Parallel FDTD Solver with Static and Dynamic Load Balancing

Gleb Balykov

Lomonosov Moscow State University, Moscow, 119991, Russia
balykov.gleb@yandex.ru

Abstract. Finite-difference time-domain method (FDTD) is widely used for modeling of computational electrodynamics by numerically solving Maxwell's equations and finding approximate solution at each time step. Overall computational time of FDTD solvers could become significant when large numerical grids are used. Parallel FDTD solvers usually help with reduction of overall computational time, however, the problem of load balancing arises on parallel computational systems. Load balancing of FDTD algorithm for homogeneous computational systems could be performed statically, before computations. In this article static and dynamic load balancing of FDTD algorithm for heterogeneous computational systems is described. Dynamic load balancing allows to redistribute grid points between computational nodes and effectively manage computational resources during process of computations for arbitrary computational system. Dynamic load balancing could be turned into static, if data required for balancing was gathered during previous computations. Measurements for presented algorithms are provided for IBM Blue Gene/P supercomputer and Tesla CMC server. Further directions for optimizations are also discussed.

Keywords: Computational Electrodynamics · FDTD · Parallel FDTD · MPI

1 Overview

The FDTD method, originated in 1966 [1], is widely used in electrodynamics solvers. Since then, sequential FDTD solvers evolved to high-performance parallel solvers, which incorporate different parallelization techniques.

Load balancing in parallel FDTD algorithms could significantly impact overall performance, as it allows to efficiently distribute load across all computational nodes of computational system based on their performance and current parameters of computation [2] [3]. For homogeneous computational systems load balancing could be performed before the start of computation, which is described in our previous work [4]. In general, for heterogeneous computational systems, characteristics of each computational node should be taken into account, because all computational nodes may have different (in general, arbitrary) performance and share time between each other.

Different load distributions across computational nodes of the computational system could be chosen as the most efficient based on these performance parameters. For example, it could be more efficient to assign different number of grid points to different computational nodes, or it could be more efficient to disable some computational nodes in order to exclude heavy share operations. One example of such computational systems could be the computational system with GPU only on a single computational node, performance of which is higher than the performance of other computational nodes.

This also applies to the current parameters of computations, as they could also lead to different load distributions. One example could be the case, when with specified number of grid points it is more efficient to perform computations sequentially on a single computational node rather than spread computations across all computational nodes.

One group of approaches for dynamic load balancing for heterogeneous computational systems is based on measurements of execution time of each computational node, after which migration of tasks from one node to another is performed [5] [6]. This approach should be updated with specifics of algorithm it is applied to in order to achieve the best balancing properties. Also, migration could be a heavy operation, especially when data is transferred between computational nodes with low bandwidth.

Another popular approach for dynamic load balancing is ”work stealing” [7] [8]. In the core of this and similar approaches lies queue of tasks, which are to be performed. Each computational node takes element of this queue, performs computations on it, and then takes the next element from queue if it’s not empty. This will balance computations across all computational nodes on granularity of the single element of queue, because faster node will finish its computations faster than slower one, i.e. faster node will take the next element of queue. Queues could exist for each computational node separately, then, faster computational node can ”steal” element from queue of the slower one.

Work stealing approach should be used cautiously, because choice of queue element could significantly impact overhead and balancing abilities of algorithm. Even for systems with shared memory incorrectly chosen granularity could lead to non-optimal balancing, for example, when there are less queue elements than overall number of computational nodes. For systems with distributed memory this problem is combined with the problem of data locality, because for many numerical algorithms data from neighboring grid points is required. This means that faster computational node may be unable to steal anything from the slower node without performance degradation.

