Bone Structure Analysis with GPGPUs

ETH Zürich
Department of Computer Science

Bone Structure Analysis with GPGPUs

Master Thesis

Submitted by:
Daniel Kellenberger

Supervisors:
Prof. Dr. Peter Arbenz
Dr. Cyril Flaig

Responsible Professor:
Prof. Dr. Peter Arbenz

Zürich, June 2, 2013

Abstract

Osteoporosis is a disease that affects a growing number of people by increasing the fragility of their bones. To improve the understanding of the bone, large scaled computer simulations are applied. A fast, scalable and memory efficient solver for such problems is ParOSol. It uses the preconditioned conjugate gradient algorithm with a multigrid preconditioner. A modification of ParOSol is presented that profits from the exorbitant compute capabilities of recent GPGPUs. Different data-structures on the GPGPU are presented and compared. The fastest achieves a speedup of more than 5 while the code is scalable for huge numbers of GPGPUs.

Contents

Chapter 1

Introduction

Osteoporosis is a disease that leads to higher risk of bone fracture due to a decrease of the bone density. Elderly people have a high risk for suffering from this disease. Obviously osteoporosis is taking an important role in recent health science. One approach to analyze the impact of this disease on the bone structure and strength is to run large scale computer simulations. As input high-resolution CT scans are used. These are translated to a structure of tiny cubes (voxels), which then can be investigated. The simulation method is often a finite element calculation, which leads to huge linear or non-linear systems of equations with possibly billions of unknowns[2, 6].

Since these linear systems of equations are often positive definite, they can be solved using a preconditioned conjugate gradient algorithm on large computers with thousands of cores [8].

ParFE, a computer program developed in recent years is able to solve these problems in an efficient way. ParFE uses a smoothed aggregation-based multilevel preconditioner in the preconditioned conjugate gradient algorithm. This resulted in a memory and time efficient solver [1, 2, 6, 9]. On top of this an even faster and much more memory efficient program called ParOSol was developed [3].

The regular structure of the grid in ParOSol offers great possibilities to be exploited by general purpose graphics processing units (GPGPUs or short GPUs). The goal of this thesis is to realize these possibilities. In a first part this is done for a single GPU. This already allows to solve problems with 100 million degrees of freedom. The second part enables the code to use many GPUs to solve really large problems up to 22.5 billion degrees of freedom. Several data-structures and communication schemes on the GPU are proposed, profiled and compared.

Chapter 2

ParOSol

In this chapter a short overview on the structure of ParOSol is given. For further information it is recommended to read the PhD thesis of Cyril Flaig [3, 4].

ParOSol is a "A Highly Scalable Memory Efficient Multigrid Solver for μ-Finite Element Analyses"[3]. In the core it uses a preconditioned conjugated gradient (PCG) solver with a multigrid preconditioner. The preconditioner uses a predefined number of levels. On the coarsest level, the problem is solved using a PCG-solver with a simple Jacobi preconditioner and a constant number of iterations. The preconditioner uses a so called w-cycle which means that on each level two iterations of the next level are applied. Algorithm 1 gives an overview over the preconditioner.

Algorithm 1 Preconditioner

```
b, x, rFine, eFine, rCoarse and eCoarse are vectors
procedure PRECONDITION(b, x, level)
    PreSmooth(b,x)
    for i ← 1, 2 do
        Apply(x,rFine)                          ▷ apply the local stiffnessmatrix
        rFine ← b − rFine
        Restrict(rFine,rCoarse)
        eCoarse ← 0
        if level > 0 then
            Precondition(rCoarse,eCoarse,level - 1)
        else
            Solve(rCoarse,eCoarse)                          ▷ solve with PCG
        end if
        Prolongate(eCoarse,eFine)
        x ← x + eFine
        PostSmooth(b,x)
    end for
    SetDiagonal(x)
end procedure
```

Throughout computation several times a matrix-vector product needs to be applied between the system matrix of the corresponding level and the solving vectors. This can efficiently be done using the local element stiffness matrix

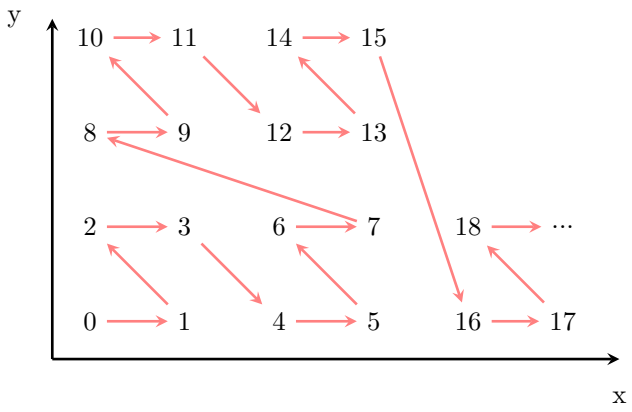


Figure 2.1: A two dimensional z-curve. The self-similarity is easy to see.

described here [2].Therefore at no point in time the matrix describing the linear equation system is formed, which of course saves a lot of memory.

2.1 Morton Ordering

ParOSol uses a Morton ordering [7] for the numbering of the elements and nodes.To understand how the grid in ParOSol is constructed and for many of the changes introduced in this thesis it is crucial to understand how the Morton ordering looks like, what guarantees it provides and how moving in the grid works.

Figure 2.1 shows a two dimensional example of the Morton ordering. Due to it's shape it is also often called z-curve. The curve is self similar and provides a good locality.

An important guarantee given by this ordering is that the key always increases if moved one step in x- or y-direction (or z-direction).

Another important property is that it is possible to move step-wise in x-, y- and z-direction. This is efficiently possible using a few bit manipulations and one addition. Further informations on this step-wise movement can be found in Cyril Flaig's PhD-thesis [3].

2.2 Problem Domain and Grid

The problem domain is embedded in a regular 3-dimensional grid. This results in equally sized voxels which will be called elements. The elements are numbered using a Morton ordering and each element has a Young's modulus (later referred to as weight). At the element vertices nodes are placed and are numbered the same way whereas always the node with lowest key shares its key with the element. The grid itself consists of an ordered list of keys with an attached weight. A negative weight means that this key encodes for only a node, otherwise it is both, a node and an element. After initialization the grid remains constant. Figure 2.3 shows these things for a small two dimensional example.

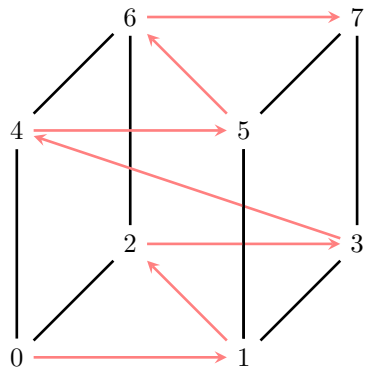


Figure 2.2: The basis pattern of the three dimensional z-curve. The numbers are local keys.

The working data are the displacements. Every node has an x-,y- and z-displacement. These are inserted in a vector in the order defined by the ordering in the grid. During computation there are multiple of these vectors in use.

