
Mathematical Modelling and Analysis
Volume 25, Issue 4, 584–607, 2020
https://doi.org/10.3846/mma.2020.12139

http://mma.vgtu.lt
ISSN: 1392-6292
eISSN: 1648-3510

A Three-Level Parallelisation Scheme and Application to the Nelder-Mead algorithm

Rima Kriauzienė[a], Andrej Bugajev[a] and Raimondas Čiegis[a]

[a] *Vilnius Gediminas Technical University*
Sauletekio al. 11, LT-10223 Vilnius, Lithuania

E-mail(*corresp.*): andrej.bugajev@vgtu.lt

E-mail: rima.kriauziene@vgtu.lt

E-mail: raimondas.ciegis@vgtu.lt

Received February 20, 2020; revised July 30, 2020; accepted July 31, 2020

Abstract. We consider a three-level parallelisation scheme. The second and third levels define a classical two-level parallelisation scheme and some load balancing algorithm is used to distribute tasks among processes. It is well-known that for many applications the efficiency of parallel algorithms of these two levels starts to drop down after some critical parallelisation degree is reached. This weakness of the two-level template is addressed by introduction of one additional parallelisation level. As an alternative to the basic solver some new or modified algorithms are considered on this level. The idea of the proposed methodology is to increase the parallelisation degree by using possibly less efficient algorithms in comparison with the basic solver. As an example we investigate two modified Nelder-Mead methods. For the selected application, a Schrödinger equation is solved numerically on the second level, and on the third level the parallel Wang's algorithm is used to solve systems of linear equations with tridiagonal matrices. A greedy workload balancing heuristic is proposed, which is oriented to the case of a large number of available processors. The complexity estimates of the computational tasks are model-based, i.e. they use empirical computational data.

Keywords: multi-level parallelisation, load balancing and task assignment, parallel optimisation, Nelder-Mead algorithm, Wang's algorithm, model-based parallelisation, finite difference methods, Schrödinger equation.

AMS Subject Classification: 65N06; 65N12; 35Q56.

1 Introduction

Current trends in supercomputing show that in order to accumulate high computing power, computers with more, but not faster, processors are used. This trend induces changes in the development of parallel algorithms. The important challenge is to develop parallelisation techniques which enable exploitation of substantially more computational resources than the standard existing methods.

This paper deals with problems that can be split into a collection of independent subproblems and this splitting step is repeated iteratively. The solutions of subproblems define the solution of the main problem. Thus, an additional parallelization level increases the potential parallelisation degree of a constructed parallel algorithm.

Any multi-level parallelisation can be considered as a way to generate a pool of tasks. After the pool of tasks is obtained, it is not important how many parallelisation levels were used. However, often such final simplification of the template leads to a loss of an important information and as a consequence to degraded efficiency of the parallel algorithm. Especially this is true if different levels of the scheme are characterised by different properties of an algorithm that should be properly addressed.

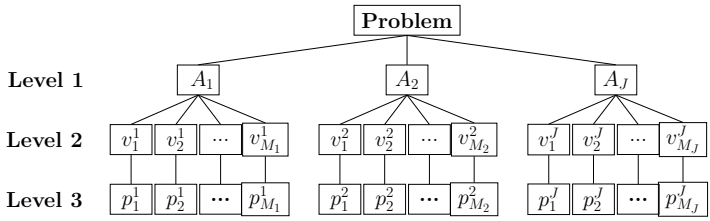

Figure 1. Three level parallelisation scheme.

In this paper, we consider a special case of a three level parallelisation. The template of this approach is given in Figure 1:

- At the first level of parallelisation we assume that there exist a few parallel alternatives A_j (see Figure 1) to the original modelling algorithm. The first level of parallelisation becomes a part of a new parallel algorithm and the degree of the first level parallelism can be selected dynamically during the computations – at this level the best algorithm is selected. In this paper as an example we consider two new parallel modifications of the Nelder-Mead method [25].