In this article, parallel FDTD solver with static and dynamic load balancing for heterogeneous computational systems is introduced [9], which incorporates dynamic balancing of computations between computational nodes for different dimensions and different virtual topologies. Characteristics of each computational node are identified dynamically, during computations, and load balancing is performed based on them. This dynamic data could be saved to disk and then used for static balancing on this same computational system even for different

Суперкомпьютерные дни в России 2018 // Russian Supercomputing Days 2018 // RussianSCDays.org

computation parameters. Presented algorithm allows to preserve good data locality over all computational nodes, because it takes into account properties of FDTD algorithm. Besides, dynamic balancing overhead can be removed fully if only static part of balancing is performed.

2 Parallel Algorithm Description

Electrodynamics modeling could be performed in one-dimensional (1D), two-dimensional (2D) or three-dimensional (3D) modes. For all dimensions Cartesian computational grid is introduced: Ox axis for $1D$ mode, Ox and Oy axes for $2D$ mode, Ox,Oy and Oz axes for $3D$ mode. Yee grid [10] for field components is then set, and all points of Yee grid are spread between all computational nodes, so that each point of Yee grid is assigned to some computational node. For sequential solver all Yee grid points remain on the single computational node.

Let N be the number of computational nodes used in computations. Let's consider it being specified by user of the solver at start of computations. Let S be the total number of Yee grid points and let T be the total number of time steps, which are also specified at start of computations by user.

Yee grid points are assigned to different computational nodes in a natural way: Yee grid is divided in rectangular chunks, which are then assigned to different computational nodes. All N computational nodes are considered to have buffers, which store data from the neighboring computational nodes. This computational nodes' layout could be mapped directly on MPI virtual topology with single MPI process launched on every computational node.

Each time step each computational node performs computations for Yee grid points from the chunk assigned to it and then performs share operations with the neighboring computational nodes. Let's consider only buffers of size 1 by the axis for which each buffer is defined, then, no additional computations are performed for buffer points.

Overall computational time τ_{total} is sum of computational time and share time for each time step τ^t_{total}, and only the maximum time for each time step is taken into account. As described in [4], share operations are performed in all available directions sequentially, one after another (for example, for 2D case there are 8 directions). τ^t_{calc} is the computational time, τ^t_{share} is the sum of maximum share times for all directions on time step t.

$$\tau_{total} = \sum_{t=0}^{T-1} \tau^t_{total} = \sum_{t=0}^{T-1} (\tau^t_{calc} + \tau^t_{share}) \tag{1}$$

$$\tau_{total} = \sum_{t=0}^{T-1} (\max_{i=0}^{N-1} \tau^t_{calc_i} + \sum_{dir} \max_{i,j \in dir} \tau^t_{share_{i,j}}) \tag{2}$$

As also stated in [4], there are several possible virtual topologies for each dimension. Each topology has its own set of directions, and both computational time and share time should be minimized in order to minimize total execution time τ_{total} .

Let S_i^t be the number of Yee grid points assigned to i computational node at time step t, $i = 0, .., N-1$, $t \geq 0$. Number of grid points assigned to i computational node may change as computations proceed. Process of load balancing, or grid balancing, distributes computations between computational nodes based on their performance parameters.

$$S = \sum_{i=0}^{N-1} S_i^t, \forall t \geq 0 \tag{3}$$

Let $U_{i,j}^t$ be the number of grid points that are shared between i and j computational nodes at time step t, $i, j = 0, .., N-1$, $t \geq 0$. If computational nodes i and j do not performed any share operations between each other at time step t, $U_{i,j}^t = 0$. Besides, $S_i^t = 0$ for nodes which are disabled on time step t.

Let $state_i^t$ be the state of i computational node on time step t. In case computational node i is disabled on time step t, $state_i^t = 0$, otherwise $state_i^t = 1$. There should exist at least one computational node j for each time step t, for which $state_j^t = 1$.

Let $perf_i^t$ be the performance of i computational node right before the time step t, in other words, number of grid points, on which operations were performed, divided by elapsed time. In order to increase accuracy of balancing, let's calculate $perf_i^t$ as average values for all previous time steps. In this case, the longer computations are running, the more accurate values of performances are obtained.

