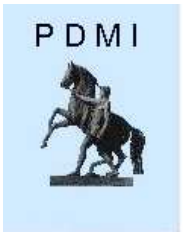


JASS 2004 - Numerical simulation

Parallel Programming and Cache Optimization for Finite Element Methods - the Benefit of Space Filling Curves (SFC)

Markus Langlotz

July 5, 2004

Abstract

This paper shall show how the use of *space filling curves* (SFC) can be used to improve the runtime properties of numerical algorithms. The encouragement of SFC has several independent benefits, which are pointed out in the following paper. Using SFCs for numerical computations one can improve an algorithm concerning to the underlying computer architecture.
Furthermore they can be used to partition a problem domain into parts, which can be processed in parallel on high performance computers.

Section 1 describes some basics about computer architecture. The center of interest is the memory hierarchy and its influence onto numerical algorithms.
Section 2 pointes out how an implementation of numerical algorithms using SFC can obtain better results concerning cache efficiency than standard implementations. Based on the basic descriptions of a SFC algorithm [**?**], further properties of the implementation of SFC are shown. The full implementation is described in [**?**].
Section 3 focuses on parallelizing. First some standard partitioning shemes will be mentioned and described shortly. Subsequent to this, the SFC approach for partitioning a domain will be depicted. Both, the SFC partitioning and the stack algorithm is going to be used together in the same implementation.

Contents

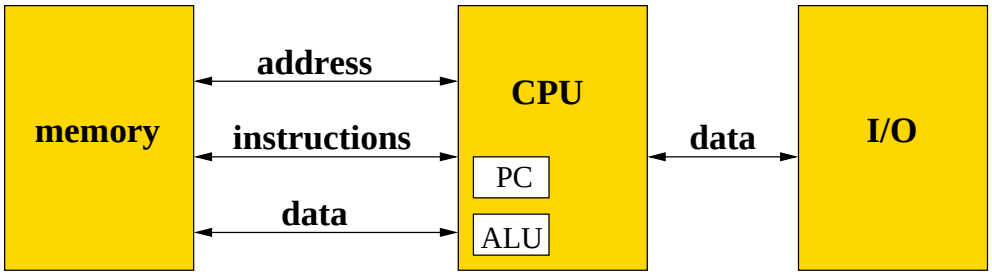

Figure 1: computer architecture

1 Computer architecture

The development of computers already lasts for several centuries. But still our most modern computers follow the architecture of John v. Neumann. In his idea a computer consists of five building blocks. They are

- an arithmetic-logic unit (ALU)
- a control unit (CU)
- a memory
- some form of input/output (I/O)
- and a bus.

From our point of view the use of this machine is most interesting. As it is designed to process every possible computation, there has to be a way to formalize its computation steps. These steps are called algorithm and are stored using instructions and control structures in the memory. In addition to that the input data and the data resulting during the computation are stored within the memory. An overview over the main parts is shown in figure 1 on page 4.
Let's have a short look at the way one instruction is executed. The program counter (PC) holds an address, which points to the position in the memory storing the instruction to be processed next. After utilizing the bus to load this instruction, the control unit can interprete it. Beside the operational part of the instruction, which describes the kind of the operation, the instruction also contains the operands. These operands also have to be fetched from the memory. After gaining all information needed, the ALU executes the operation and produces a result, which has to be stored and thereby again the memory has to be accessed. Summarizing this procedure, it can be observed, that in the worst case, it is necessary to access the memory three times, while the central processing unit just processes one instruction.

