VIII International Conference on Textile Composites and Inflatable Structures
STRUCTURAL MEMBRANES 2017
K.-U.Bletzinger, E. Oñate and B. Kröplin (Eds)

HIGH SPEED DESIGN AND ANALYSIS OF CABLE-MEMBRANE STRUCTURES ON GRAPHICS CARDS

PÉTER IVÁNYI

Faculty of Engineering and Information Technology
University of Pécs
Boszorkány str 2, Pécs, H-7624, Hungary
e-mail: ivanyi.peter@mik.pte.hu, web page: http://english.mik.pte.hu

Key words: Dynamic relaxation, Parallel implementation, NVIDIA, CUDA

Abstract. This paper discusses a new parallelization approach of the dynamic relaxation method, which is programmed with the NVIDIA CUDA API and executed on the graphics cards (GPU) of a computer. The main advantage of a GPU card is that it has a very large number of computing cores and a separate memory from the computer and they may reside inside a normal desktop computer. However due to architectural simplifications of the GPU systems, synchronization of cores is rather limited. This has a major effect on the parallelization, since the contribution of calculated values at the boundary nodes would require some form of synchronization. This limitation resulted in the new parallelization approach, where the nodes of the finite element mesh are distributed between the cores of the GPU and the elements are "duplicated". The paper discusses the implementation details of this new parallel approach and some performance measurements of the new parallel dynamic relaxation method on GPU systems are also presented.

1 INTRODUCTION

The method of dynamic relaxation, initially invented by A. Day [1] is applicable for different engineering problems, however probably most often it is used for the design and analysis of cable-membrane structures. The method has several advantages compared to other methods. For example there is no need to assemble a global stiffness matrix, as the results are always expressed in terms of the current coordinates. It also means that it can handle geometrically non-linear cases when there are large displacements. The method is iterative therefore one way to speed up the analysis was to parallelize the algorithm. Topping and Khan [2, 3], Topping and Iványi [4] researched the parallelization of the dynamic relaxation method. In these references the approach is mainly using the MPI [5] environment and geometric parallelism.

In the past years significant interest has been generated by the use of the GPU architecture for parallel computing due to their power and wide spread availability. At the same time the special architecture of GPU should be considered when a numerical code is parallelised. For example the GPU has a separate memory, therefore the programmer should consider the distributed programming paradigm. At the same time the GPU itself has many core and generally a shared memory. Despite the shared memory the synchronization of parallel threads is difficult and requires special attention when the parallel algorithm is designed.

An earlier work [6] has already presented a parallel procedure of the dynamic relaxation method for the GPU architecture, however in this paper the method is completely reorganised to take advantage of the architecture. In the proposed parallel implementation the calculation will be based on nodes and not on elements as in the case of the MPI environment.

The paper is organised in the following way. Section 2 introduces the dynamic relaxation method. Section 3 discusses the parallelisation of the dynamic relaxation method with the MPI environment. Section 4 presents the new parallelisation scheme for the GPU architecture. Results and performances of the implementation are presented in Section 5.

2 DYNAMIC RELAXATION

The dynamic relaxation method calculates the dynamic behaviour of a structure by direct application of Newton's second law which states that "*the rate of change of momentum of a body is proportional to the applied force and takes place in the direction of the applied force*".

$$F = M \cdot a = M \cdot \dot{v} \quad , \tag{1}$$

where F denotes the force, M denotes the mass and a or $\dot{v}$ denotes the acceleration. Although the method determines the static solution of a problem, however it traces the fictitious movement of a structure. Since this movement should follow a path, which converges to the final solution as quickly as possible, therefore in this fictitious analysis the masses are also fictitious. Furthermore the mass of the structure is assumed to be concentrated at the joints and an additional viscous damping term which is proportional to the velocity of the joint is included in the formulation. Considering these assumptions at any given time t the out of balance or residual force in the x coordinate direction at joint i may be expressed as follows:

$$R^t_{ix} = M_{ix} \cdot \dot{v}^t_{ix} + C_{ix} \cdot v^t_{ix} \quad , \tag{2}$$

where R^t_{ix} is the residual force at time t in the x direction at joint i, M_{ix} is the fictitious mass at joint i in the x direction, C_{ix} is the viscous damping factor for joint i in the x direction and $v^t_{ix}, \dot{v}^t_{ix}$ are the velocity and acceleration at time t in the x direction at joint i.

