves more than two processes in this kind of thrashing. These processes are not all direct neighbours and the elements a process moves to a neighbour are possibly different from the elements received before. As an example consider a network where the processes are connected in a linear network. The first process shifts its overload to its bottom neighbour. Due to the heavy load of the elements shifted (or due to the number of elements shifted), it is now overloaded, so it shifts its load to the bottom etc. If the last process in the pipeline gets this overload, it is also overloaded and shifts this overload back (Figure 8)

3.3.2 Global Cyclic Movement

The phenomenon of global cyclic movement involves the whole process network or parts of it. The processes are not all direct neighbours. Figure 9 shows an example with three processes connected in a cycle. This cycle can be expanded to more than three neighbours. The example shows borders which

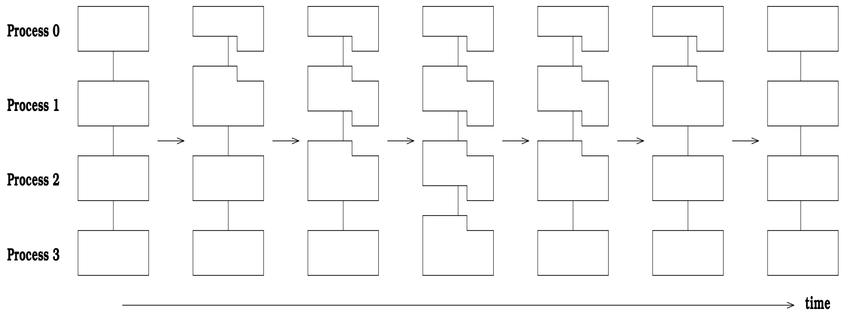


Figure 8: Swapping in a pipeline

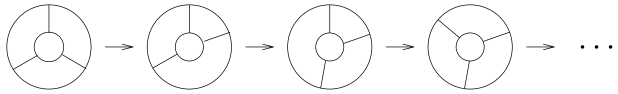


Figure 9: Global cyclic thrashing

are moved around and around. This may appear if too many elements or too heavily loaded elements are moved to a neighbour.

3.4 Irregular Movement

One of the most difficult kinds of movement to detect is the irregular movement of elements when global thrashing appears. There is no underlying structure in this kind of movement and none of the movements described above apear for a long time. The load balancer tries to do its work as well as possible but the global load imbalance remains, worsens or differs only slightly.

4 Visualisation

For the visualisation of the distribution of irregular meshes two tools are mainly used at EPCC. Both tools use Silicon Graphics Explorer, which provides several functions for visualising data. Two Explorer maps were created to visualise meshes in two different ways. These tools are postmortem tools, which means the that visualisation of the meshes has to be done after the application is fineshed. The application has to store several time steps of the mesh distribution during the run. These time step files are then used for visualising the mesh.

Each process writes out its part of the mesh using its own process identification as a colour code. All processes write to one file using PUL-GF, which provides functions for parallel in- and output. PUL-GF also provides the option to set a specified file access mode so that every individual process is able to write at any time and at any positiom within a global file. Each process may be considerd as having its own global file pointer. Using this mechanism each process writes its data to a special position in the file, using the global number of each element as an address. This file has to be merged with the center coordinates of the elements to create the file which can be used by either one of the tools.

The first tool (*Quick*) shows a mesh where the elements are represented by spheres around their centers. Different colours are used for different processes. It is used for a quick visualisation of meshes, to get an idea of the distribution. The shapes of the spheres can be changed for faster visualisation. Actually this tool doesn't show the load of each element but with a few changes this might be possible in the future. Figure 10 shows a screenshot from *Quick*.