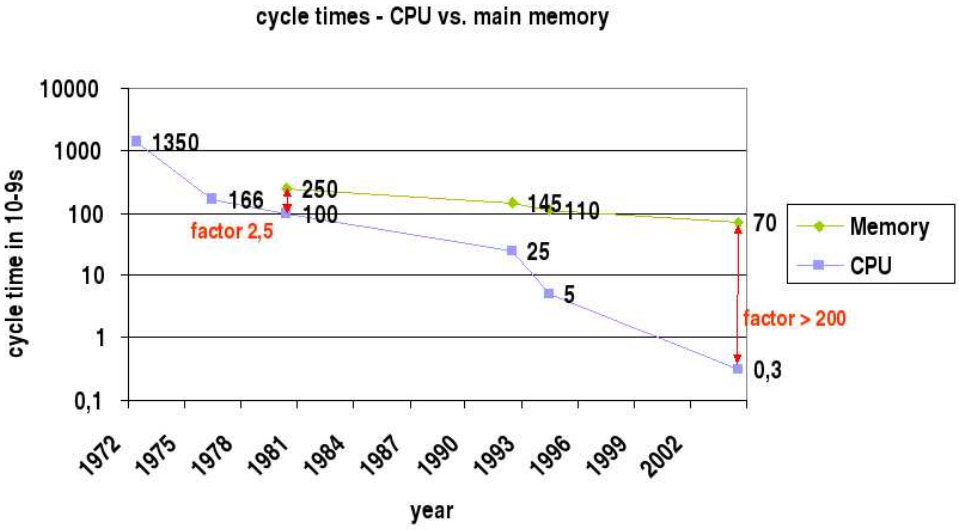

Figure 2: development of cycle times

This wouldn't be a reason to worry about, if the memory would be fast enough to enable three accesses per instruction. Unfortunately this wasn't the case during the last 30 years. The CPU cycles were even faster than the memory cycles. Furthermore the development of memory and cpu speed evolves differently. As shown in figure 2 the gap between the cycle times of CPU and memory has increased to a factor greater than two hundred. Up to now, there is now sign, that the gap can be closed by newer technologies in future.

Obviously the memory induces a slowdown of the computation of a programm, in comparison to the optimal computation time. One can easily assume, that each time the processor starts to execute an instruction it has to wait for hundreds of cycles until the operation code and all operands are read from the memory. This would be true, if modern engineers had not invented techniqus to close that gap. First of all there are memory types with cycle times in the same order as the CPU cycle times. But such memory types are expensive and small. As the smallness is the main reason for the short access times, memory cannot easily be enlarged.
For that reason a memory hierarchy was introduced. As shown in figure 3 on page 6 there are 5 memory types, which differ in size and speed. The main idea behind this hierarchy is, that one can store data, which is used frequently in fast memory and data, which won't be important for a certain time, in slower but bigger memory. In real-

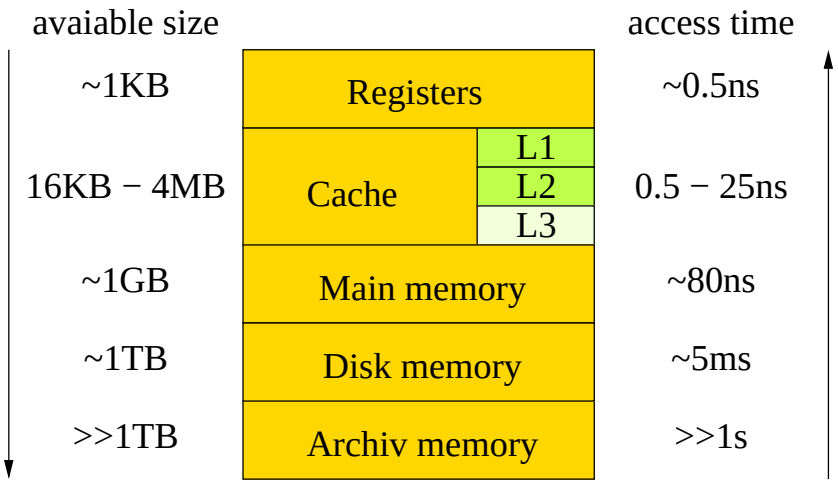

Figure 3: memory hierarchy

ity, cache and registers hold copys of data and instructions, which are stored in the main memory. The main memory stores running programms and their data. Data on disks and archive memory is stored in order to access them some time in the future. This hierarchy is further extended by the different levels of a cache. Within a cache the L1 (level one) cache is the smallest and fastest one providing the same speed as the registers. The L2 cache is bigger but has, as the logical drawback, higher access times. Some of the more modern processor architecture even have a third cache level called L3 cache. All together a modern computer provides up to 8 memory types, which differ in size and speed.