Since ParOSol uses a multigrid preconditioner, multiple grids need to be initialized. On every level the numbering starts from 0. Thanks to the structure of the Morton ordering, the key of a parent node can be obtained by dividing the key of the child node by 8 (in 3D).

2.3 Working with the Grid

During the computation several operations will rely on the grid. These are explained later in this chapter. Working with the grid usually means iterating over all elements and accessing or modifying the displacements of the nodes.

To do this the keys of the 8 nodes need to be determined. The first key is given by the key of the element. The keys of the other nodes are calculated by moving single steps in x-, y- and z-direction.

Given the keys, their position in the grid is determined. For this step ParOSol uses an improved binary search. With these indexes also the indexes of the desired displacement values are known and the actual computation can start.

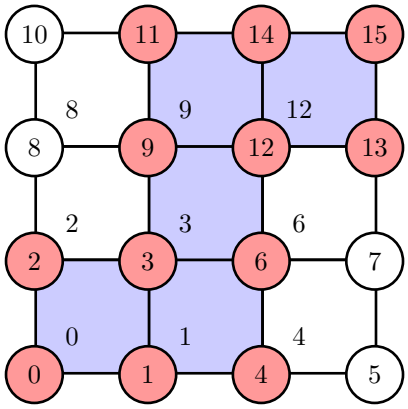
Now the operations which are used during solving in ParOSol are introduced:

2.4 Vector Operations

There is quite a bunch of vector operations working on one or more displacement vectors. Most of them work on a component by component basis. The only exception is the dot product.

2.5 Matrix-Vector Product

The matrix-vector product is one of the operations mentioned earlier which relies on the grid. It is also the computationally most expensive operation



Keys:	0	1	2	3	4	6	9	11	12	13	14	15
E-modulus:	10	10	-1	10	-1	-1	10	-1	10	-1	-1	-1

Figure 2.3: This figure shows some aspects of the grid representation in ParOSol. For the sake of simplicity it's shown in 2D. First the blue squares are elements. Second there are nodes (red) at each vertex. Third the linearized version of the grid: blue encodes for an element and a node and has a positive E-modulus (10 in this example), red encodes for a single node and has a negative E-modulus.

during solving. It is applied on every element separately. After having found the displacements of the nodes in the input-vector, this 24-component vector is multiplied with the precomputed 24×24 stiffness matrix [2]. The resulting 24-component vector is multiplied with the element weight and added to the solution at the correct positions in the output-vector.

2.6 Restrict and Prolongate

The restrict and prolongate operations translate between the different levels of the tree. They also rely on the grid for determining the parent-child relationships. Other than the matrix-vector product they don't always need all 8 nodes.

2.7 Domain Decomposition and Communication

The domain decomposition in ParOSol is done along the ordered set of the Morton keys. Every process gets a continuous piece of the set of about the same size. This means that for a process k all keys (elements/nodes) with a smaller key than its locally smallest key reside on processes with a lower rank than k. It is similar for larger keys which reside on processes with a higher rank.

Since the elements share nodes, communication is needed for the operations that rely on the grid (Matrix-Vector Product, Restrict, Prolongate). The com-

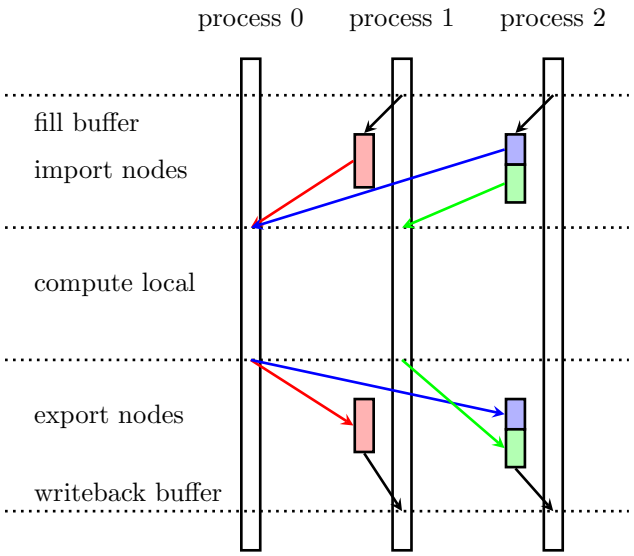


Figure 2.4: The communication graph of a grid operation with three processes. Red and blue: node data that need to be imported by process 0. Green: node data that need to be imported by process 1.

putation of these elements is done on the process which owns the corresponding element.

As mentioned earlier elements share their key with the node with the lowest key. Therefore if one or more of the 7 other nodes are missing on the current process, these nodes must reside on processes with higher ranks. In the first step they are imported. Due to the domain decomposition the imported nodes can be appended at the end of the displacement vector rank by rank of all sending processes and the local grid stays sorted. This is not true for the sending processes. They need to gather the required node data in a send buffer.

Then the computation can be done locally and afterwards the nodes in the result vector that don't belong to this process are sent to their corresponding processes and summed up with the locally computed result. Figure 2.4 shows the communication graph of a grid operation.

Chapter 3

Grid Representation on GPU

In this chapter two different representation schemes of the octree grid on the GPU are shown. The first one, called "Hash Approach" uses perfect spatial hashing [5] for the mapping of the Morton key to the index in the displacement vector. The second scheme, referred to as "Offset Approach" tries to benefit from the locality behavior of the Morton ordering.

3.1 Hash Approach

The hash approach uses the perfect spacial hashing algorithm proposed in [5]. The paper describes the algorithm for 2- and 3-dimensional domains. It is adapted for the 1-dimensional case. This is because the used Morton ordering linearizes the 3 dimensional domain to 1 dimension. This also reduces the instruction count for the evaluation of the hashing function.

The main idea behind this algorithm is to use two hashing functions with a main hash table H and and an offset table Φ:

$$h(k) = h_0(k) + \Phi[h_1(k)] \mod size_H$$

$h(k)$ returns the index of the desired value in H. H is hopefully not much bigger than the number of elements to store.

In the 1 dimensional case, the two hashing functions are defined as follows:

$$\begin{aligned} h_0(k) &= k \mod size_H \\ h_1(k) &= k \mod size_\Phi \end{aligned}$$

$size_H$ and $size_\Phi$ are the sizes of the hash table and the offset table respectively. To ensure uniqueness of $h(k)$, $size_H$ and $size_\Phi$ have to be coprime.

3.1.1 Task of the Hash

The task of the hash is to provide a mapping between a Morton key of a node and its corresponding index in the displacement vector. As the name of the hash tells, it should be possible to build a hash table that has good memory

efficiency and contains few empty elements. In this case it didn't work out well. The occupancy for large domains is on the order of only 50% or less.

3.1.2 Construction

The constructions is done completely on the CPU and has three main steps:

1. determine the hash table size
2. determine the offset table size
3. fill in the tables

These 3 steps rely on each other, which basically means that after a hash table size and an offset table size are defined, it is tried to find a valid population. If none is found try another, larger offset table size. If still no valid population can be found enlarge the hash table size. Of course this works also in the other direction (reduce the size if a valid population is found).