The second tool *AnimMesh* is used for the animation of several time steps of a mesh distribution, but it can also be used to show different meshes one after the other. This tool is able to show the elements

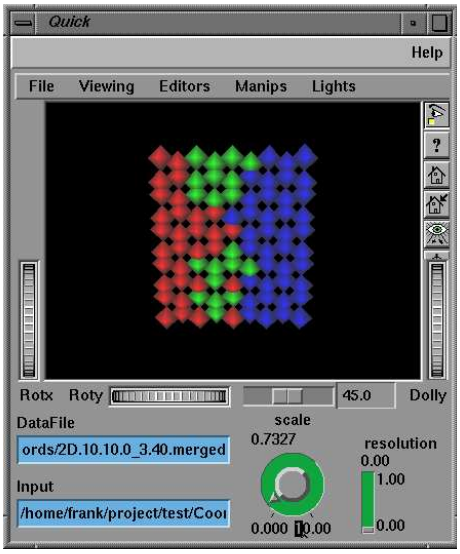


Figure 10: Visualisation Tool for Meshes

in different ways (spheres around the centers, only the coloured edges of the elements and flat shaded elements). It is possible to extend *AnimMesh* to 3D meshes but it is very hard to see the distribution of the whole 3D mesh during the animation. It is possible to show the part of the 3D mesh for only one process but then thrashing cannot be seen. Figure 11 shows two major modes (edge and flat shade mode) of *AnimMesh*.

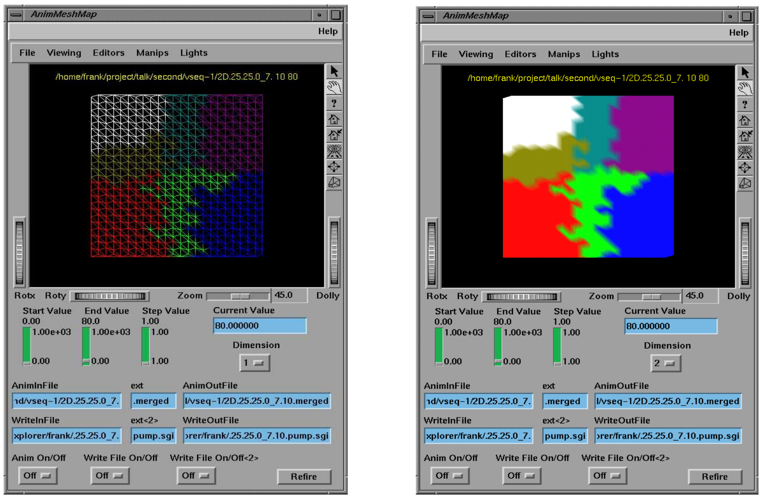


Figure 11: Animation Tool for Meshes

Another feature of *AnimMesh* is the possibility of storing each animated time slice of the mesh distribution in a special file format so that it is possible to produce a faster animation of the mesh distribution using the *movieplayer* or the *movie* tool from Silicon Graphics. Converting these files to a special video file format allows the user to produce a video of the mesh distribution.

5 Generating Thrashing in PUL-SM

Variation of the parameter of the PUL-SM load balancing algorithm has an effect on their tendency to thrash. To investigate this relationship it is necessary to generate thrashing. Once this information is available, it is possible to avoid these configurations and to establish methods for detecting thrashing.

This project tries to produce thrashing using the fine dynamic load balancing functions of PUL-SM because they give the application a lot of influence over the load balancing itself. The application controls the number of elements and which elements to move. These are the parameters explored in generating thrashing.

The process of producing thrashing was successful for some kinds of movements which are described in more detail below. The following kinds of movements have not been produced so far: cyclic movement (direct neighbour thrashing), global swapping and global cyclic movement (both global thrashing).

The first attempt to produce thrashing gave a random load to each element of the irregular mesh. The heavily loaded elements were moved first and the number of elements were moved due to the load difference between two processes (i.e. less than 10% difference, no elements were moved, between 10% and 20% difference, 10 elements were moved etc.). With this strategy it was impossible to generate thrashing, because the load of each process was nearly the same after several load balancing steps.