We focus on the cache of a computer, as its utilization is from vast importance for the runtime of numerical algorithms. As the cache size is much smaller than the size of the main memory, only some rather small parts of the data can be stored there. To decide, which data is stored in the cache, the corresponding algorithms use the locality properties of computer programs. On the one hand programs tend to use variables, which were used once, serveral times after another. This property is called temporal locality. On the other hand spatial locality can be observed. Thereby one can exploit, that programs often access memory next to memory locations, which were just accessed before. Using this information, cache algorithms can be implemented, which most often are able to deliver the requested data/instruction out of the fast cache instead of accessing relativly slow main memory. It can be verified, how successful a architecture achives this aim. By counting the cache misses (memory access can't be delivered from cache) and

cache hits one can calculate the cache hit rate.

$$hitrate = \frac{n_{cachehits}}{n_{cachemisses} + n_{cachehits}} \tag{1}$$

The *hitrate* is a measure for the cache efficiency of an algorithm. If this rate exeedes 95%, the algorithm can be regarded as cache efficient. But even with a value of 99% an algorithm still wastes hundreds of CPU cycles every 100th instruction. The results of the following chapter show, how it is possible to reach cache *hit rates* of $> 99.99\%$!

2 Cache efficiency

This section first shows why the standard approach of accessing memory in numerical algorithms leads to terrible results concerning cache efficiency. Subsequent to that, a new idea is introduced including new ways of memory usage and access. The main idea of this architecture is the usage of a set of stacks instead of memory fields. Stacks allow an efficient access to the required data, while the algoritm is traversing through the domain. The traversion path follows a Peano curve. In order to use a consistent term the architecture subsequently is called *stack architecture*.

2.1 Original approach without SFC

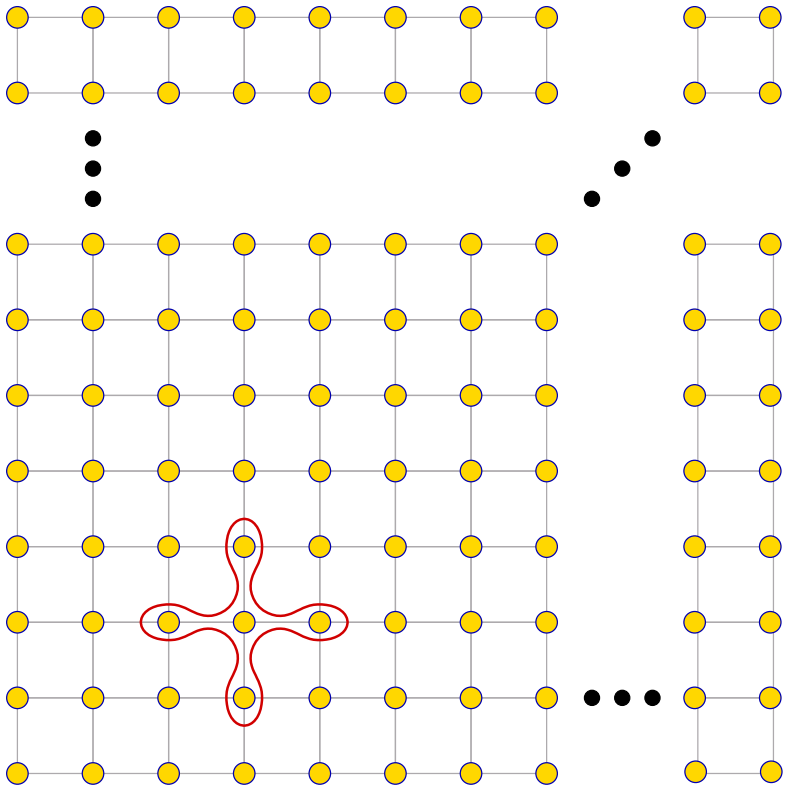

Figure 4: evaluation of the 5-point-stencil

To solve partial differential equations (PDE) the following steps have to be done. First of all the PDE is discretized which leads to a linear equation system (LES) like $Au = b$. To solve this, one can use

iterative LES solvers, as Jaccobi or Gauss-Seidel. Performing FDM in the two dimensional case usally means to evaluate a 5-point stencil on the field u. Imagine a two dimensional field as can be seen in figure 4. To evaluate the node $u_{5,4}$ one needs all neighbour elements. Thus the nodes $u_{5,3}$, $u_{4,4}$, $u_{5,4}$, $u_{6,4}$ and $u_{5,5}$ are needed. In the case of $n = 5000$ these elements are distributed within the memory. If the field is stored linewise, the position of the elements are between 10005 and 20005. This wouldn't lead to a problem if all elements were stored in the cache of the CPU. Unfortunalty the whole field would need about 200MB and still the 3 accessed lines are to big too be completly stored in the cache (120KB > L1 cache). An implementation, which only simulates this data structure and the above described way of accessing it led to the following results:

Architecture	Pentium IV Xeon
# elements	$1,25 * 10^8$
Cache size	512 KBytes (128 Bytes each line)
L2 cache miss rate	**15,0 %**

Obviously the results are rather bad, hence every seventh usage of the memory leads to a cache miss resulting in many wasted CPU cycles. These results are even more interesting as the Xeon family has a big cache size and so is optimized to achieve low cache miss rates. The following section shows how to reach this.