- On the second level, a set of computational tasks $V^j = \{v_1^j, v_2^j, \ldots, v_{M_j}^j\}$ (see Figure 1) with different computational complexities is defined. For simplicity of presentation, we assume that all these tasks can be solved in parallel. As an example we investigate the case when computation of one value of the objective function requires to solve numerically M partial differential equations. The computational complexities of tasks

are non-equal because different sizes of discretisation steps must be used for different equations in order to achieve the same accuracy for each equation.

- The third level defines parallel algorithms to solve tasks from the second level. As an example we use Wang's algorithm to solve systems of linear equations with tridiagonal matrices [39].

In this paper we propose a new three-level methodology for parallelisation some classes of algorithms. As a representative example, we use it to solve one applied optimisation problem. The superiority of the three level parallelisation scheme is shown, comparing it with two level parallelisation scheme.

The necessity of the additional first level comes from the assumption of having more computational resources than can be utilised by the two-level parallelisation approach. It is a consequence of the efficiency saturation for parallel algorithms when the size of the problem is fixed and the number of processes is increased.

We have defined a set of optimization methods (or a modification of the basic solver) which give an additional degree of parallelisation and enable the possibility to use efficiently more processors. At the first level of the template the optimal algorithm is selected. This part requires to find a compromise between the increased parallelisation degree and the decreased convergence rate of the modified parallel optimization algorithm.

This paper makes the following contributions:

1. We propose to extend the typical two level parallelisation, which is usually accompanied by some load balancing technique, by adding one additional level. Also, we investigate the possibility to limit the number of used processors to sustain the parallelisation efficiency at the selected level. This approach let us to avoid the inefficient calculations and supports the green computing technology.

   As an example two different families of parallel Nelder-Mead methods were investigated: the family of the generalised parallel Nelder-Mead method [22] and the parallel versions of the classical Nelder-Mead method. In order to perform the load balancing on the second and third levels of the proposed template, we use the complexity model of tasks which is based on the computational data (also known as model-based), as it is done in recent state-of-the-art works [20, 21, 30]. We demonstrate a big potential of this new technique.

2. A parallel version of the Nelder-Mead method is proposed, which does not change the convergence properties of the sequential optimisation algorithm. We note, that there were some attempts to parallelise this optimisation method before [19, 22]. However, in these papers the convergence properties are changed and these changes are not studied comprehensively enough. Moreover, it is questionable whether these parallel algorithms are applicable in the case of small-dimension problems.

   Our parallel algorithm leads to an increasement of the parallelisation degree up to factor of three. However, the introduced changes do not affect

the convergence of the sequential optimisation algorithm. The experimental comparison of this new parallelisation algorithm with the state-of-the-art technique [22] is provided. The obtained experimental results show that in the case of the Rosenbrock function the convergence properties of the parallel algorithm [22] are much worse than of the classical sequential Nelder-Mead algorithm.

The rest of this paper is organised as follows. In Section 3, the workload balancing problem is formulated, the selection of the optimal algorithm is provided and a strategy for workload distribution is presented along with the efficient workload distribution algorithm. In Section 4, the detailed description of three parallelisation levels are given for the studied case.

As a real world application, we consider the approximation of absorbing boundary conditions of Schrödinger equation. The modified Nelder-Mead method is used to solve local optimisation problems on the first level, on the second level a set of partial differential equations are solved numerically, and on the third level Wang's algorithm is used to solve systems of linear equations in parallel. In Section 5, the results of computational experiments are provided and the efficiency of the proposed three-level parallelisation template is analysed. In Section 6, the comparison of different Nelder-Mead parallelisation methods is presented. The final conclusions are done in Section 7.

2 The related works

We investigate a three-level parallelisation algorithm for optimization problems and different parallelisation levels give different possibilities but also challenges. The parallelisation speed-up on the first level in many cases is not linear, thus it can reduce the efficiency of the whole parallel algorithm. But this level enables to use much more processes and finally to solve the given problem faster than using two-level parallelisation. In this paper, as an example on the first level we investigate two modified Nelder-Mead methods.

In some cases the Nelder-Mead method is not the best method to solve local optimization problems. In paper [24], it was shown that the method can converge to a non-stationary point. However, for the most practical problems it gives good results with the reasonable amount of computations. That is why we use NM method as an example to test the proposed methodology. For all studied cases it gave sufficiently good solutions. It is important to note, that our methodology can be applied to other methods such as the Spendley Hext and Himsworth simplex [35], Hooke-Jeeves algorithm [17], etc.