The analysis traces the behaviour of the structure at a series of points in time t, $t+\Delta t$, $t+2\Delta t$. Over a time step the velocity is assumed to vary linearly with time, hence the velocity can be expressed

$$v_{ix}^{t} = \left(v_{ix}^{(t+\Delta t/2)} + v_{ix}^{(t-\Delta t/2)}\right)/2 \quad , \tag{3}$$

and the acceleration is assumed constant over the time step hence:

$$\dot{v}_{ix}^{t} = \left(v_{ix}^{(t+\Delta t/2)} - v_{ix}^{(t-\Delta t/2)}\right)/\Delta t \quad . \tag{4}$$

Substituting these equations into Equation 2 the result is:

$$R_{ix}^{t} = \frac{M_{ix}}{\Delta t}\left(v_{ix}^{(t+\Delta t/2)} - v_{ix}^{(t-\Delta t/2)}\right) + \frac{C_{ix}}{2}\left(v_{ix}^{(t+\Delta t/2)} + v_{ix}^{(t-\Delta t/2)}\right) \quad . \tag{5}$$

This equation can be rearranged to calculate the velocity at the new time step $(t+\Delta t/2)$ from the velocity of the previous time step $(t-\Delta t/2)$:

$$v_{ix}^{(t+\Delta t/2)} = v_{ix}^{(t-\Delta t/2)}\left(\frac{M_{ix}/\Delta t - C_{ix}/2}{M_{ix}/\Delta t + C_{ix}/2}\right) + R_{ix}^{t}\left(\frac{1}{M_{ix}/\Delta t + C_{ix}/2}\right) \quad . \tag{6}$$

Using Equation 6 the displacement of joint i in the x direction during time interval from t to $(t+\Delta t)$ is given by:

$$\Delta x_{i}^{(t+\Delta t)} = \Delta t \cdot v_{ix}^{(t+\Delta t/2)} \quad , \tag{7}$$

and the current coordinates may be expressed as

$$x_{i}^{(t+\Delta t)} = x_{i}^{t} + \Delta t \cdot v_{ix}^{(t+\Delta t/2)} \quad . \tag{8}$$

Similar equations can be written for the y and z coordinate directions. In Equation 6 the residual forces can be calculated from the contribution of each element connected to joint i as:

$$R_{ix}^{(t+\Delta t)} = F_{ix} + \sum_{m} T_{ixm}^{(t+\Delta t)} \quad , \tag{9}$$

where m represent the indices of all elements connected to joint i, F_i is the applied loading and T_i is the internal forces.

2.1 The general iteration of the dynamic relaxation method

The general iteration of the dynamic relaxation method contains the following steps:

- Calculation of the out of balance force (residuals) at each node of the structure using Equation 9;
- Calculation of the nodal velocities using Equation 6;

- Calculation of the coordinates of the nodes using Equation 8; and
- Check convergence, for example the kinetic energy is smaller than a threshold ($U_k \leq U_\varepsilon$) .

The kinetic energy can be calculated as:

$$U_k = \sum_i^n \sum_j^m M_{ij} v_{ij}^2 \quad , \tag{10}$$

where n is the number of nodes and m is the number of dimensions.

2.2 Control of the method

To ensure the initial conditions ($v_{ix}^0 = 0$ and $R_{ix}^0 = F_{ix}$) the velocity at time $\Delta t/2$ must be given by:

$$v_{ix}^{(\Delta t/2)} = \frac{\Delta t}{2M_{ix}} F_{ix} \quad , \tag{11}$$

where F_{ix} is the initial external force at joint i in the x coordinate direction. The controlling parameters for the stability of the method are:

- the nodal mass components which may be fictitious;
- the damping factor; and
- the time interval.