2.2 Stack architecture (SFC)

As an introduction to this section it is strongly recommended to read [**?**] first, as some previous knowledge is required. The general idea of traversing the domain Ω using Peano curves leads to the stack architecture, which needs 3^d stacks, where d is the number of dimensions of the domain. The following part motivates the number of different stacks. Computation within a cell of Ω requires an access to all edges of the cell. As introduced one needs 2 stacks to store them. One for each side of the curve. Experiments have shown that 2 stacks aren't sufficient in every case. Figure 5 shows one possible problem. The yellow marked coarse node is stored on the (right side) stack when the coarse cell in the center is left (mark 1). The brown marked node lies on refinement level deeper then the yellwo one. While the algorithm traverses the domain the brown one is stored on top of the stack (mark 2 & 3). When the coarse cell in the middle of the top row is accessed, the yellow one is going to be read, but the brown one resides above it.

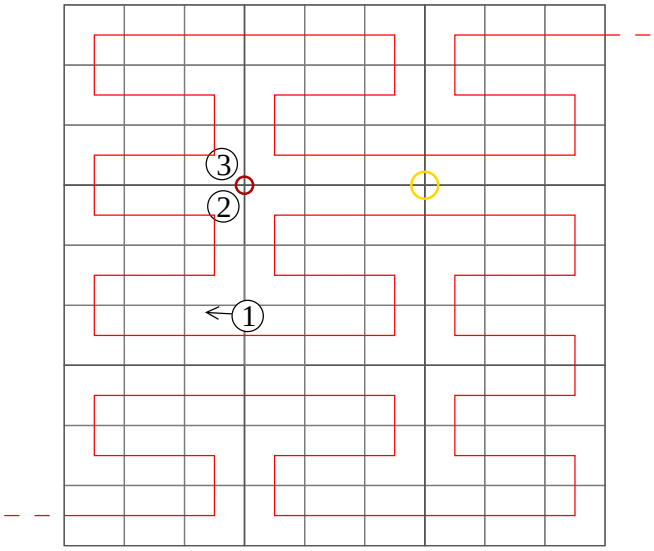

Figure 5: motivation for stack architecture

To solve this problem, each of the nodes has to be regarded seperatedly. A node chances the stack depending on number of accesses to it. So after each access, a node is stored to another stack. In general there are 5 stacks in the two dimensional case (left/right is not taken into account). As shown in figure 6 on page 12 the nodes are stored onto stacks corresponding to the number of already perofrmed accesses. Hence a node is stored on stack 2 after it has been accessed two times. E.g. this is the case for the red node in figure 6, when the algorithm reaches the grey cell. The continuous access onto neighboured cells can be seen in figures 6 to 8.
This concept can be enhanced by collecting similar stacks together. Thereby not the number of accesses but the "completeness" of a node plays the crutial role. As it is easier to imagine the following example is made for the 3 dimensional case. One should note that a node, which isn't located at the border is accessed 8 times in 3D. After a node has been accessed 4 times it can be regarded as "plane complete". This is because all cells on one side of the node have been traversed. Hence we store this node on a so called plane- or *2D Stack*. Because of the same reason we store a node on the *1D Stack* as soon as it is "line-complete". This happens after the second and the sixth access. If a node has been accessed 1,3,5 or 7 times is is stored on the *0D Stack*. Finally after the last access a node is "volume-complete" and hence can be stored on the *3D stack*.
For the same reasons, as mentioned in [**?**] one needs multiple stacks

for each dimensional completeness. For further information the reader is advised to read [**?**].

Using the above described approach of 3^d stacks to traverse through Ω we have achieved the following results.