This cries for a binary search for optimal table sizes [5]. Since filling in the tables is an expensive operation, this is not the way it is done. It's still good enough if the table sizes are near to optimal. Therefore greedy algorithms are implemented which determine a start size and in case of failure increase this size. For the hash table size this is done linearly and for the offset table size geometric progression is used (as proposed in [5]).

If these sizes are determined, the filling in of the hash table starts. This is pretty much done as proposed in [5]. They propose 3 steps which are executed for every input value:

1. try to put neighbours in the domain in a way that they are also neighbours in the hash table
2. try to assign them the same offset in the offset table as their neighbours in the domain
3. try random offsets until it fits

If none of these steps succeed, try again with an other random seed for a few times. In case none of these tries is successful, increase the offset table size or even the hash table size.

3.1.3 Multi Hash

The construction of the hash is expensive and not or hard to parallelize. Therefore only one of possibly many cores of the CPU can be used. Instead of parallelizing the construction of a single hash (consisting of two hash tables), the set of keys is split in n_h disjunct subsets. For each subset a hashing function is generated. These hashing functions can be generated in parallel to speed up construction and use more of the available hardware resources. The number of hashing functions n_h is given by

$$n_h = 2^{\lfloor \log_2(n_c) \rfloor}$$

where n_c is the number of cores on the CPU. With this constraint, dividing the set of keys in n_h distinct sets is easily possible by using the modulo operation

on the keys with n_h as divisor. This operation can efficiently be implemented by one bitwise and.

To distinguish between the different hash approaches later in this thesis hash means that only one hashing function is used and hashx means that x hashing functions are used.

3.1.4 Grid Representation

For the hash approach the grid representation reduces to two separate arrays. The first array contains element keys for the matrix-vector product. At the end of this array also the keys of the nodes are put, which are additionally used for the restrict and prolongate operation. The second array contains the element weights (only for the elements). The grid is completed by an hash table and an offset table per generated hashing-function.

3.1.5 Memory Consumption

The total memory consumption can therefore be calculated by summing up the sizes of the key array and the hash tables:

- key array, 8 byte/key
- hash table, 4 byte/entry
- offset table, 2 byte/entry
- weight array, 8 byte/key

Since the hash table size can't be predicted it's not possible to estimate the memory consumption a priori. The lower bound is given by:

$$key + hashtableentry + weight = 8 \text{ byte} + 4 \text{ byte} + 8 \text{ byte} = 20 \text{ byte}$$

This corresponds to an hash table that has no empty elements and an offset table that has only one entry.

3.2 Offset Approach

Instead of computing the keys of the vertices of an element every time, the offset approach works with precomputed indexes in the displacement vector. If the indexes of all 8 nodes are simply stored for every element this would be too memory inefficient ($8 * 4$ byte $= 32$ byte/element). By exploiting the good locality behaviour of the Morton ordering, a grid representation is suggested which is memory efficient and needs only a few operations.

The Morton ordering leads to the following observation: the difference between the smallest and the largest index of an element is small in most cases. Empirically it is found that for 65%+ of all elements this difference is less than 256 and for 95%+ of all elements the difference is less than 65536. In table 3.1 the distribution over these cases is shown for some sample problems.

Therefore 3 kinds of elements are defined, each having it's own storage scheme.

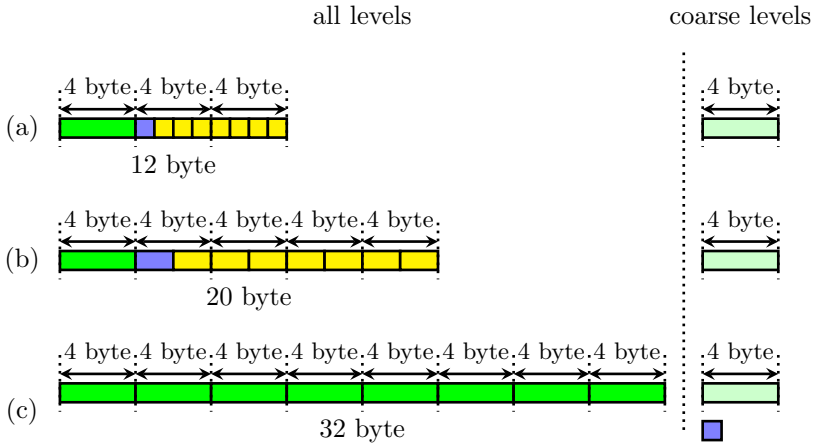


Figure 3.1: Element representations: (a) short elements, (b) medium elements, (c) long elements. Left: layout for all levels. Right: additional data on the coarse levels.

3.2.1 Short Elements (Distance < 256)

First the index of the smallest (first) vertex of the element is stored. This is the entry point. Since the distance of all vertices to this entry point is less than $256 = 2^8$ (by definition), this offset can be stored in one byte. Due to alignment reasons the differences of all vertices are stored (including the first which is always zero). This gives a memory usage of $4 \text{ byte} + 8 * 1 \text{ byte} = 12 \text{ byte}$ for an element. Figure 3.1 (a) shows the layout.

During computation the desired indexes are calculated by adding the offset of the corresponding node with the index of the entry point.

3.2.2 Medium Elements ($256 \leq$ Distance < 65536)

The storage scheme is pretty much the same as in the previous case, but this time two bytes instead of one are used for every offset. This leads to a memory usage of $4 \text{ byte} + 8 * 2 \text{ byte} = 20 \text{ byte}$. Figure 3.1 (b) shows the layout.

Index computation is the same as in the previous case.

3.2.3 Long Elements (Difference ≥ 65536)

In this case nothing can be gained by splitting the indexes into an entry point and an offset. The indexes of the 8 vertices are stored directly, which leads to a memory usage of $8 * 4 \text{ byte} = 32 \text{ byte}$. Figure 3.1 (c) shows the layout.

3.2.4 Grid Representation

The grid is a list containing all elements in the following order: long elements $\rightarrow$ medium elements $\rightarrow$ short elements. There is also a second list containing the corresponding element weights. The exact memory layout of this list is described in chapter 5.

Sample	offset approach				hash approach	
	p_{short}	p_{medium}	p_{long}	size	hash	hash8
cubeb1000	0.739	0.233	0.028	14.4	18.5	16.7
cubeb4000	0.737	0.230	0.033	14.5	16.7	17.2
cubesolid1000	0.655	0.296	0.049	15.4	-	17.9
cubesolid4000	0.645	0.302	0.053	15.5	-	17.7
sphere	0.654	0.313	0.033	15.2	20.4	16.7
cubeb16g	0.733	0.229	0.038	14.6	-	18.2

Table 3.1: Distribution of the different elements categories and their average size for some problems compared to the average sizes per element of the hash approaches. The size of the element weight is not included. Sizes in bytes.