Further details about this parameters can be found in [4].

Kinetic damping is an alternative to viscous damping and the kinetic energy of the complete structure is traced as the undamped oscillations proceed and all current nodal velocities are reset to zero whenever an energy peak is detected [4].

3 PARALLEL DYNAMIC RELAXATION WITH MPI

The parallel implementation of the dynamic relaxation algorithm has been investigated in the past[2, 3, 4]. These papers have concluded that geometric parallelism offers the best opportunity for parallelization of the method. Geometric parallelism takes advantage of solving subregions of the structure on a distributed array of processors. Figure 1 shows a partial view of the partitioning of a finite element mesh. The figure shows that the elements are distributed between the partitions, while the nodes on the boundary are duplicated in the partitions. Values at these nodes must be communicated between the partitions, which is also marked in the figure. The parallel algorithm of the dynamic relaxation is organized in such a way, that only the partial contribution of the sum of the forces at the boundary nodes are exchanged. Once the total forces at every node in every

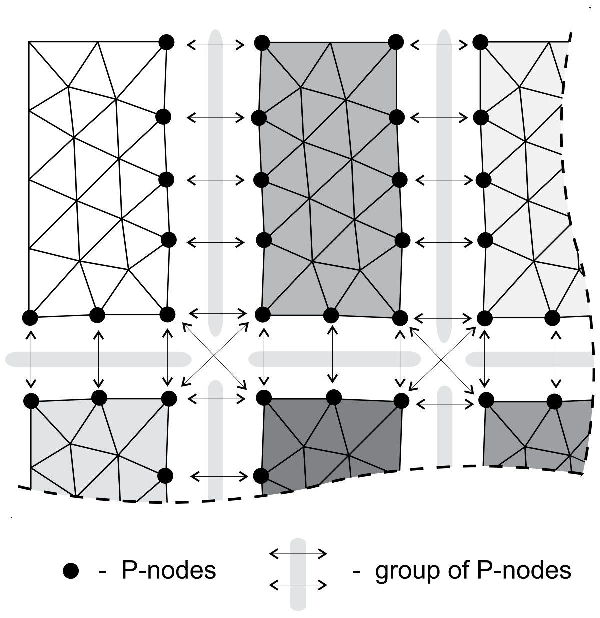


Figure 1: Element based partitioning of a finite element mesh and the required communications between partitions.

partition are known then all other values can be calculated in the partition independently according to Equations 6 and 8. Another communication is also required when the total kinetic energy of the structure must be determined. The iteration for parallel computation with the communication steps can be seen in Figure 2a.

4 PARALLEL DYNAMIC RELAXATION WITH GPU

Rek and Němec [6] have presented a parallel implementation for the GPU architecture. In their implementation they still use the element based partitioning, but they had to pay attention to the synchronization between the block of threads on the GPU. They have created two kernels for the GPU. The first kernel calculates the internal and external forces while the second kernel calculates the equation of motion. The division of the kernels is due to the same reason as it was discussed in Section 3, where after the calculation of the residuals they are communicated between the processes. In the case of a GPU the main synchronization possibility between the threads is to quit the kernel, execute some code on the CPU and start the new kernel to calculate the velocities and coordinates with the already known residuals.

This paper presents a different approach. To explain the reason behind this different approach Listing 1 shows a code snippet from the dynamic relaxation method. In the code the loop iterates over all cable elements and their internal force is added to the residual vector. From the code it can be seen that at lines 28 and 29 there would be a race condition on a shared memory architecture system, since all elements connecting to node `n1` and `n2` would add its contribution to the residual vector and it is not possible to determine in what order the threads will do the adding. Theoretically this can be solved by introducing a critical section around this part of the code, however a critical section requires synchronization between the threads. Unfortunately on the GPU architecture synchronization is only possible between a block of threads or by quiting the GPU kernel and then restarting another kernel.