Architecture	Pentium IV Xeon
# elements	$4 * 10^8$
Cache size	512 KBytes (128 Bytes each line)
L2 cache miss rate	**0,01 %**

As can be seen easily, the results show an excellent improvement in the relative number of L2 cache misses. How can this be explained and what are the responsible properties of the space filling curves, which lead to that improvement?
In order to find out the reason one has to concentrate on the construction idea of space filling curves. As mentioned in [**?**] a space filling curve always fills a full quadrat (two dimensional case) or a cube (three dimensions). This leads to an always compact area covered by the curve. Furthermore the surface (border) of the traversed part of Ω stays small compared to the number of nodes within its area. The following rules can be obtained:

$$\begin{aligned}
\text{\# nodes per dimension on curve} &= n && (2)\\
\#dimensions &= d && (3)\\
\text{distance between } n_1 \text{ and } n_2 &= l && (4)\\
\text{distance in space} &= d(x,y) && (5)\\
d(n_1,n_2) &= O(l^{1/d}) && (6)\\
\text{surface of curve} &= O(d(n_1,n_2)^{d-1}) && (7)\\
&= O(l^{\frac{d-1}{d}}) && (8)
\end{aligned}$$

As can be seen in equation 8 the surface of a space filling curve with length l tends to be small in comparision to l. It is known from the beginning of this section, that the nodes on the surface of the alredy traversed domain have been accessed at least once and at most $(max - 1)$ times $(max = 2^d)$. So the corresponding stacks can be considered to be small. Therefore they can be held inside the main memory completely. Only the 3D stack is large, as it stores all elements, which were not used yet or which are finished. Hence this stack has to be stored on the disk. But small stack sizes don't lead inevitably to the cache efficiency mentioned before. Again looking onto the figure sequence 6 to 8 one can see, that the stacks are oszillating within a small bound. This narrow area can completely be held in the

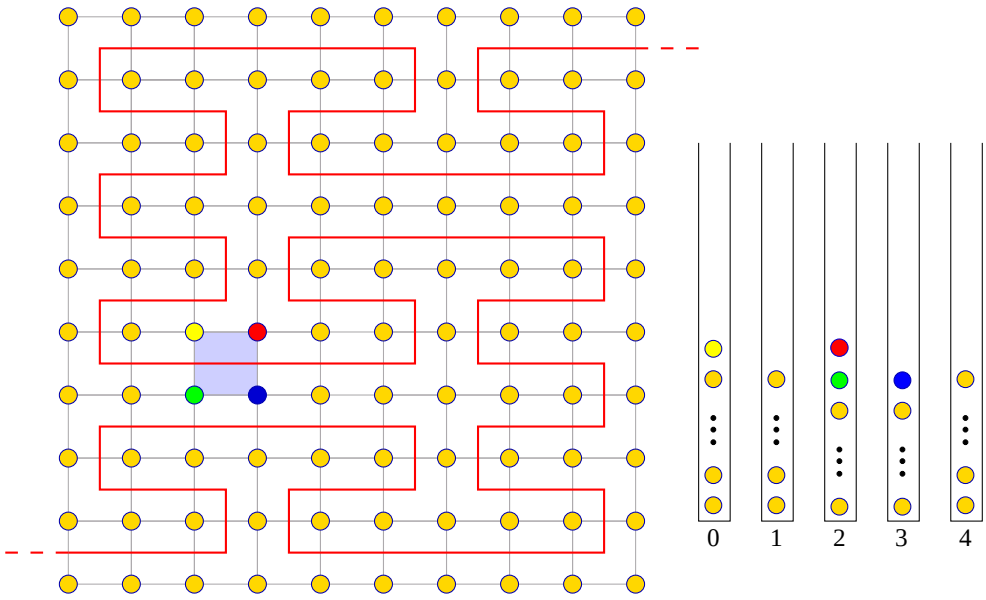

Figure 6: image sequence 1/3

cache. Up to now, no exact measurement or formalization was done to document this idea. Further work has to be done to achieve exact theoretical knowledge on the practical results.