3.2.5 Memory Consumption

The total memory consumption can be calculated by weighting the size of every kind of element with its occurrence and 8 byte for the weight. The memory consumption per element e_{size} in byte is given by:

$$e_{size} = p_{short} * 12 + p_{medium} * 20 + p_{long} * 32 + 8$$

Please refer to table 3.1 for the numbers of a few sample problems.

3.2.6 Restrict and Prolongate

Using the offset approach, any information in the Morton keys about parent-child relationships between the different levels is lost. Therefore some additional data structures are needed to overcome this issue. But again the structure of the Morton ordering helps.

Other than in ParOSol or in the hash approach the restrict and prolongate operations do not iterate over the fine grid, but over the coarse grid. This means a parent element now searches its children. The keys of all possible children of a parent element with key k are $\{8*k, 8*k+1, 8*k+2, ..., 8*k+7\}$. Therefore all children of a specific element are contiguous in the displacement vector. For every element on the coarser grids the index of the first child in the fine displacement vector is stored (figure 3.1 the light green block on the right side).

The only things that is missing is the knowledge, which of the children do exist. This is stored in a small 8 bit bitmap. This bitmap fits nicely in the first offset of the short and medium elements, since this offset is always zero by definition (figure 3.1 the blue blocks). For the long elements these bitmaps are stored in an extra array.

3.3 Discussion

In this section a short discussion is given about the two proposed grid representations, their memory consumption and their initialization cost.

Comparing the sizes in table 3.1 the offset approach generates in all cases a 13 to 25% smaller grid (in terms of memory consumption). For really dense

sample	offset app.	hash app.	
		hash	hash8
cubeb1000	5.25	35.5	10.1
cubeb4000	21.6	173	43.4
cubesolid1000	9.71	-	12.0
cubesolid4000	37.8	-	14.4
sphere	0.527	8.29	0.910
cubeb16g	174	-	413

Table 3.2: Times for preparing the grid for the GPU for some sample problems. Times in seconds.

Sample	Sum	Grid	Disp. Vectors	Other
cube128	58.8	3.32	40	15.4
cubesolid	54.9	7.75	40	7.12

Table 3.3: Sizes in bytes per degree of freedom for two different kinds of problems. The sizes of the grid and the displacement vectors are given for the finest grid. The sizes of the coarser grids and the corresponding displacement vectors are summed up in other.

problems the hash approach can't even generate a grid, because the hash tables become to big. Also the construction of the hash is much more expensive (see table 3.2 for some timings).

Expanding the hash approach to multi hash overcomes some of these problems. In most times the generated grid is smaller than with only one hash (but still larger than the one from the offset approach) and the time for the construction is reduced strongly.

Looking at the memory consumption of the whole application on the GPU (see table 3.3) it can be seen that impact of grid size is rather small. Therefore all three approaches could be used in terms of memory consumption for porose samples. For regular problems the size of the hash tables increases strongly and the size of the grid begins to matter. Also initialization cost matters and more importantly the evaluation cost of the grid, which will be shown later in the results chapter.

Chapter 4

Multiple GPUs

So far everything was done for one GPU and therefore involved no inter-CPU communication and only the dot-product uses GPU-CPU communication to return its result. In this chapter it will be shown how this code can be brought to run on multiple GPUs (on distinct nodes). Basically the CPU is in charge of the communication and uses MPI for this purpose.

4.1 Straightforward Solution

In a first version the structure for interprocess communication is reused as given by ParOSol [3]. This means that in a first step the node data that are needed by other processes are gathered in a send buffer, then sent and on the receiving side appended at the end of the vector. Computation starts when the import phase has completed and ends before the export phase starts.

The main difference to ParOSol is that before sending, the data needs to be shifted from GPU memory to main memory and vice versa after receiving.

4.2 Reordering the Front of the Displacement Vector

In ParOSol communication there are two cases:

- The sending process gathers the necessary data and sends it whereas the receiving process simply puts the data a the end of its vector.
- The sending process sends the data from the end of its vector and the receiving process adds the data at the correct positions.

The sending and receiving of the data at the end of the vector is perfectly fine for the GPU, but the gathering and redistributing of the data could be improved. It results in an extra kernel launch or bigger data transfers between GPU and CPU.

Therefore it is proposed to rearrange the displacement vector to get around this problem. In a first step this means that all components that need to be sent are placed at the front of the vector, in the order they are sent. This

way no extra kernel is needed and the data transfer between GPU and CPU is minimized. The only special case that needs to be taken care of is when a node needs to be sent to two or more different processes. These nodes are duplicated. In the displacement vector on the GPU the first node (the node with the lowest index) holds the displacements where as all duplicates are set to 0 [1]. The CPU handles these duplicates before sending.

Since GPU memory transfers and kernels can run in parallel this enables overlapping of communication and computation.

4.3 Reordering the Back of the Displacement Vector

It's still necessary to insert a synchronization point somewhere to ensure that the imported elements at the end of the vector have fully arrived. To tackle this issue also the back of the displacement vector is reordered. The basic idea is to put all nodes which depend in any way on the imported nodes (except those already put in the front) in the back of the vector, immediately before the imported nodes.

Now instead of transferring the imported nodes to the GPU, the tail of the vector is transfered from the GPU to the CPU. Now all nodes that depend on the imported nodes are computed on the CPU and after computing their part of the solution are transfered back to the GPU and added.

This way no synchronization point on the GPU is needed and in the ideal case everything is completely overlapping, meaning that transferring front and tail to the CPU, exporting step, computing the tail nodes, importing step and adding the partial solution from the CPU is faster than the grid operation on the GPU. Figure 4.1 shows both parts of the reordering (front and back).

4.4 Impact on Grid Representation

Of course such a reordering has an impact on the grid representation.

4.4.1 Hash Approach

For the hash approach the impact is rather small. There are less keys that need to be inserted in the hash (imported nodes). Also the keylist representing the grid shrinks, since some of the elements are computed on the CPU.

4.4.2 Offset Approach

The impact for the offset approach is bigger. It's not guaranteed anymore that the first vertex has the smallest offset in the displacement vector. If this happens the element is defined as long element, since for the long elements it doesn't matter where in the displacement vector the vertices reside. This has only a small impact on the distribution of the three kinds of elements (see table 3.1, numbers of cubeb16g).

[1] These nodes can't be found by the grid operations. Since they are 0 they do not affect the dot-product.