Generally, on the first level different parallel algorithms can be used, however, the proposed approach is oriented to cases when the increased degree of parallelisation gives the speed-up at the cost of efficiency. We note that such a situation is typical for many parallel algorithms due to Amdahl's law.

As one more example we mention new algorithms developed to solve the global optimisation problems. The modification of the well-known DIRECT method [9] was presented in [36], it is called DIRECT-GL. The new modification is based on the idea at each iteration to analyse more potential optimal rectangles. This approach increases the global sensitivity of the method

but in many cases this property is achieved at the cost of additional computations. The potential parallelisation degree of DIRECT-GL algorithm can increase up to 2-3 times. But the results of computational experiments in [36] show that for many benchmark problems (in [36] these cases are numbered 1,2,5,6,20,21,22,24,35,37,38,47,48,49,52) the DIRECT-GL algorithm increases the computational costs to achieve the same accuracy of approximations as DIRECT algorithm. Thus, the classical DIRECT algorithm and its modification DIRECT-GL fit well into the proposed three-level parallelisation template. Then the degree of parallelisation should be increased only if this increasement compensates the reduced efficiency of the modified algorithm. Thus we state, that in order to apply the proposed three level parallelisation scheme, first the computations should be parallelised by using a classical a two-level parallelisation approach. Then alternative cases of parallel algorithms with additional degrees of parallelisation should be identified and the optimal algorithm should be selected.

The second and the third levels define a well-investigated two-level parallelisation template. We note that load balancing techniques for two-level parallelisation are widely used in applications, see, e.g., [5], [14].

The scheduling problem can be formulated representing a parallel algorithm by a directed acyclic graph (DAG). The vertices define computational tasks, the edges define connections/order among tasks. Then a set of partially ordered computational tasks is scheduled onto a multiprocessors system to minimise the computational time (or to optimise some other performance criteria). It is well known that the scheduling problem is NP complete. Many interesting heuristics are proposed to solve it, we mention greedy algorithms [6], genetic algorithms [33], [34], simulated annealing and tabu search algorithms [18], [12], [13]. Such algorithms include a possibility of dynamic scheduling and allow for tasks to arrive continuously and they can consider variable in time computational resources.

A scheduling task can be very challenging due to specificity of a given application problem and the necessity to parallelise it on modern parallel architectures. As an example we mention the particle simulation which is solved by appropriate domain decomposition techniques [11]. Another example is the dynamic load balancing on heterogeneous clusters for parallel ant colony optimisation [23]. In the recent work [7] it is concentrated on the problem of high-dimensionality of the data while solving subspace clustering problem.

In this article we focus on the scheduling problem, when all tasks in the set are independent and can be solved in parallel. It is well known that the given optimisation problem can be redefined as a problem to equalise the computational times of all processes. The simplest load balancing algorithm is based on the assumption that the computation time is proportional to sizes of subtasks. Then the domain decomposition algorithm is applied to guarantee that the sizes of subtasks scheduled for each group of processors are equal [5].

The quasi-optimal distributions of tasks can be obtained using the greedy strategy to distribute the work on demand, i.e. to apply dynamic load balancing techniques such as work-stealing [16], self-organising process rescheduling [31].

However, the efficiency of two-level approach is limited due to a typical

saturation of the speed up of parallel algorithms for increased numbers of processors and fixed sizes of tasks. Exactly this situation has motivated us to introduce an additional level of parallelisation template. In most cases the usage of it leads to a less efficient algorithms than the initial state-of-the-art algorithm. But the additional degree of parallelism on the second level gives a large overall speed-up, if the number of available resources is large.

Recent developments of new architectures of parallel processors make even more challenging the task to build accurate theoretical performance models. The empirical data shows that for some advanced algorithms the efficiency of parallel computations can depend non-monotonically on the size of a task. Thus the model-based load balancing method starts to become the main tool in developing efficient and accurate task scheduling algorithms. In our work we build the model for prediction of computation time empirically by solving the specialised benchmarks for a wide range of problem sizes and numbers of processors. In fact this analysis resemblance the classical experimental strong scalability analysis of a given parallel algorithm. We note, that these measurements are always done for all processes working simultaneously in order to reflect their actual performance during the execution of real applications (see, also [20, 21]).