The next idea gave a very high load to border elements of one specified process and a low load to all other elements. Heavily loaded elements were moved at first and the number of elements to be moved was calculated from the difference between two processes. If the difference was less than 10%, no elements were moved, if the difference was between 10% and 20% one element was moved etc. This strategy produced pumping, a direct neighbour movement of elements. The location of the process with heavily loaded border elements did not appear to matter. This kind of thrashing appears in any case (Figure 12 shows an example).

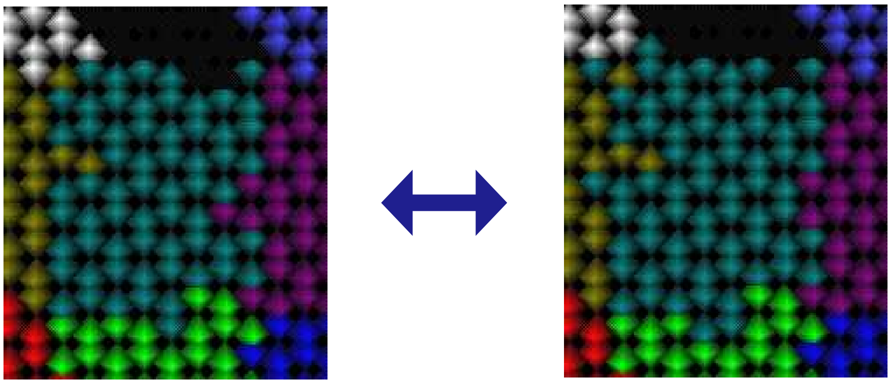


Figure 12: Swapping (pumping) in PUL-SM

6 Detecting Thrashing

6.1 Detection via Visualisation

Until this project started, detecting thrashing in PUL-SM was only possible using the described visualisation tools. This process is very time consuming and is postmortem detection of thrashing rather than dynamic detection. Nevertheless all the described kinds of direct neighbour thrashing which were generated (see Section 5) were detected with either one of these tools.

6.2 Detection Mechanism

Because of the disadvantage of postmortem detection of thrashing in PUL-SM, the need for a detection mechanism which can be used during the load balancing process is obvious.

6.2.1 Method for Detecting Thrashing

The basic method for detecting thrashing is very simple. Every process counts how often it has given away an element. After a specified number of load balancing iterations, each process examines independently how often each element was sent. If an element has been sent too often, it is assumed to be involved in thrashing. While this method is quite simple it is capable of detecting all kinds of direct neighbour thrashing which were generated and detected via visualisation.

For the implementation of this method, an abstract data type *List* was used. Each entry in this list stores the global number of the elements lost, a count of how often it was given away and a pointer to the next entry in this list. Figure 13 shows an example of such a list.

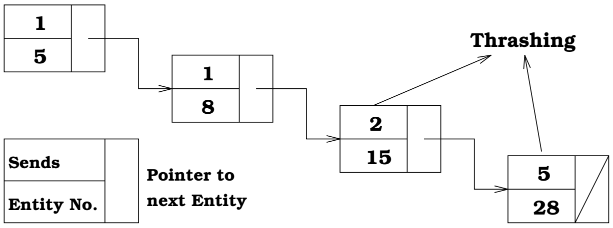


Figure 13: List of lost elemenst

For each element lost, a new entry in this list is created. If this element has been lost more than once the send counter for this element is increased. After a specified number of load balancing iterations (10 load balancing iterations turned out to be a good number), the list is examined for elements lost twice or more. If an element has been lost at least two times it might be involved in a process of thrashing. After the examination of the list, it is removed and a new detection procedure starts with the next load blancing step working on an initial empty list for each process.

7 Avoiding Thrashing

The method for avoiding thrashing is as simple as the method for detection. The basic concept is based on the strategy PUL-SM uses to move elements to other processes during the load balancing step.

PUL-SM first moves the elements which have a high cost. This does not mean that these elements necessarily have a high load. For example, it is possible to give a high cost to elements which are lightly loaded to move these elements first. This offers a way to establish different strategies of which elements to move. There are three main approches:

- move elements with a high load at first
- move elements with an average load at first
- move elements with a light load at first

The decision as to which elements to move has to be done before the load balancing process is called as well as calculating the cost for each element in the local part of the mesh. The costs of the elements are given as a parameter to the fine dynamic load balancing functions of PUL-SM. It is possible that each process will operate with a different strategy.