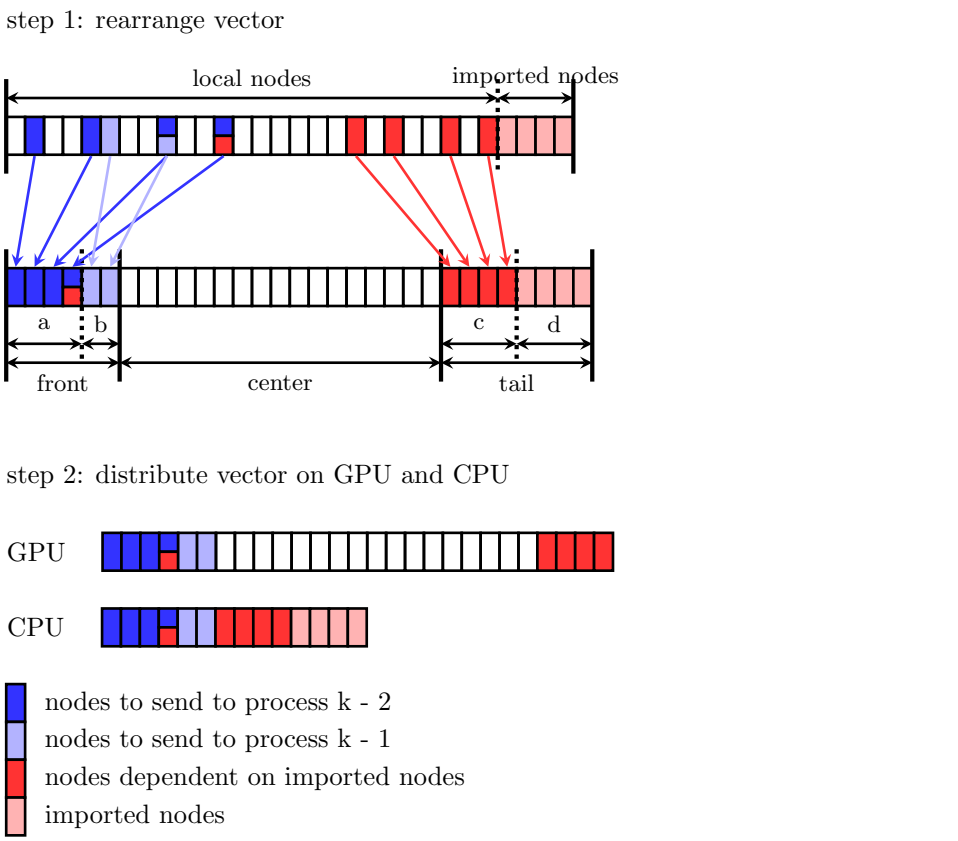


Figure 4.1: The 2 steps for rearranging the displacement vector on process k. In the import step, parts a, b and c are transfered to the CPU memory, then parts a and b are sent to process $k-2$ and $k-1$ respectively and part d is received from processes with a higher rank. In the export step part d is sent to processes with a higher rank and part a and b are received from the corresponding processes. Then parts a, b and c are transfered to the GPU memory and added.

Chapter 5

Implementation

In this chapter it is explained how important parts of the program are implemented with CUDA.

5.1 CUDA

For people who aren't familiar with CUDA, a short introduction is given on how CUDA works.

CUDA is an extension for programming languages like C, C++ and Fortran developed by NVIDIA. It introduces models to run code on GPGPUs [1].

Functions called from the host (process on CPU) are called kernels. Instead of one thread that executes the code (like on the CPU) a whole grid of threads is launched, all executing the same piece of code. Typically every thread works on its own piece of data which can be identified by the thread id. A grid is divided into blocks. Threads within blocks can communicate via shared memory. Blocks are again divided into warps. In recent GPUs a warp consists out of 32 threads which are executed in SIMD-fashion.

There are a lot of functions that make the life of programmers easier and allow to implement more complex algorithms, for example global atomic operations. Also the set of these functions is increasing with newer GPUs.

To benefit from the GPU it is important to split the work into thousands of threads since there are also more than thousand cores that execute the threads and, because of the latency of the operations (16 cycles), it is favorable to have more than 16 times more threads than execution cores.

To have many threads also helps to overcome memory latencies that are typically bigger than on CPUs. This means while one thread is waiting for data another can jump in.

5.1.1 Memory Hierarchy on the GPU

The memory hierarchy on the GPU differs from the one known from CPUs. First of all there is the large but slow off-chip memory. There are multiple ways how the data can be transfered from there to the requesting threads. The first step is always the L2-Cache. Then three paths are possible:

[1] see: `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

- Through the L1-Cache. This is the only read-write path.
- Through the read-only-cache and the texture unit.
- Through the read-only-cache.

By default the first path is chosen. The texture unit is an extra piece of hardware that allows filtering of the requested data. The use of it needs to be specified explicitly by the programmer. The third path is chosen by the compiler automatically if it is guaranteed (by the programmer) that the data are not modified.

Other types of memory that are available are shared memory and constant memory. Shared memory is accessible for all threads in a block and allows communication between the threads. The constant memory is a small block block of read-only memory for the whole device. It needs to be allocated explicitly by the programmer.

5.2 Grid on GPU

5.2.1 Hash Approach

The grid for the hash is simple. There is an hash table and an offset table (one dimensional arrays) for every hashing function, the array with the element keys and an array with the weights of the elements.

It gets a little bit more complicated to evaluate the hashing function. First the corresponding hashing function is determined. Since the number of hashing functions is a power of two, the modulo operation can efficiently be implemented using a bitwise "and" operation. Then the hashing functions themselves are evaluated. These are modulo operations, too, but usually not powers of two and they are unknown at compile time so the compiler can't help. Therefore the whole modulo operation needs to be done, which is slow on GPU. Furthermore the keys are 64 bits wide which makes the operation even more costly.

Since only the dividends are 64 bit and the divisors fit into 32 bit, optimizations can be applied. It is possible to replace one 64 bit modulo operation with three 32 bit wide ones, which happens to be cheaper.

Any 64 bit integer x can be represented as $x = a * 2^{32} + b$ where a and b are 32 bit integers. Therefore we can define:

$$x \mod n := ((a \mod n) * (2^{32} \mod n) + (b \mod n)) \mod n$$

given that $n < 2^{32}$. $(2^{32} \mod n)$ is constant after initialization of the hashing function and can be precomputed. Now the hashing function is evaluated.

5.2.2 Offset Approach

In the offset approach the evaluation becomes simple (one addition), while the grid representation gets rather complicated.

A short reminder: There are three kinds of elements.

- short elements, 12 byte wide
- medium elements, 20 byte wide

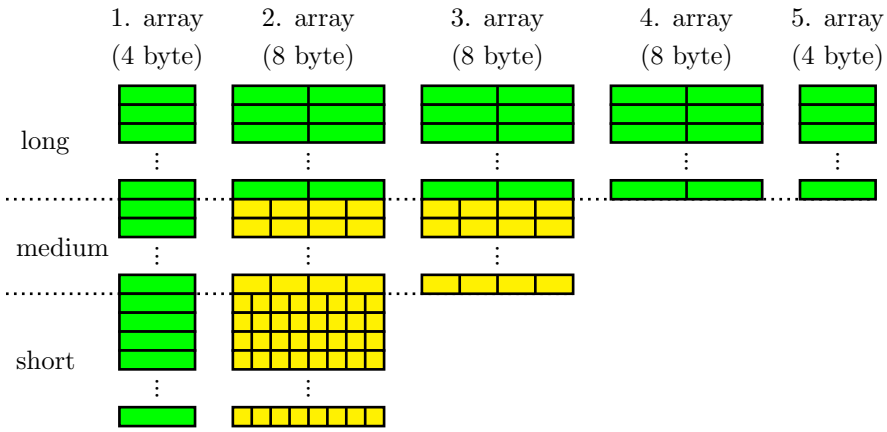


Figure 5.1: The struct of arrays of the finest level.