Here we mention two interesting papers, where the model-based task scheduling algorithms are considered. In [21], it is concentrated on multicore coprocessors Xeon Phi, where the empirical computation time curves are used to find optimal parameters for a workload distribution. The obtained model predicts non-monotonic dependence of computation speed on the sizes of problems. The authors call their approach "load imbalancing", however, it can be considered as an advanced balancing which adapts the scheduling algorithm to the specificity of Xeon Phi processors. Obviously in this case the assumption that computation time is proportional to the task size is not valid. In a similar research [20], computations were performed on non-uniform memory access (NUMA) parallel platform with various shared on-chip resources such as Last Level Cache. Again the model-based approach enables to take into account the specific properties of the algorithm and processors. The matrix multiplication and Fast Fourier Transform are used as benchmark problems. It is interesting to note that, according to the presented results, the globally optimal solutions may not load-balance the sizes of sub-tasks. The authors pay a special attention to the energy efficiency of calculations. We note, that there are some papers that are specifically dedicated to load balancing of energy efficiency [27]. In our work we formulate some restrictions that are connected to energy efficiency as well – we do not use additional available computational resources if the parallelisation efficiency drops below some specific level. The other work [30] is dedicated to model-based optimisation on hybrid heterogeneous systems composed of CPUs and accelerators. In that research authors investigate the problem of communications costs due to uneven workload distribution between accelerators and CPUs. They propose to generalise the τ-Lop [29] model for heterogeneous computations.

In this work we are also interested to address some green computing (GC) challenges. In a broader sense GC is the practices and procedures of designing,

manufacturing, using of computing resources in an environment friendly way while maintaining overall computing performance and finally disposing in a way that reduces their environmental impact [32]. The research in green computing is done in many areas [26]: Energy Consumption; E-Waste Recycling; Data Center Consolidation and Optimization; Virtualization; IT Products and Ecolabeling. One of approaches for optimisation of energy consumption on the software level is the autotuning software, which is able to optimise its own execution parameters with respect to a specific objective function (usually, it is execution time) [4]. Well known examples of autotuning software are: FFTW [10] (fast Fourier transformations); ATLAS [41], PHiPAC [2] (dense matrix computations); OSKI [38], SPARSITY [15] (sparse matrix computations).

Usually, the goal for any autotuning software is to achieve the same result with the same resources, however, reducing the computation time – in terms of parallelisation it means to increase the parallelisation efficiency. Another way to decrease the power consumption is to increase the efficiency by avoiding inefficient calculations; this may slightly increase the execution time, however will give a reasonable increase of parallel efficiency, which leads to the energy savings. We propose to control the efficiency of the parallel algorithm on the load balancing stage of the parallelisation template. In many cases this strategy reduces the amount of computational resources used in computations. This analysis is done a priori, meaning that the user knows how many cores should be used for solving a specific parallel task even before starting real computations.

3 Workload balancing problem

In this section, we formulate the workload balancing problem for the two level parallelisation. Also we present a greedy scheduling algorithm to distribute the processes among tasks. Next, we introduce the additional level – the first and second levels of the two level parallelisation technique become the second and the third levels, accordingly and the first level is a new parallelisation level. On the first level the selection of the optimal algorithm is performed.

First, we will present two level parallelisation template. Assume that we solve a given problem by using the basic method A. The solution process consists of K blocks of tasks (a simple DAG)

$$A = \{V_1 \prec V_2 \prec \ldots \prec V_K\}, \tag{3.1}$$

and all blocks must be solved sequentially one after another. Each block consists of M tasks

$$V_k = \{v_1(X_k), v_2(X_k), \ldots, v_M(X_k)\}, \quad k = 1, \ldots, K,$$

where X_k defines a set of parameters for the V_k block and all M tasks can be solved independently. V_k defines the first level of two level parallelisation scheme. Each task v_m can be solved by some parallel algorithm – this is the second level of the scheme.