This leads to the basic method of avoiding thrashing and is used as an extension to the detection mechanism. After a process has detected thrashing it changes its strategy by which the elements are chosen to be moved. This switch in the strategy is done independently by each process if necessary.

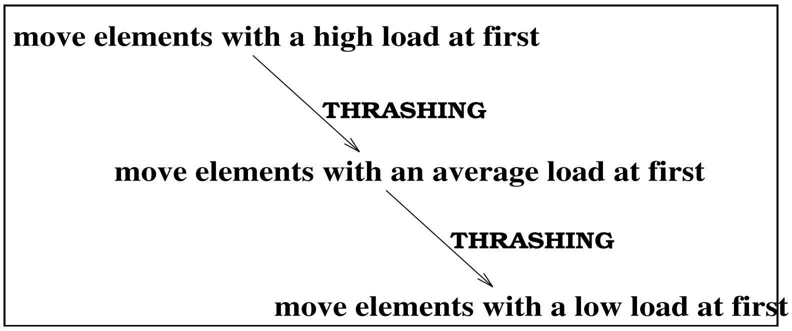


Figure 14: Strategy for avoiding PUL-SM

Initially the most heavily loaded elements are moved during the load balancing step. If thrashing is detected, the strategy is changed to move elements with average load. If thrashing is detected again, the strategy will switch to choose the lightly loaded elements at first (see Figure 14).

8 Results

With these methods of detecting and avoiding thrashing, it was possible to avoid most of the detected situations which are classified as thrashing. Most configurations reached a steady state in which they are load balanced within a load difference of 10% between each process. Figure 15 shows an example.

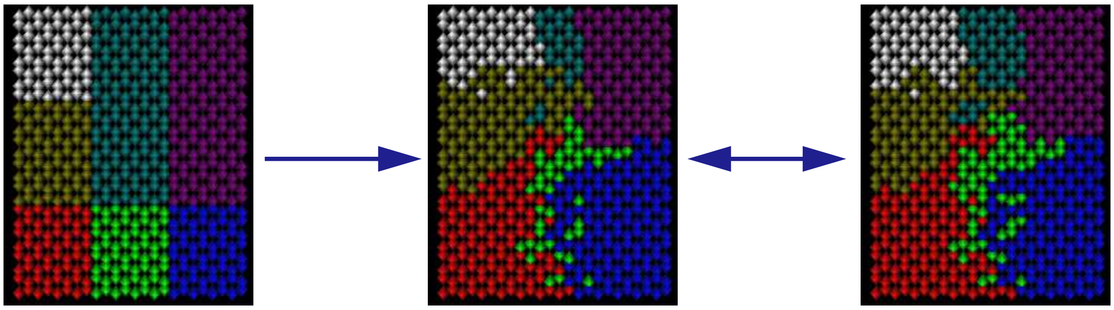


Figure 15: Example for Thrashing without a change in the strategy

In this example, a mesh with 600 elements was used as a test candidate. It was distributed to seven processes. The left picture shows the initial distribution of the mesh. The state of direct neighbour thrashing (pumping) was reached after several load balancing steps in which a switch between the middle and the right picture occurs with the same elements moved.

Figure 16 shows the same inital distribution (left picture) as in Figure 15, but in this example, the mechanisms for detecting and avoiding thrashing were successfully used. The right picture shows the final distribution in which the processes are load balanced within an 10% difference.

In a few configurations the attempt to avoid thrashing failed and instead produced other kinds of thrashing. Figure 17 shows an example.