Let T_{perf} be the number of time steps, after which performance values are updated. Then, right before time step $t = l * T_{perf}$, $l = 0, 1, .., T/T_{perf}$:

$$perf_i^{l*T_{perf}} = \frac{T_{perf} * \sum_{v=0}^{l-1} S_i^{v*T_{perf}}}{\sum_{v=0}^{l*T_{perf}-1} \tau_{calc_i}^v} \tag{4}$$

$$S_i^{v*T_{perf}} = S_i^{v*T_{perf}+k}, U_{i,j}^{v*T_{perf}} = U_{i,j}^{v*T_{perf}+k} \tag{5}$$

where $k = 1, .., T_{perf} - 1, v = 0, .., l-1$. $\tau_{calc_i}^v = 0$ if computational node i is disabled on time step v. If no previously computed data is loaded from file, $perf_i^0 = 0$, and computations are spread evenly between all computational nodes. Computational overhead of balancing could be minimized by tuning values of parameters T_{perf} based on the values of T and S.

2.1 1D case

Let $A > 0$ be the size of Yee grid by Ox axis, with a_i^t grid points assigned to i computational node on time step t: $S = A$, $S_i^t = a_i^t$, $i = 0, .., n-1$, $t = 0, .., T$. Let n be the number of computational nodes, between which Ox axis is spread. For one-dimensional case $N = n$.

Let $x^t_{L_i}$ and $x^t_{R_i}$ be the start and end coordinates of chunk assigned to i computational node on time step t, $x^t_{L_0} = 0$, $x^t_{R_i} = x^t_{L_{i+1}}$, $x^t_{R_{n-1}} = A$, $x^t_{R_i} - x^t_{L_i} = a^t_i$.

In one-dimensional case all computational nodes have same buffer sizes buf, which means that no matter how computations are spread between computational nodes, buffer sizes remain the same. Then, if computational nodes i and j do not perform share operations at time step t, $U^t_{i,j} = 0$, otherwise, $U^t_{i,j} = buf$.

Share time between i and j computational nodes could be estimated using Hockney [11] communication model:

$$\tau_h(i,j,x) = latency_{i,j} + \frac{x}{bandwidth_{i,j}} \tag{6}$$

where x - is the number of grid points, which are shared between i and j computational nodes, $latency_{i,j}$ is the time required to setup the sharing procedure between i and j computational nodes, $bandwidth_{i,j}$ is the speed of communication between i and j computational nodes.

Let $\tau^t_{s_{i,j}}(x)$ be the measured average share time between i and j computational nodes for x grid points for time steps $t - T_{perf}, .., t-1$. For each number of grid points x there could be different amount of performed share iterations $D^t_{i,j}(x)$, by which the average value is taken. For in-process measurements $D^t_{i,j}(U^{t-T_{perf}}_{i,j}) = T_{perf}$, for additional measurements $D^t_{i,j}(x)$ could vary based on the overhead of such measurements.

Then, for in-process measurements, i.e. for measurements, which were taken during computations:

$$\tau^t_{s_{i,j}}(U^{t-T_{perf}}_{i,j}) = \frac{\sum_{v=t-T_{perf}}^{t-1} \tau^v_{share_{i,j}}}{D^t_{i,j}(U^{t-T_{perf}}_{i,j})} \tag{7}$$

Latency could be obtained by performing additional measurements of share time $\tau^t_{s_{i,j}}(0)$ for empty messages or share time $\tau^t_{s_{i,j}}(x)$ for sizes $x \neq U^{t-T_{perf}}_{i,j}$. Another option is to choose $\tau^t_{s_{i,j}}(1)$ as latency, as $1 << bandwidth_{i,j}$.