- long elements, 32 byte wide

To avoid possible overheads by 3 different kernel launches for the three different kinds of elements a single datastructure is defined that allows to insert all kinds of elements. A key idea behind this data structure is to use a struct of arrays instead of an array of structs. This is especially important for the GPU because of its 32-SIMD execution fashion. Each of the 32 threads of a warp is at the same instruction and requests the same part of an element, e.g. the index of the first node (entry point). If these reside contiguous in memory the request can be served faster.

The elements are inserted in the data structure one after the other, starting with the long ones, then the medium ones and finally the short ones. Figure 5.1 shows the 5 different arrays. The first 4 bytes of an element are inserted in the first array, the next 8 bytes in the second array and so on until the whole element is stored. Of course the medium and short elements don't need all the arrays.

A thread that wants to evaluate the indexes of an element checks first what kind of element to read. Since the number of each kind of elements is known this can be determined with the thread id. Then the thread reads the arrays one after the other using its own id as index in the array. Always the whole block in an array is read. For the short elements this means that with reading the second array all offsets are already read and loaded in the thread local registers, where they are split and summed up with the start index.

5.3 Matrix-Vector Product

Every thread handles exactly one element of the matrix-vector product on its own. This means computing the indexes of the nodes, loading the displacements, applying the 24×24 matrix-vector product and adding the results to the output vector.

There are several reasons not to apply finer grained parallelism. First of all there are usually enough elements available to keep the cores on the GPU

busy. Any splitting of a thread would introduce a synchronization overhead. Only the coarsest levels could possibly profit from dividing the work further since there are much fewer elements to be computed. But since the most time is spent on the finest level there is not so much to be gained. An other reason is the way the local stiffness matrix is stored. This matrix is stored in the small (64 kB for the whole GPU [2]) but fast constant memory. The constant memory is read-only during a kernel launch but has a small latency. Furthermore it has a broadcast mechanism. It can serve half a warp (16 threads) at once if all threads request the same piece of memory. If every thread requests another piece of memory the requests become serialized. When every thread in a warp (or half a warp) computes its own matrix vector product, all threads always request the same element of the local stiffness matrix and can profit from the broadcast mechanism. If multiple threads work on a single matrix vector product this benefit is lost.

Of course there is also synchronization necessary between the elements, since the elements share nodes. Up to 8 threads could possibly access the same node when writing back the result. This is solved using atomic operations. The maximal contention on a memory location is with a maximum of 8 requests at the same time rather small. Furthermore, elements that share a node are often within the same warp, which means they always access their locally same node. The locally same node (e.g. the first node) is globally never the same for two elements. This also reduces the contention.

[2] This is enough to store the local stiffness matrices for up to 13 levels.

Chapter 6

Results

In this chapter the results of the performance measurements are presented. It's split into two sections, first on a single GPU with different problem sizes up to 72 million degrees of freedom. Second on a GPU-cluster with up to 256 GPUs and problem sizes up to 22.5 billion degrees of freedom. The different grid representations are compared with ParOSol as baseline.

In all performance tests the problem was solved up to a precision of 10^{-5} with 5 levels for the preconditioner. In tests that involved more than 64 nodes the computation was stopped after 20 iterations to avoid using too many computational resources.

6.1 Single GPU

6.1.1 Hardware and Baseline

The test machine (called Obiwan) is a small CPU-cluster with one attached GPU. Obiwan has a total of 16 Intel Xeon E5-2660 (@ 2.2 Ghz) CPUs with 8 cores and 16 GB DDR3 memory each and one NVIDIA GeForce GTX 670 GPU with 4 GB of GDDR5 memory.

The baseline to compare with is given by Cyril Flaig's implementation of ParOSol [4] and is run using 8 processes. It's considered fair to compare one GPU with one 8-core CPU.

Sample	**offset**		**hash**		**hash8**	
	iter	total	iter	total	iter	total
cube128	5.11	2.45	4.83	1.86	4.41	2.17
cubesolid	5.07	1.39	-	-	4.66	1.35

Table 6.1: Average speedups in time per iteration and total run-time compared to ParOSol for both kind of problems.

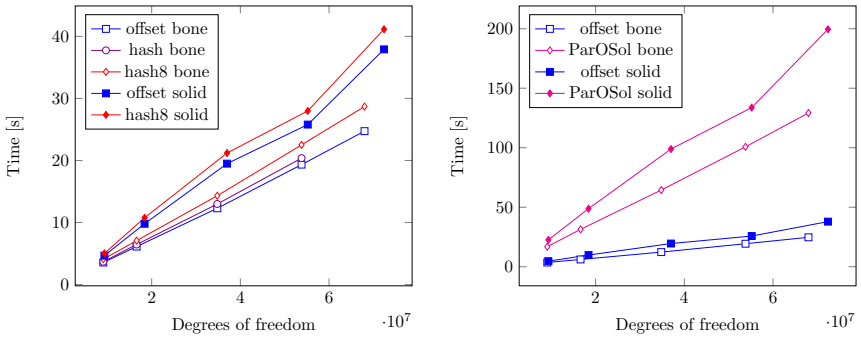


Figure 6.1: Time per iteration. Left: the different GPU-grid representations for different problem sizes. Right: the best GPU-grid representation against ParOSol (baseline) for different problem sizes.

6.1.2 Time per Iteration

In figure 6.1 the measured timings per iteration are shown. In the plot the different approaches are applied to two different kinds of problems and for different numbers of degrees of freedom each. It can be seen that the offset approach outperforms the hash approach slightly and the multi hash approach clearly in any case. Also the linear scaling with the problem size is visible.

Comparing now the offset approach with the timings from ParOSol as baseline, an average speedup of 5.11 for the bone samples and 5.07 for the solid samples is obtained (see table 6.1).

Discussion

First of all there is a large gap between the timings of the bone samples and the solid samples. The reason for this is an increased number of elements per degree of freedom for the solid samples since the surface is much smaller than the surface of the bone samples. Beside from that the time per iteration grows linearly with the number of degrees of freedom. This is a bit surprising since one could think that the workload for GPU is to small for the smallest samples which would result in an increased time since not all of the available hardware is used.

The measurements show that there is no big difference between the offset approach and the hash approach. But the offset approach is more flexible. The hash8 approach is too slow.

The speedup of more than 5 against ParOSol as baseline is very nice and proves that this kind of simulation profits from GPUs.

6.1.3 Total Run-Time

In figure 6.2 the total run-times of the applications are plotted. The picture changes since initialization and output times show their influences. The hash8 approach can now compete with the offset approach for the solid sample. The average speedup for the solid samples is only 1.39 compared to 5.07 for the time per iteration (see table 6.1).