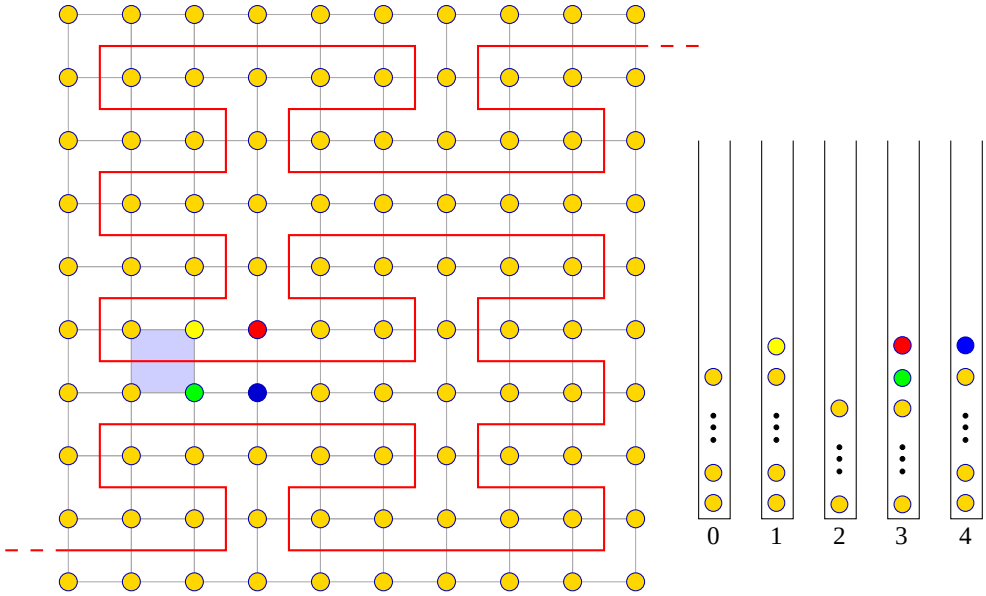

Figure 7: image sequence 2/3

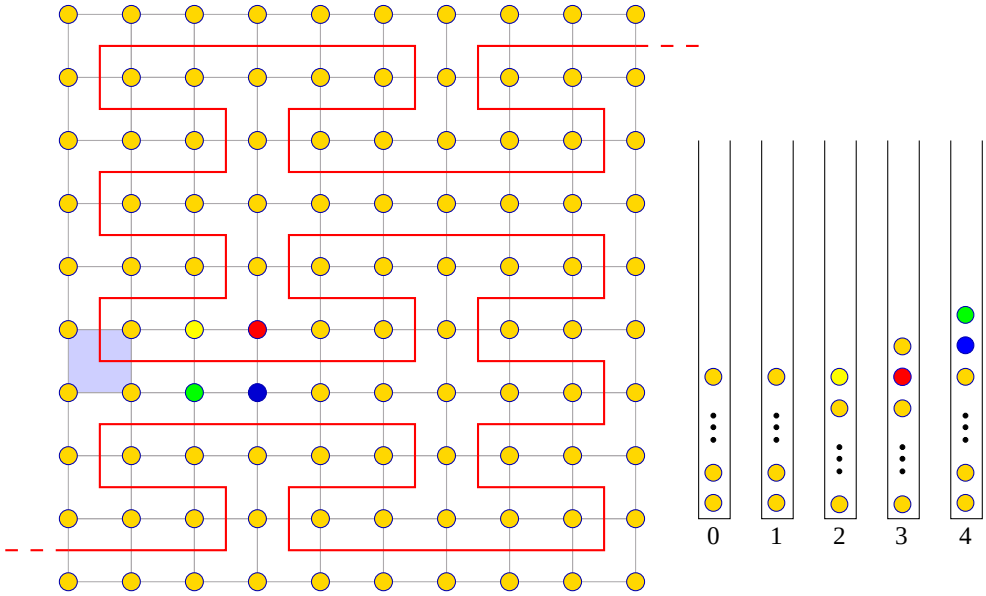

Figure 8: image sequence 3/3

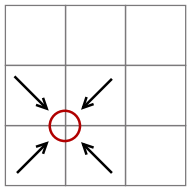

Figure 9: access on one node

3 Parallelization

One of the most frequently used approaches of parallelization within numerics is the *partitioning* of a discretized domain Ω. Partitioning can be regarded as the task to divide Ω into parts and to organize communication between the processes, which work on these parts. This section first briefly describes the requirements of a parallelization approach with respect to determine the partitioning. The second part describes algorithms, which produce the partitioning. The last section shows how space filling curves can be used to find a partitioning, which surpasses the traditional algorithms with respect to all requirements.

3.1 Requirements

As mentioned above, parallelization can be seen as partitioning of Ω. As the partitioning plays an important role with respect to runtime and communication efforts, it has to be chosen according to the the following requirements. When using finite element methods (FEM) to solve partial differential equations, many approaches store the information in nodes, which lie on the vertices of the cells (elements). As a node usually is part of several cells one has to access the node each time when calculating within a neighbouring cell (see figure 9 on page 14). This leads to the first requirement. The number of nodes on all borders has to be minimized. Reducing the number of nodes on the border also implies a reduction of communication between two neighbouring processes as all of the node values have to be sent through the network.
Secondly the partitioning algorithm has to be capable to handle unregularly refined grids.