The complexities of tasks v_m are different, however, they are known in advance and do not depend on k. For each task v_m the prediction of computation

time $t_m(p)$, $p \le P$, $m = 1, \ldots, M$ is given – it is based on the modelling results, P is the number of processors in a parallel system. We assume that up to P_m processes the computation time monotonically decreases:

$$t_m(p_2) < t_m(p_1), \quad \text{for } p_1 < p_2 \le P_m.$$

For P_m the predicted computation time function $t_m(p)$ reaches the minimum value:

$$t_m(p) \ge t_m(P_m), \qquad p > P_m. \tag{3.2}$$

Such a model of computation time $t_m(p)$ is typical for algorithms with limited scalability such as Wang's algorithm. In Figure 2 we present speed-ups of this algorithm for different sizes of linear systems. It is important to mention that the provided results include some additional costs for computation of the objective function along with Wang's algorithm computational costs. These additional calculations slightly increase the overall parallelisation scalability, thus the provided figure represents the optimistic scenario for general Wang's algorithm and the realistic scenario for actual computations, that were done in presented computational experiments. The detailed specification of processors is presented in Section 4.

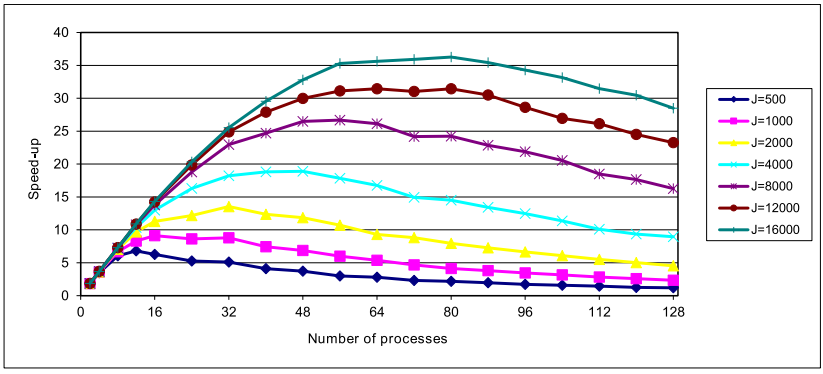

Figure 2. The speed-ups of Wang's parallel algorithm for different number of processes p and sizes J of systems. The detailed specification of processors is presented in Section 4.

In our specific case this data was derived from a simple benchmark implementing Wang's algorithm. This benchmark performs computations using different numbers note, that nodes were artificially loaded with calculations to imitate the real situation. For example, with the number of processes $p = 4$ there were 32 tasks that were solved by 128 processes at the same time. Thus this benchmark must be run once, using all processes available. of processes and different problem complexity parameters J. It is important to

From Figure 2 it follows that the computation time monotonically decrease till some critical number of processes and therefore the efficient usage of processes is limited to this number of processes. Even for large size systems, when the number of equations is $J = 16000$, the maximum number of processes P_m does not exceed 80. This analysis justifies our motivation to use the multi-level

approach in order to solve the given applied problem efficiently and to use as many available processors as possible.

In the two-level parallelisation scheme for each block of tasks V_k we select the number of processes such that the overall solution time is minimised:

$$\arg \min_{(p_1,\ldots,p_M)\in Q} \max_{1\le m\le M} t_m(p_m),$$

where a set of feasible processors distributions Q is defined as

$$Q = \{(p_1,\ldots,p_M) : p_1 + \ldots + p_M \le P\}.$$

Remark 1. In the case when we solve only few large size tasks and the remaining tasks are much smaller and the number of processes P is not very big, the optimal scheduling is obtained when a few smaller tasks are combined into one group $\widetilde{v}_m$. Then sub-task $\widetilde{v}_m$ consists of tasks $v_{l_1},\ldots,v_{l_n}$. The computation time for this combined task is predicted by the model:

$$\widetilde{t}_m(p_m) = \sum_{i=1}^{n} t_{l_i}(\widetilde{p}_{l_i}), \quad \widetilde{p}_{l_i} = \min\left(p_m, P_{l_i}\right).$$

In this work we are interested to solve the scheduling problem, when the number of processes is large, so the aggregation step is not used.