Listing 1: Calculation of residuals for every cable element

```
for(i = 0; i < nelems; i++)
{
  /* node numbers of the cable element */
  n1 = nodes[i][0]-1;
  n2 = nodes[i][1]-1;

  /* determine current length and length components */
  link_length = 0.0;
  for(k = 0; k < dimension; k++)
  {
    dl[k] = (coords[n2][k] - coords[n1][k]);
    link_length += dl[k] * dl[k];
  }
  link_length = sqrt(link_length);

  young = material[i]->YMod;     /* Young's modulus*/
  area  = material[i]->dParam1; /* area of cross section */
  deltal = link_length - init_length[i];
  /* determine internal force */
  force = deltal * young * area / init_length[i];

  for(k = 0; k < dimension; k++)
  {
    /* determine force components in directions X, Y and Z */
    dr = dl[k] * force / link_length;

    /* add component forces to both nodes of the element */
    residual[n1][k] += dr;
    residual[n2][k] -= dr;
  }
}
```

To avoid this synchronization problem, in the proposed approach of this paper a similar loop would iterate through all nodes. Listing 2 shows the code snippet of this node oriented loop, which should be compared to Listing 1. On the GPU architecture this loop would translate to the situation where one thread handles one iteration of the loop, specifically one node of the finite element mesh. In this case there is no race condition since at lines 35 and 36 the force contribution is added only to the current node (`j`) and no other thread is writing to the same position in the residual vector. This approach allows to have a single GPU kernel which calculates the residuals, velocities and coordinates together.

It is important, that in this case the internal forces of the elements are calculated as many times as the number of nodes of the element, which means recalculation of certain values on different threads. For example in the case of a cable element node 11 belongs to thread 157 and node 24 belongs to thread 231, then both threads (157 and 231) will read material data of the element and calculate the stiffness of the element and determine the internal force of the cable element. Thread 157 will update the residual vector for node 11 and thread 231 will update the residual vector for node 24 respectively. Although it seems very inefficient, but theoretically this double calculation should not be a problem since the GPU architecture is designed for throughput-oriented calculation, where calculation is cheaper then accessing the memory or synchronize.

Listing 2: Calculation of residuals for all nodes considering only cable elements

```
for(j = 0; j < nnode; j++)
{
  /* number of elements connected to node j */
  ne = n_node_elem[j];
  /* go thorugh all elements connected to node j */
  for(ie = 0; ie < ne; ie++)
  {
    /* element index connected to node j */
    i = node_elem[j][ie];

    /* node numbers of cable element i */
    n1 = nodes[i][0]-1;
    n2 = nodes[i][1]-1;

    /* determine current length and components */
    ...

    /* determine internal force */
    ...

    for(k = 0; k < dimension; k++)
    {
      /* determine force components in X, Y and Z direction */
      dr = dl[k] * force / link_length;

      /* add component forces only to node j */
      if(j == n1) residual[n1][k] += dr;
      if(j == n2) residual[n2][k] -= dr;
    }
  }
}
```

4.1 Summation of kinetic energy on GPU

Once the residual forces, the velocities and the new coordinates of all points have been calculated then the total kinetic energy must be determined. One approach would be to copy the mass and velocity of all nodes back to the memory of the CPU, however this communication would reduce the efficiency of the algorithm. Another approach would be to perform the summation of the kinetic energy on the GPU since all data is available in the memory of the GPU. A highly optimised implementation of the parallel reduction on GPUs is available [7], therefore this very efficient version of the reduction is used in this research.