Linear regression for all the measurements $\tau^t_{s_{i,j}}(x)$ is used to determine latency and bandwidth. Also, in order to increase accuracy, latency and bandwidth are calculated as average values for all time steps. Share time $\tau^t_{share_{i,j}}$ for the next time step t then could be estimated using Hockney model.

Spreading computations. In case $perf^0_i$, $\forall i$ are equal to 0, computations are spread evenly between all computational nodes. In case $perf^0_i$, $\forall i$ are somehow defined, grid points are spread proportionally to performances of computational nodes. Let α^t_i be the ideal number of grid points, which could be assigned to i computational node on t time step, based on performance of computational node, $\alpha^t_i \in R$, $\alpha^t_i \geq 0$. α^t_i could be non-integer, in this case integer value to assign to a^t_i has to be calculated.

Let $I^t_{L_i}$ and $I^t_{R_i}$ be the start and end coordinates of ideal chunk, which could be assigned to i computational node on time step t, $I^t_{L_0} = 0$, $I^t_{R_i} = I^t_{L_{i+1}}$,

$I^t_{R_{n-1}} = A$, $I^t_{L_i}, I^t_{R_i} \in R$. Ideal number of grid points α^t_i is calculated in the next manner:

$$\alpha^t_i = A * \frac{perf^t_i * state^t_i}{\sum_{v=0}^{n-1} perf^t_v * state^t_v}, i = 0, .., n-1, t \geq 0 \tag{8}$$

The goal is to choose a^t_i having the lowest deviations from ideal values α^t_i. For computational node $i = 0$ left border is always the same. As calculation time on i computational node is proportional to the number of grid points assigned to it, there are only two candidates for $x^t_{R_0}$: $[I^t_{R_0}]$ and $[I^t_{R_0}] + 1$, because they have the lowest deviation from the ideal value $I^t_{R_0}$.

Let $\delta^t_{L_i}$ be the deviation of $x^t_{L_i}$ from the ideal left border value $I^t_{L_i}$, same for $\delta^t_{R_i}$ and $I^t_{R_i}$: $x^t_{g_i} = I^t_{g_i} - \delta^t_{g_i}, g \in \{L, R\}, i = 0, .., n-1$.

Also let $\delta^t_i = I^t_{R_i} - [I^t_{R_i}]$. Algorithm consists of the following steps:

1. Update $\delta^t_{L_i}$: $\delta^t_{L_0} = 0$, for other computational nodes: $\delta^t_{L_i} = \delta^t_{R_{i-1}}$.

2. Update $\delta^t_{R_i}$ for computational nodes $i = 0, .., n-1$: $\delta^t_{R_{n-1}} = 0$

a) $\delta^t_{R_i} = \delta^t_i$, if $|\delta^t_{L_i} - \delta^t_i| \leq |\delta^t_{L_i} + 1 - \delta^t_i|$

b) $\delta^t_{R_i} = \delta^t_i - 1$, otherwise

3. Update $x^t_{L_i}$, $x^t_{R_i}$, a^t_i for computational nodes $i = 0, .., n-1$ according to the formulas above. i computational node is disabled if $a^t_i = 0$.

Both directions (from start to end and from end to start) should be checked and the minimum between them should be chosen. This algorithm allows us to choose integer values of a^t_i, which are the closest to the ideal values α^t_i, and these values lead to the lowest τ^t_{calc}.

Disabling computational nodes. In some cases it could be more efficient to disable some computational nodes at time step t in order to decrease overall computational time by decreasing τ^t_{share}. For 1D case there are two possible directions (positive and negative across Ox axis), and total share time is next:

$$\tau^t_{share} = 2 * \max_{i=0}^{N-2} \tau^t_{share_{i,i+1}} \tag{9}$$

Let's consider connections between computational nodes across Ox axis, starting from the connection with the highest share time between i and $i+1$ computational nodes. $perf^t_{all}$ is performance of all enabled computational nodes at time step t, $perf^t_R = perf^t_{all} - perf^t_L$ and

$$perf^t_L = \sum_{j=0}^{i} perf^t_j \tag{10}$$

There are next possibilities.