Next, we propose a simple greedy partitioning algorithm, which is described in Algorithm 1.

Algorithm 1. The algorithm for distribution of P processes between M tasks

```
1: Set p[m] = 1, for m = 1,...,M
2: P = P − M
3: Compute t_m(p[m]), for m = 1,...,M
4: stop = 0
5: while P > 0 & stop == 0 do
6:     find j such that t_j(p[j]) = max_{1≤m≤M} t_m(p[m])
7:     if p[j] == P_j then
8:         stop = 1
9:     else
10:        p[j] = p[j] + 1
11:        P = P − 1
12:    end if
13: end while
```

It aims to find a near-optimal distribution of M tasks of different sizes between homogeneous P processes by using the model-based complexity model $t_m(p)$ (similar ideas are also used in [20]). We assume that $P \ge M$. The interesting feature of the presented algorithm is that for a given number of processes P the number of active processes can be taken less than P to minimise the overall execution time of the parallel algorithm.

The algorithm starts from the initial distribution when one process is assigned for each task and the predictions of parallel execution times are calculated using the selected performance model. Then, the greedy iterative procedure is applied to distribute the remaining processes. At each iteration, one additional process is assigned to the task which has the largest predicted computation time. Then its parallel execution time is updated. Iterations are repeated until all processes are distributed or the number of processes for some task reaches the limit P_m.

Note, that before $t_m(p)$ has reached the minimum, value starts to decrease slowly, thus the parallelisation efficiency drops. Therefore, it may be wise to restrict the number of processes by taking into account the efficieny value.

We define the maximum number of processes $\widetilde{P}_k$ for which the efficiency condition is still satisfied

$$E_p(V_k) \geq E_{min}, \quad \text{for } p \leq \widetilde{P}_m, \tag{3.3}$$

where $E_p(V_k) = S_p(V_k)/p$ ir $S_p(V_k) = t_k(1)/t_k(p)$, $E_{min} \in [0,1]$ is a given efficiency lower bound. Estimate (3.3) is used to modify the limit of the maximum number of processes (3.2) that can be used to solve the j-th task

$$P_m = \min(P_m, \widetilde{P}_m).$$

Therefore, in the presented technique P_m includes two restrictions:

- The number of processes cannot exceed the number after which the speed-up drops down (see Figure 2).
- The number of processes is limited by efficiency requirement (3.3), which states: the number of processes per block of tasks V_k is not allowed to be increased if the efficiency of the parallel algorithm on the third level reaches the critical value E_{min}.

In fact the second level of the two-level scheme can be used alone, however, it is limited due to Amdahl's law [1], i.e. the efficiency begin to drop as the number of processes increases for a fixed size of problem. Two-level approach let us to solve this issue up to some point.

Exactly this situation has motivated us to introduce an additional level of parallelisation template.

In the new three-level parallelisation scheme, the second and third levels represent the two-level scheme described above. Additionally, we add a new first level of the template. We assume that there exist parallel alternative algorithms A_j:

$$A_j = \{V_1^j \prec V_2^j \prec \ldots \prec V_{K_j}^j\}, \quad j = 1, \ldots, J.$$

Each block V_k^j consists of M_j independent tasks

$$V_k^j = \{v_1^j(X_k), v_2^j(X_k), \ldots, v_{M_j}^j(X_k)\}, \quad k = 1, \ldots, K.$$

The numbers of blocks of tasks K_j, the numbers of tasks per block M_j, the sizes of tasks $|v_m^j|$ may be different for different j.

Next, we select the optimal algorithm according to the number of resources available. We denote

$$T_P(A_j) = T_P(V^j)K_j$$

the total solution time for algorithm A_j. The block of tasks V^j is solved by using the heuristic proposed above. Then the optimal algorithm is defined as

$$\arg \min_{1 \le j \le J} T_P(A_j).$$

The usage of $j > 1$ may lead to a less efficient algorithm than the initial basic algorithm. But the additional degree of parallelism gives a large overall speed-up.