4.2 General iteration of dynamic relaxation on GPU

Figure 2b shows the general iteration of the parallel dynamic relaxation method for the GPU architecture. In the figure the shaded boxes denote kernel functions executed on the GPU and all other operations are executed on the CPU. The `kernel step` function performs the residual, velocity and coordinate calculations at one node of the finite element mesh on the GPU. When the kernel exits, there is a barrier, therefore all threads will finish their calculation together. The `kernel reduction` function sums up the total kinetic energy on the GPU. At the end of this kernel a single floating point value must be copied back from the GPU to the CPU. This energy value will be used to determine whether the method has reached an energy peak or convergence.

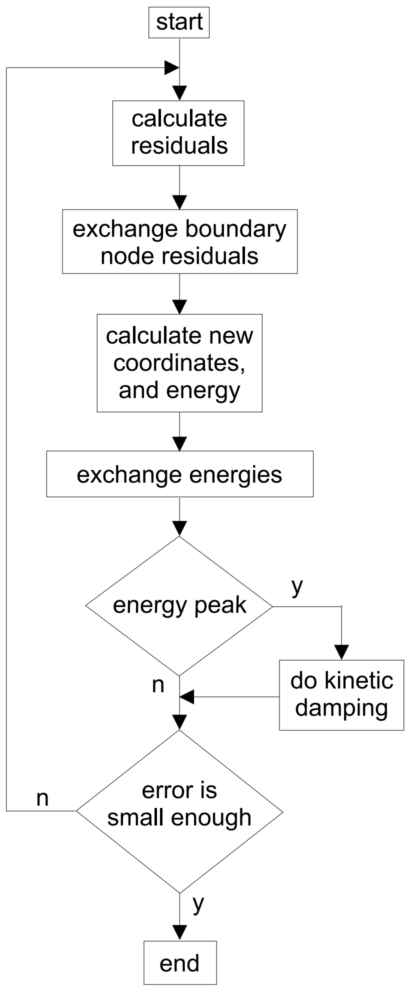
5 PERFORMANCE OF THE METHOD

The previously presented new implementations of the dynamic relaxation method have been tested on a PC (with Intel Core2 Quad CPU, 2.83GHz, 4GB RAM) with an NVIDIA GeForce GTX 780 graphics card (Compute Capability: 3.5). The operating system is a 64 bit Windows 10 with an NVIDIA driver version 353.62. All programs were compiled with double precision. Unfortunately this is a requirement, since the method does not converge with single precision.

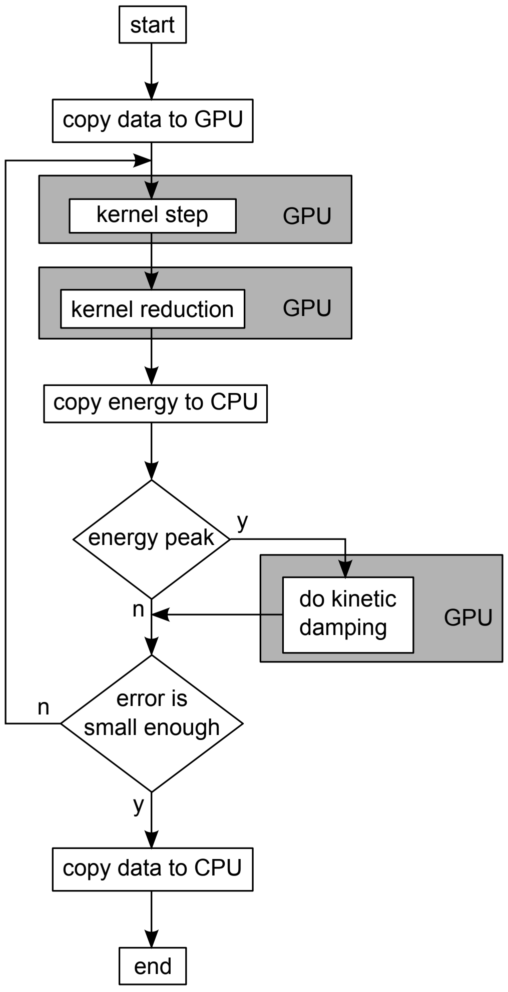
To measure the performance of the implementations a mesh of a cable net is considered in this paper. Figure 3 shows the geometry of this cable net. Further parameters are shown in Table 1, for example the number of vertical and horizontal lines and the total number of cable elements in the mesh. The table shows the sequential execution time of 10000 iterations that is executed on the PC with a single a processor core. The synchronous execution in the table corresponds to the iteration shown in Figure 2b. The table also shows the speedup calculated from the sequential and the synchronous parallel execution time.