1. Disable all computational nodes to the left or to the right from the border between i and $i+1$ computational nodes, if computational time on this reduced set of computational nodes is lower. The next condition checks if total computational time on computational nodes $0, .., i$ is less than total computational time on $0, .., N-1$ (i.e. condition for disabling nodes $i+1, .., N-1$):

$$\tau^t_{calc_L} + \tau^t_{share_L} < \tau^t_{calc_{all}} + \tau^t_{share_{all}} - \varepsilon \tag{11}$$

As the border with the highest share time is considered, $\tau^t_{share_{all}} > \tau^t_{share_L}$. Similar condition could be written for disabling computational nodes $0, .., i$. Accuracy ε is the parameter, which helps to deal with inaccuracies of computations of performance parameters. In case Yee grid points are spread between all computational nodes proportionally to performance, computational and share time could be estimated in the next manner:

$$\tau^t_{calc_L} \approx \frac{S}{perf^t_L} \tag{12}$$

$$\tau^t_{share_L} = 2 * \max_{j=0}^{i-1} \tau^t_{share_{j,j+1}} \approx 2 * \max_{j=0}^{i-1} (latency^t_{j,j+1} + \frac{U^t_{j,j+1}}{bandwidth^t_{j,j+1}}) \tag{13}$$

2. Disable K computational nodes to the left or to the right from the border between i and $i+1$ computational nodes, in case computational time on these reduced sets of computational nodes is lower.

For this case additional measurements have to be performed for share operations, which have not been performed yet, otherwise, share time can't be estimated.

Let's check the case of disabling K computational nodes to the left from the border between i and $i+1$ computational nodes. There are $i+1$ cases for K from 1 to $i+1$, i.e. cases for disabling computational nodes from $i-K+1$ to i. From all values of K the one with the smallest computational time is chosen.

$$perf^t_1 = \sum_{j=0}^{i-K} perf^t_j + \sum_{j=i+1}^{N-1} perf^t_j \tag{14}$$

The condition for disabling computational nodes from $i-K+1$ to i is next:

$$\tau^t_{calc_1} + \tau^t_{share_1} < \tau^t_{calc_{all}} + \tau^t_{share_{all}} - \varepsilon \tag{15}$$

Computational and share time could be estimated as:

$$\tau^t_{calc_1} \approx \frac{S}{perf^t_1} \tag{16}$$

$$\tau^t_{share_1} = 2 * \max(\max_{j=0}^{i-K} \tau^t_{share_{j,j+1}}, \max_{j=i+1}^{N-1} \tau^t_{share_{j,j+1}}) \tag{17}$$

3. Disable all computational nodes, except the one w with the highest performance $perf^t_w$. This case could be checked once per rebalance. The disabling condition is similar to previous cases considering 0 share time.

After all cases are checked, the resulting balancing is chosen, so that the minimum computational time for all the cases is reached:

$$\tau^t_{total} = \min(\min_K \tau^t_{total_1}, \min_K \tau^t_{total_2}, \tau^t_{total_L}, \tau^t_{total_R}, \tau^t_{total_w}) \tag{18}$$

This process could be continued for the next highest share time between i and $i+1$ computational nodes, until the conditions from all cases are no longer satisfied or until all the pairs of neighbors are checked, or there is nothing else to disable.

Enabling computational nodes. As performance parameters of computational nodes are measured dynamically, inaccuracies arise. This leads to the case, when some computational nodes were disabled, performance parameters were updated, and now it is more efficient to enable them. Partially, ε should help with this. Other thing that should help to reduce inaccuracies is the accumulated average values of performance parameters, so the further computations continue, the more accurate values are obtained.