Usually the partitioned parts are computed on different processors. In order to optimize the overall computation time, one has to organize the partitioning in a way, that none of the processors has to wait for others. In other words, the overall idle time should be as small as possible. If a homogenous network of machines is used, all the processors

can handle the same amount of workload. In that case, the partitioning has to assure that all domain parts approximately consist of the same number of elements. This requirement is called load balancing. To sum up a partitioning algorithm has to meet the following requirements:

- Minimize border size of the partitions,
- handle adaptivly unregular refined grids and
- assure load balancing.

3.2 Partitioning with basic algorithms

As the partitioning problem which optimizes all the above requirements is NP complete, one uses heuristics to find convenient solutions. Many of the heuristic algorithms are graph based. Hence the *recursive spectral bisection* uses a graph which represent the finite elements. This graph is now divided using the Fiedler-Vektor. One can read in [**?**] about the astonishing fact that the Fielder-Vektor contains informations, which can be used for partitioning. Some other graph based algorithms are Recursive Coordinate Bisection and Inertial Recursive Bisection, which are named here to enable the reader to find more informations concerning these approaches. Other approaches like *Scheduling* use a slightly more implicite approach to solve the partitioning problem. The Scheduling algorithm works as follows. If one processor has finished its work, it starts to support others to do so. If one processor often helps another one, he adds a part of the others workload to its own. This leads to a fair distibution of workload, but can also lead to big border sizes, as there is no one taking care for that.

3.3 Partitioning using space filling curves

Partitioning a domain Ω using space filling curves is straight forward. As SFCs map n dimensional areas into the one dimension of the curve. The partitioning can be seen as a partitioning of a line into equally sized parts. As this can be done easily, also a fair partitioning can be achived with ease. Furthermore a SFC can handle adaptivly unregularly refined grids as can be seen in figure 10 on page 16. One has to show, that the communication of the parts stays within acceptable bounds. As mentioned in section 2.2 SFCs tend to stay compact and have a small border. Hence also the last requirements are fullfilled. Measurements and comparisons can be found in [**?**] They show that the space filling curve approach leads to faster algorithms, less communication effort due to small borders and to optimal load balancing.

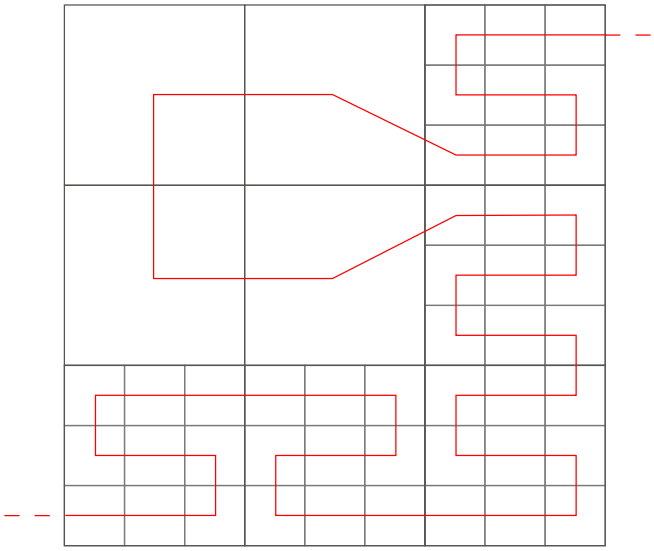

Figure 10: SFC within adaptivly refined domain

3.4 Focus of research

At the moment the chair of Prof. Zenger is developing a FEM solver which applies the approach of the *stack architecture*. The performance results presented in this paper are based on an implementation of Markus Pögl, which uses multi level solvers, hierachical bases, peano curves etc. Right now we are implementing further techniques to show that this approach not only works for Poisson equation on single processors. Following techniques are going to be implemented during this year:

- Solving fluid dynamic problems like Navier Stokes,
- parallelization including adaptivly repartitioning,
- more sophisticated algorithms to improve convergence speed,
- transferring the technique from 2D & 3D to n dimensions and
- refactoring of the framework in order to improve maintainability.