The left picture shows the initial distribution of the mesh with 600 elements over 9 processes. The upper branch shows the state of direct neighbour swapping (pumping). This state was reached without using a method to detect and avoid thrashing. The lower branch shows the distribution of the mesh after 65

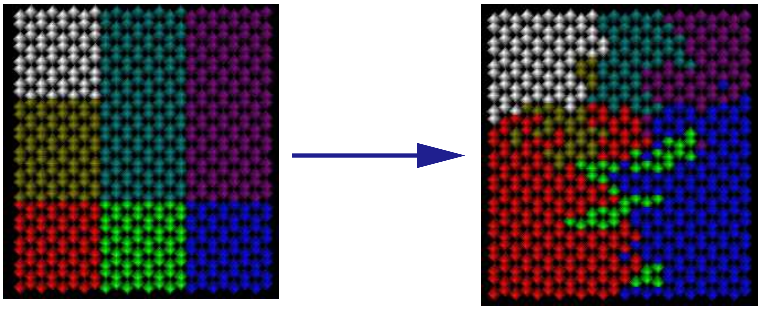


Figure 16: Example for Avoiding Thrashing

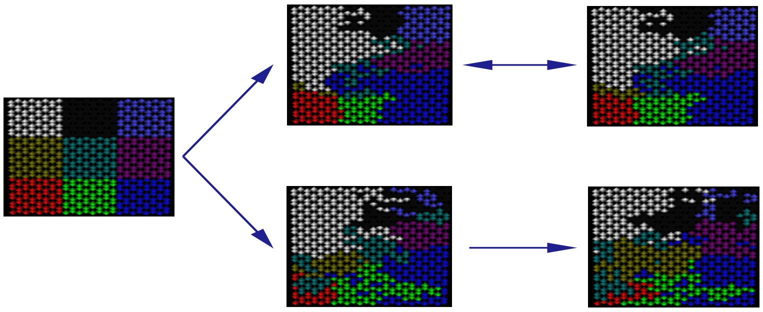


Figure 17: Example for switching to an other kind of thrashing

load balancing steps (middle picture) and after 80 load balancing steps (right picture). The detection and avoidance methods were used resulting in avoidance of direct neighbour swapping but turning into an irregular movement of elements. An acceptable load balance was not achieved within the 100 steps executed in the experiment.

The main result, and a very interesting one too, is that the smaller the mesh, the greater the tendency to thrash. This shows that the fine dynamic load balancing algorithms in PUL-SM are are better suited to larger meshes, so far as thrashing is concerned.

9 Conclusion

This report has shown the need for a mechanism for detecting and avoiding thrashing in the dynamic load balancing functions of PUL-SM. The project was successful in detecting the different kinds of direct neighbour thrashing which were generated and described in Section 5. But on the other hand, there are still some kinds of global thrashing which were not detected automatically. Detection of global thrashing seems to be a much more complicated problem. Another unsolved problem is the switch to other kinds of thrashing while trying to avoid thrashing.

Another further extension to this project is the use of other load balancing strategies which are not used in PUL-SM at the moment. Establishing a theoretical background for thrashing might also be helpful for possible future projects.

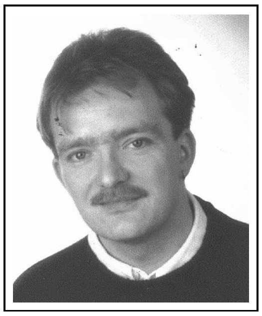
References

[KTG, 1993a] EPCC KTG. Pul-gf user guide. 1993.

[KTG, 1993b] EPCC KTG. Pul-sm user guide. 1993.

[L. Ridgway Scott, 1991] Reinhard v. Hanxleden L. Ridgway Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing 13, pp 312 - 324*, 1991.

[Williams, October 1989] Roy D. Williams. Performance of dynamic load balancing algortihms for unstructured meshes. *Concurrency: Practice and Experience, Vol 3(5), pp 457-481*, October 1989.



Frank Stangenberg is a student of Computer Science at the University of Paderborn, Germany. He is currently working on his diploma thesis. Beside his studies he work in the system and network administration group of the Heinz Nixdorf Institut Paderborn. After graduation in 1995 he hopes to find an interesting job in a computer graphics company.
This project was supervised by Shari Trewin (EPCC).