The conditions for enabling computational nodes are similar to the condition for disabling, except that they are reversed:

$$\tau^t_{total_L} > \tau^t_{total_{all}} + \varepsilon \tag{19}$$

Other conditions could be written in the same way. It would be inefficient to check all combinations of computational nodes for each joint set of disabled nodes. Solution is to enable computational nodes in the same sets, as they were disabled, which will significantly reduce the number of possible combinations.

So, each T_{perf} time steps conditions for disabling and enabling computational nodes should be checked.

2.2 2D and 3D cases

Same logic could be applied to higher dimensions in case only one axis is spread between computational nodes, i.e. for $2D-X$, $2D-Y$, $3D-X$, $3D-Y$, $3D-Z$ virtual topologies. All changes in formulas are related to the size of chunks assigned to computational nodes, as they are two- or three-dimensional for $2D$ and $3D$ modes correspondingly.

For $2D-XY$, $3D-XY$, $3D-YZ$, $3D-XZ$ and $3D-XYZ$ virtual topologies similar algorithms with minor changes could be applied. The most significant change is that nodes can't be disabled or enabled separately from the line or plane for $2D$ and $3D$ modes correspondingly.

Let's consider $2D$ case with $2D-XY$ virtual topology. Let $S = A * B$, where A and B are the sizes by Ox and Oy axes correspondingly, $N = n * m$, where n and m are the sizes of nodes' grid by Ox and Oy axes correspondingly. Both A and B are defined same to $1D$ mode.

$$S = \sum_{k=0}^{N-1} S^t_k = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} S^t_{(i,j)} = \sum_{i=0}^{n-1}\sum_{j=0}^{m-1} a^t_i * b^t_j, \forall t \geq 0 \tag{20}$$

In the same way to $1D$ case $U^t_{(i,j),(k,l)}$ and performance parameters $perf^t_{(i,j)}$, $latency^t_{(i,j),(k,l)}$, $bandwidth^t_{(i,j),(k,l)}$ are setup.

Computations could be spread for each axis independently by the same algorithm as discussed in the previous sections, except that $perf^t_i$ now is the performance of the whole line, for example:

$$perf^t_i = \sum_{j=0}^{m-1} perf^t_{(i,j)} \tag{21}$$

Another difference arises in the disabling and enabling conditions of computational nodes. Let the latencies for share operations between i and $i+1$ columns be the next:

$$latency^t_{(i,all),(i+1,all)} = \frac{\sum_{j=0}^{m-1} latency_{(i,j),(i+1,j)}}{m} \tag{22}$$

$$latency^t_{(i,all),(i+1,all+1)} = \frac{\sum_{j=0}^{m-2} latency_{(i,j),(i+1,j+1)}}{m-1} \tag{23}$$

$$latency^t_{(i,all),(i+1,all-1)} = \frac{\sum_{j=1}^{m-1} latency_{(i,j),(i+1,j-1)}}{m-1} \tag{24}$$

In the same way bandwidth is setup. Share time between i and $i+1$ columns could be estimated using Hockney model:

$$U^t_{(i,all),(i+1,all)} = \sum_{j=0}^{m-1} U_{(i,j),(i+1,j)} \tag{25}$$

$$\tau^t_{share_{(i,all),(i+1,all)}} \approx latency^t_{(i,all),(i+1,all)} + \frac{U^t_{(i,all),(i+1,all)}}{bandwidth^t_{(i,all),(i+1,all)}} \tag{26}$$

All connections across Ox and Oy axes are considered in sorted order, descending by the values of share time across that axis and across diagonals. For border between i and $i+1$ columns across Ox axis this value is next:

$$\tau^t_{share_{(i,all),(i+1,all)}} + \tau^t_{share_{(i,all),(i+1,all+1)}} + \tau^t_{share_{(i,all),(i+1,all-1)}} \tag{27}$$

Disabling conditions are similar for the ones from $1D$ case, except that whole lines are considered, and there are 8 possible directions for communications, and same for enabling conditions.