4 Application of the three-level parallelisation scheme

First, we briefly present the problem which is used to test our methodology. We solve an initial-boundary value Schrödinger problem formulated in a finite space domain [3]:

$$\begin{cases} i\dfrac{\partial u}{\partial t} + \dfrac{\partial^2 u}{\partial x^2} = 0, \quad x \in (A,B), \;\; t \in (0,T], \\ u(x,0) = u_0(x), \quad x \in [A,B], \\ L_l u(A) = 0, \quad L_r u(B) = 0, \quad t \in (0,T], \end{cases} \tag{4.1}$$

where operators L_l, L_r define the nonlocal/transparent boundary conditions.

Let ω_h and ω_τ be discrete uniform grids with space and time steps h, τ:

$$\begin{aligned} \omega_h &= \left\{x_j : \; x_0 = A,\, x_J = B,\, x_k = x_{k-1} + h,\, k = 1, \ldots, J\right\}, \\ \omega_\tau &= \{t^n : \;\; t^n = n\tau, \quad n = 0, \ldots, N,\, N\tau = T\}. \end{aligned} \tag{4.2}$$

Let U_j^n be a numerical approximation of the exact solution $u_j^n = u(x_j, t^n)$ at the grid points $(x_j, t^n) \in \omega_h \times \omega_\tau$. For functions defined on the grid we introduce the forward and backward difference quotients with respect to x

$$\partial_x U_j^n = (U_{j+1}^n - U_j^n)/h, \quad \partial_{\bar{x}} U_j^n = (U_j^n - U_{j-1}^n)/h$$

and similarly the backward difference quotient and the averaging operator with respect to t

$$\partial_{\bar{t}} U_j^n = (U_j^n - U_j^{n-1})/\tau, \quad U_j^{n-0.5} = 0.5\left(U_j^n + U_j^{n-1}\right).$$

We approximate the differential equation (4.1) by the Crank-Nicolson finite difference scheme [28]

$$i\partial_{\bar{t}} U_j^n + \partial_x \partial_{\bar{x}} U_j^{n-0.5} = 0, \quad x_j \in \omega_h, \;\; n > 0. \tag{4.3}$$

A very interesting approach to construct the approximate local artificial boundary conditions is based on approximation of the transparent boundary condition

$$\partial_n u + e^{-i\frac{\pi}{4}} D_t^{1/2} u = 0$$

by rational functions. The discrete boundary conditions can be written as:

$$\partial_n u = -e^{-i\frac{\pi}{4}}\left(\left(\sum_{k=0}^{l} a_k\right)u - \sum_{k=1}^{l} a_k d_k \varphi_k\right),\ x = a, b,$$

where $\partial_n u$ is the normal derivative, φ_k are solutions of the initial value problem for ODEs [3]:

$$\frac{d\varphi_k(x,t)}{dt} + d_k\varphi_k(x,t) = u\,(x,t)\,,\ x = A, B,\ k = 1,\ldots,l.$$

Our aim is to find optimal values of parameters $\{a_0, a_1, \ldots a_l, d_1, a_2, \ldots d_l\}$, when the following minimisation problem is solved

$$\begin{aligned} E^c_\infty &= \min_{\{a_k,d_k\}}, \quad E_k(a_k,d_k) = \max_{1\le m\le \widetilde{M}} e_m(X_k), \\ e_m &= \max_{j\in[0,J_m],\, n\in[0,N_m]} \left|u(x_{j,m}, t^n_m) - U^n_{j,m}\right|, \end{aligned} \tag{4.4}$$

and $\widetilde{M}$ specially selected benchmark PDEs are solved.

In all examples we use $l = 3$, i.e., the dimensionality of the optimization problem (4.4) is equal to 7. Here discrete approximations of PDEs represent the tasks v_m in (3.1). To solve v_m we must find solutions of N systems of linear equation with tridiagonal matrix [3]. According to our three-level parallelisation scheme, the calculation of a single value E_k in minimisation problem (4.4) defines the block of tasks V_k.