6 CONCLUSION

This paper has presented a new parallel implementation of the dynamic relaxation method for the GPU architecture. The results are very promising, since even in the case



(a)



(b)

Figure 2: (a) Parallel implementation of the dynamic relaxation with the communication steps. (b) Parallel implementation of the dynamic relaxation with GPU.

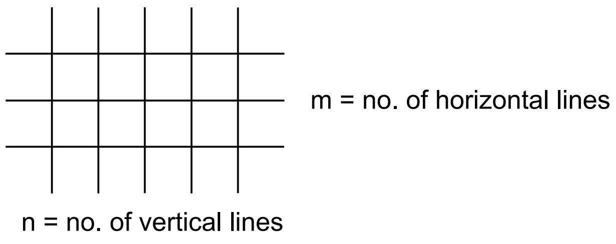


Figure 3: The geometry of the example cable grid.

Mesh	No of elements	Sequential execution time	Synchronous execution time	Speedup for synchronous execution
n x m		[sec]	[sec]	[sec]
100x100	19 404	13.65	2.98	4.58
200x200	78 804	62.73	5.38	11.66
300x300	178 204	167.77	9.27	18.10
400x400	317 604	313.02	14.47	21.63
500x500	497 004	491.27	21.54	22.81
600x600	716 404	708.33	30.20	23.45
700x700	975 804	962.69	40.89	23.54
800x800	1 275 204	1254.30	53.68	23.37
900x900	1 614 604	1583.02	68.58	23.08
1000x1000	1 994 004	1946.06	85.58	22.74

Table 1: Parameters of the example meshes and the sequential and parallel execution times on the GPU

of two million cable elements the solution can be obtained in approximately 85 seconds on a GPU, instead of running the program for half an hour sequentially.

In the current implementation the CPU is idle. This is very inefficient, therefore one approach for future improvement can be, that the calculation of nodes in one iteration is divided up between the CPU and the GPU. Although it is a promising approach, however in this case more data must be transmitted between the CPU and the GPU and this may reduce the efficiency. This will be a future direction of the research.

Another possibility for the optimization of the method, as usually suggested in the literature, would be the use of the shared memory of the GPU during the calculation, however it will make the code more complex.

ACKNOWLEDGEMENT

The support of the NVIDIA Corporation for the donation of graphics cards used for this research is gratefully acknowledged. The support of the SmartDiag Research project (PTE Excellence Center) is also acknowledged.

REFERENCES

[1] A.S. Day, "An introduction to dynamic relaxation", *The Engineer, January*, 1965.

[2] B.H.V. Topping, A.I. Khan, "Parallel computation schemes for dynamic relaxation", *Engineering Computations*, 11(6): 513–548, 1994.

[3] B.H.V. Topping, A.I. Khan, *Parallel Finite Element Computations*, Saxe-Coburg Publications, Edinburgh, 1996.

[4] B.H.V. Topping, P. Iványi, *Computer Aided Design of Cable-Membrane Structures*, Saxe-Coburg Publications, Stirling, 2007.

[5] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[6] V. Rek, I. Němec, "Parallel Computing Procedure for Dynamic Relaxation Method on GPU Using NVIDIA's CUDA", in *21st International Conference ENGINEERING MECHANICS 2015*, pages 254–255, May 2015.

[7] M. Harris, "Optimizing Parallel Reduction in CUDA", Technical report, NVIDIA Developer Technology, 2007.