2.3 Saving profiling data

Gathered dynamic data could be saved to disk for further re-usage. Specifically, this includes performance values $perf^T_i$, $latency^T_{i,j}$, $bandwidth^T_{i,j}$. In order to maintain average values of these parameters, they should be saved in the special manner. At time step T all parameters could be described as: $perf_i^T = \frac{Q_{perf_i}}{P_{perf_i}}$, $latency_{i,j}^T = \frac{Q_{latency_{i,j}}}{P_{latency_{i,j}}}$, $bandwidth_{i,j}^T = \frac{Q_{bandwidth_{i,j}}}{P_{bandwidth_{i,j}}}$. Q_{perf_i} is the total number of grid points, on which computations were performed on i computational node, P_{perf_i} is the total time, which was spent on computations on i computational node, similar for latency and bandwidth.

The values of Q and P are then saved to disk. When file with Q and P is loaded, performance parameters right before time step t could be calculated in the next manner:

$$perf_i^t = \frac{Q_{perf_i} + T_{perf} * \sum_{v=0}^{l-1} S_i^{v*T_{perf}}}{P_{perf_i} + \sum_{v=0}^{t-1} \tau_{calc_i}^v} \tag{28}$$

$$latency_{i,j}^t = \frac{Q_{latency_{i,j}} + \sum_{v=1}^{l} lat_{i,j}^{v*T_{perf}}}{P_{latency_{i,j}} + l} \tag{29}$$

where $l = t/T_{perf}$. Bandwidth is calculated similar to latency. Balancing could now be performed before the start of computations, as $perf_i^0$, $latency_i^0$, $bandwidth_i^0$ are now defined. If balancing is performed only before the start of computations, balancing overhead is fully removed, and balancing is static. Also, this approach allows to improve accuracy of performance parameters by storing just 6 values on disk.

2.4 Further work

Performance parameters, measurements of which were discussed in previous sections, describe the performance of the computational node. However, this implied that all computational nodes perform the same amount of measurements for each grid point. This could be incorrect in case some electromagnetic sources are setup, for example point source or plane wave source, or some additional measurements have to be performed only for part of grid points, for example near-to-far field computations. In such cases additional computations have to be considered in load balancing algorithm, which will be discussed in further work.

3 Measurements

All measurements were performed on IBM Blue Gene/P supercomputer and Tesla CMC server. IBM Blue Gene/P is a massively parallel computational system. It contains 8192 calculation cores (2048 calculation nodes, 4 core each) with peak performance at 27.9 tflops. Single calculation core is a PowerPC 450 with frequency at 850 MHz having 4GB of RAM. Communicational network is a three-dimensional torus and unites all the nodes: single node has 6 bidirectional connections with 6 neighbors. Tesla CMC [12] is a computational system of Moscow State University with two Intel Xeon E5620 CPUs and set of Nvidia GPUs, including Nvidia Tesla K20c with 5 Gb memory.

Basic FDTD computation was chosen as a benchmark (no PML, no TF/SF, point wave source for each computational node). In each computation virtual topology was mapped on computational nodes of Blue Gene/P in such a way that virtual topology matches physical topology, so, computational nodes, which are neighbors in virtual topology, will be neighbors in physical topology too, and no additional share expenses arise.

In order to demonstrate dynamic balancing, computational nodes with different performances were emulated on IBM Blue Gene/P nodes for $3D$ Yee grid of size $S = 100 * 100 * 100$. Computations were performed on 8 nodes, 4 of which were emulated to be slower by specific delay at each time step. Delay varies from large delay of 2 seconds, to small delay of 0.02 seconds, for which performance of slower computational node was 80% of performance of normal computational node for each time step.

The measurements in table 1 show, that load balancing could significantly improve total execution time, especially for long running tasks, even when computational nodes' delay is fairly small (i.e. computational nodes have nearly similar performances).