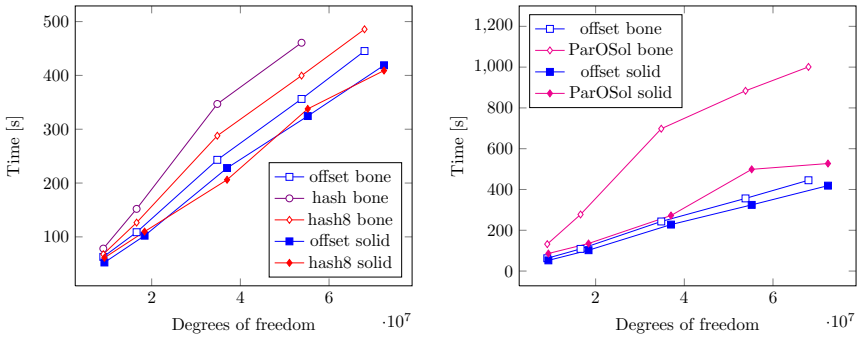


Figure 6.2: Total run-time. Left: the different GPU-grid representations for different problem sizes. Right: the offset approach against ParOSol (baseline) for different problem sizes.

Discussion

When looking at the total run-times the picture is not as good, because the initialization phase can consume fairly large parts of the total runtime. This especially true for the solid samples since there are only 2 to 3 iterations necessary to solve the problem, which results in a smaller parallel section.

Because of the high initialization cost the hash approach loses most of the good performance while solving.

Compared to ParOSol much of the speedup is lost. This can again be explained by the size of the parallel section since also the bone sample needs only 7-10 iterations. With an average speedup of ~ 2.5 for the bone samples still a strong reason is given to use GPUs to accelerate the solving of these problems.

6.2 Multiple GPUs

6.2.1 Hardware and Baseline

The test machine for multiple GPUs is a Cray XK7 called Tödi at the CSCS in Lugano [1]. Tödi has a total of 272 nodes, equipped with one 16-core AMD Opteron CPU with 32 GB DDR3 memory and one NVIDIA Tesla K20X GPU with 6 GB of GDDR5 memory each.

The baseline to compare is again ParOSol [4]. To achieve a fair comparison, this time ParOSol gets 16 processes for each node. Therefore one 16-core CPU is compared with one GPU.

6.2.2 Weak Scaling

Weak scaling means that the size of the problem grows linearly with the number of hardware resources. Ideally the runtime stays constant. Figure 6.3 shows the measurements done. Shown is the time used per iteration. The different problem sizes have different iteration counts and it makes little sense to compare the total

[1] see: `user.cscs.ch/hardware/todi_cray_xk7`

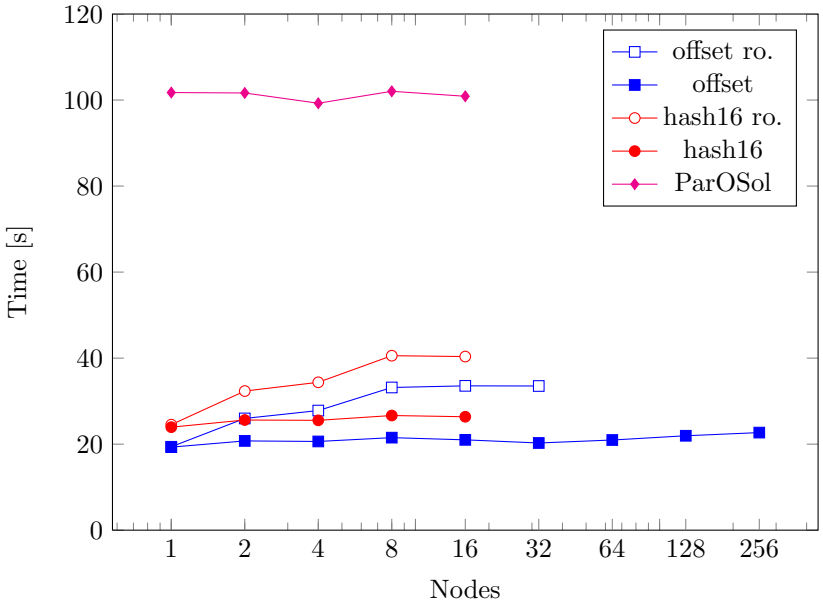


Figure 6.3: Time per iteration.

runtime. The number of degrees of freedom for each problem can be found in appendix A. “cubebxg” is the problem generated for x nodes.

The measurements show no increase of time per iteration for ParOSol as well as for the the straightforward versions (offset and hash16). For the implementations with reordering the time per iteration increases up to 8 nodes and then stays on this level.

The straightforward offset approach achieves an average speedup of 4.91 compared to ParOSol. The hash16 approaches take about 5 seconds more per iteration compared to the corresponding offset approaches.

Figure 6.4 shows the time used to initialize the grids on all levels. For ParOSol this time is stable for any number of nodes. For the plain offset and hash16 approach the time is not as stable as for ParOSol, but a stable trend is visible. For the reordered offset and hash16 approaches, the time increases strongly up to 8 nodes, then it seams they become stable, too.

Discussion

It can be seen that ParOSol scales very well as already shown by Cyril Flaig [3]. It’s therefore not surprising that the plain approaches scale as well. After an initialization phase also the reordered approaches seem to scale, but other than expected they are slower in all cases compared to the straightforward approaches (except for only one node, where no reordering takes place). Profiling shows that on the finest grid the overlapping works as expected, but on the coarser grids it doesn’t. Further investigations showed that the computation on the CPU is the bottleneck. Parallelizing the work on the CPU (using OpenMP) didn’t help, it even got worse.

As already seen in the previous section the hash16 approach can not compete

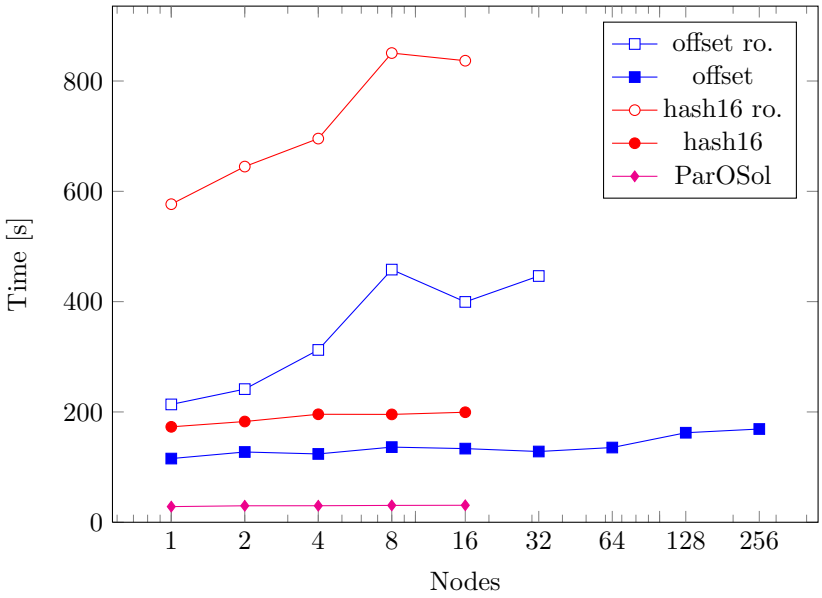


Figure 6.4: Time for initializing the grids on all levels.

with the offset approach.