On Tesla CMC server computations were performed on both GPU and CPUs. For cases, when computation data could be fully located in GPU memory, usually, it is more efficient to perform all computations just on GPU, because this will remove overhead of data copying to/from GPU.

However, this is not the case for most FDTD modeling, especially for three-dimensional cases, as large numerical grids are required for good accuracy. In such cases there is no way to remove data copying to/from GPU at each time step, as data can not be fully located in GPU memory. Additional share time with GPU could be taken into account in dynamic load balancing algorithm described in previous sections.

Measurements on Tesla CMC were performed on 2 computational nodes, one with Tesla K20c, for grid of size 768*512*512, which can't be fully located in GPU memory. Balanced computation took on 15% less execution time than computations on a single node with GPU, and on 50% less execution time than on a equal spread between two computational nodes.

Table 1. Measurements for 3D mode for 8 computational nodes for Yee grid with size $S = 100 * 100 * 100$.

Nodes delay, seconds	Time steps	Execution time decrease with rebalance
2	100	61.3%
2	1000	above 90%
0.2	1000	14.4%
0.2	10000	28.2%
0.02	10000	3.6%
0.02	20000	5.0%

4 Conclusion

Developed FDTD solver provides features for dynamic load distribution between computational nodes. Measurements prove that described dynamic load balancing algorithm could significantly improve overall computational time on heterogeneous computational systems. This approach could be useful for computational systems with different CPUs on different computational nodes. Besides, this approach could also be used on computational systems with different GPUs on some computational nodes, where performance of GPU and additional communication time with GPU are taken into account. In further work dynamic load balancing would be improved for cases where different amount of computations is performed for different Yee grid points.

References

1. Yee, K.S.: Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. IEEE Transactions on Antennas and Propagation, vol. 14, No.3, pp. 303–307 (1966)
2. Franek, O.: A simple method for static load balancing of parallel FDTD codes. Proceedings of the International Conference on Electromagnetics in Advanced Applications (ICEAA 2016), pp. 587–590 (2016)
3. Shams, R., Sadeghi, P.: On optimization of finite-difference time-domain (FDTD) computation on heterogeneous and GPU clusters. Journal of Parallel and Distributed Computing, vol. 71, no. 4, pp. 584–593 (2011)
4. Balykov, G.: Parallel FDTD Solver with Optimal Topology and Dynamic Balancing. In: Voevodin, V., Sobolev, S. (eds) Supercomputing. RuSCDays 2017. Communications in Computer and Information Science, vol. 793, pp. 337–348. Springer, Cham (2017)
5. Sharma, R., Priyesh, K.: Dynamic Load Balancing Algorithm for Heterogeneous Multi-core Processors Cluster. In: Communication Systems and Network Technologies (CSNT), pp. 288–292. IEEE (2014)
6. De Grande, R., Azzedine, B.: Dynamic balancing of communication and computation load for HLA-based simulations on large-scale distributed systems. In: Journal of Parallel and Distributed Computing, vol. 71, no. 1, pp. 40–52 (2011)
7. Ravichandran, K., Lee, S., Pande, S.: Work Stealing for Multi-core HPC Clusters. In: Jeannot E., Namyst R., Roman J. (eds) Euro-Par 2011 Parallel Processing. Euro-Par 2011. Lecture Notes in Computer Science, vol 6852. Springer, Berlin, Heidelberg (2011)
8. Cederman, D., Tsigas, P.: Dynamic load balancing using work-stealing. In: GPU Computing Gems, Elsevier (2012)
9. Parallel FDTD solver, `https://github.com/zer011b/fdtd3d`
10. Taflove, A., Hagness S. C.: Computational Electrodynamics: The Finite-difference Timedomain Method. Artech House, 3rd ed. (2000)
11. Hockney, R.: The communication challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Computing, vol. 20, No.3, pp. 389–398 (1994)
12. Tesla CMC Server of Moscow State University, `http://hpc.cmc.msu.ru/tesla`