The initialization times give another reason to stick to the straightforward solution since the reordering takes up to 4 times longer. The additional overhead taken for preparing the grid for the GPU with the fastest approach opposed to ParOSol is in the order of the time saved during one iteration.

6.2.3 Strong Scaling

In strong scaling the number of compute hardware is increased while the problem size is fixed. The measurement is done with three different problem sizes.

- 100M degrees of freedom (cubeb1gs), fits on 1 GPU
- 355M degrees of freedom (cubeb4g), fits on 4 GPUs
- 715M degrees of freedom (cubeb8g), fits on 8 GPUs

The plots in figure 6.5 show the time per iteration and the total runtime on the left and the corresponding parallel efficiencies on the right. For up to 8 times increased hardware resources a strong decrease in time per iteration and total runtime is visible. Using more resources the gain in time is reduced.

Discussion

The strong scaling test shows what can be expected. The first increment steps result in a reasonable speedup but then the time spent on communicating increases, resulting in savings of a few seconds for doubling the hardware resources.

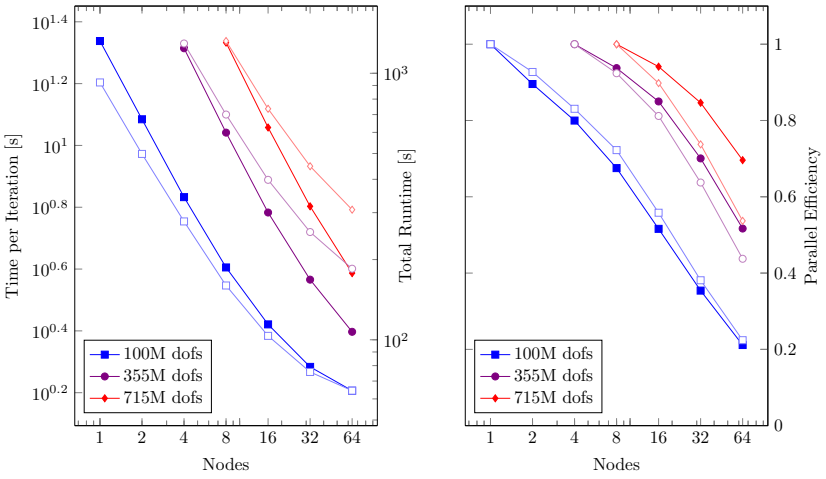


Figure 6.5: Timings for three differently sized problems on the left and their corresponding parallel efficiencies on the right. The dark graphs (filled marks) belong to the time per iteration measurement, the lighter ones (empty marks) belong to the total runtime.

Chapter 7

Summary

Out of all presented configurations the offset approach as grid representation combined with the straightforward communication scheme inherited from ParOSol gives the best results. The measurements of this configuration with the achieved speedup give strong evidence for using one ore more GPUs to accelerate the solving of the targeted problems.

One possible drawback is the lack of memory on recent GPUs. On the given hardware with 32 GB of main-memory and 6 GB of memory on the GPU per node, ParOSol would be able to solve up to 3.5 times larger problems. Another drawback is that during computation only one core of the CPU is used.

7.1 Future Work

To allow the solving of larger problems on a single GPU, one could try a streaming approach. For the matrix-vector product this would mean that while a chunk of elements is computed, the previous chunk is transfered to CPU and the next chunk is transfered to the GPU, then the next chunk of elements is computed and so on, until all elements are computed.

One could try heterogeneous MPI-ranks with ranks that have a GPU and ranks that have one core of the CPU. This would allow to use more cores on the CPU. For this the domain composition of ParOSol needs to be modified.

Appendix A

Samples

The bone sample problems are generated by 3D-mirroring of the basis element shown in figure A.1. After that the problems are cropped to the required dimension.

The solid sample problems are fully filled cubes of the given dimension.

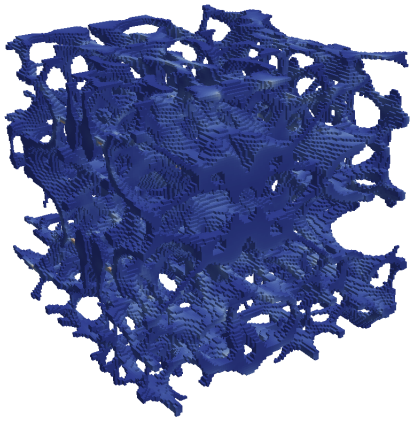


Figure A.1: The basis element for the construction of the bone samples.

Name	DoFs	Dim	Iter
cubeb500	9.09M	256	7
cubeb1000	16.6M	320	8
cubeb2000	34.8M	403	10
cubeb3000	53.8M	458	8
cubeb4000	68.0M	500	7
cubeb1g	86.3M	562	18
cubeb1gs	100M	600	24
cubeb2g	182M	708	42
cubeb4g	355M	892	42
cubeb8g	715M	1123	38
cubeb16g	1414M	1414	40
cubeb32g	2710M	1775	73
cubeb64g	5571M	2243	67
cubeb128g	10920M	2820	20
cubeb256g	22481M	3584	20
cubesolid500	9.34M	145	3
cubesolid1000	18.4M	182	2
cubesolid2000	37.0M	230	2
cubesolid3000	55.2M	263	3
cubesolid4000	72.4M	288	2
sphere	1.46M	101	3

Table A.1: The different test problems used with their number of degrees of freedom and their dimension. The dimension is always equal in all three directions. Iter is the number of iterations required to solve the problem up to a precision of 10^{-5} with 5 levels for the preconditioner.

Bibliography

[1] The parfe project home page. `http://parfe.sourceforge.net`.

[2] P. Arbenz, G. H. van Lenthe, U. Mennel, R. Müller, and M. Sala. A scalable multi-level preconditioner for matrix-free μ-finite element analysis of human bone structures. *International Journal for Numerical Methods in Engineering*, 73(7):927–947, 2008.

[3] C. Flaig. *A Highly Scalable Memory Efficient Multigrid Solver for μ-Finite Element Analyses*. PhD thesis, ETH Zürich, 2012.

[4] C. Flaig. Parosol project home page, 2012. `https://bitbucket.org/cflaig/parosol`.

[5] S. Lefebvre and H. Hoppe. Perfect spatial hashing. *ACM Transactions on Graphics (TOG)*, 25:579–588, 2006.

[6] U. Mennel. A multilevel pcg algorithm for the μ-fe analysis of bone structures. Master's thesis, Institute of Computational Science, ETH Zürich, Mai 2006.

[7] G. M. Morton. A computer oriented geodetic data base; and a new technique in fil sequencing. Technical report, IBM Ltd., 1966.

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[9] A. J. Wirth, J. Goldhahn, C. Flaig, P. Arbenz, R. Müller, and G. H. van Lenthe. Implant stability is affected by local bone microstructureal quality. *Bone*, 49(3):473–478, 2011.