The systems of linear equations with tridiagonal matrices are solved using Wang's algorithm. It is well known that if the size of a system is J and p processes are used then the computation time can be estimated as

$$T_{Wp} = 17\frac{J}{p} + 8p + T_{c1}(p), \tag{4.5}$$

where $T_{c1}(p)$ defines communication costs. The time to compute a value of the objective function f for the specified equation can be estimated as

$$T_{Op} = c_1\frac{J}{p} + T_{c2}(p). \tag{4.6}$$

In this work instead of theoretical complexity models (4.5) and (4.6) we use $t_m(p)$, $m = 1,\ldots,M$, based on empirical computations for a selected set of benchmark problems (so called model-based estimates). Such an approach takes into account all specific details of the parallel algorithm and the computer system.

It is interesting to note that the complexity of computational task v_m depends on both parameters: the number of linear equations J_m of the system and the number of integration in time steps N_m. The computation time T_{mp} is equal to $N_m t_m(p)$, but the scalability of the parallel algorithm depends on J_m only, since the integration in time is done sequentially step by step.

Next, we present an example with $M = 4$, where four different benchmark PDE problems (4.1) with explicit solutions [37, 42] are defined as:

1.
$$u(t,x)=\frac{\exp\left(-i\pi/4\right)}{\sqrt{4t-i}}\exp\left(\frac{ix^2-6x-36t}{4t-i}\right), \tag{4.7}$$
$x\in[-5,5]$, $t\in[0,0.8]$. The problem is approximated on the uniform grid $J\times N=8000\times 4000$.

2.
$$u(t,x)=\frac{1}{\sqrt[+]{1+it/\alpha}}\exp\left(ik(x-x^{(0)}-kt)\ -\frac{(x-x^{(0)}-2kt)^2}{4(\alpha+it)}\right), \tag{4.8}$$
where $k=100,\alpha=1/120,x^{(0)}=0.8$, $x\in[0,1.5]$, $t\in[0,0.04]$. We use the uniform discretisation grid $J\times N=12000\times 4000$.

3. The solution is defined by (4.7), $x\in[-10,10]$, $t\in[0,2]$. We use the uniform discretisation grid $J\times N=16000\times 10000$.

4. The solution is defined by (4.8), where $k=100,\alpha=1/120,x^{(0)}=0.8$. $x\in[0,2]$, $t\in[0,0.08]$. We use the uniform discretisation grid $J\times N=16000\times 8000$.

Next, we consider the problem (4.4) as a local optimisation problem, which can be solved using an iterative algorithm with a given initial starting point. As a local optimiser Nelder-Mead (see Figure 3 [40]) algorithm is used [25].

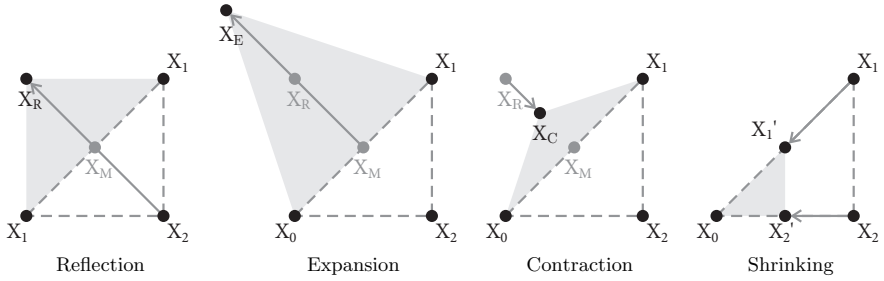

Figure 3. One possible step of the Nelder-Mead algorithm applied to a problem in $\mathbb{R}^2$ [40], here X_i is vertices, X_M – centroid of two worst vertices.

We propose a family of modifications of the original Nelder-Mead algorithm in order to increase the parallelisation degree of it.

Let N be the number of parameters being optimized. Then the simplex has $N+1$ vertices, each of them is represented by a set of parameters written into N-dimensional vectors X_i, $i=1,\ldots,N+1$ ($N+1$ N-dimensional vectors total). We denote $X=[X_1,X_2,\ldots,X_{N+1}]$, $f(X)$ is the objective function, ε defines the stopping criteria. Then Nelder–Mead method is defined by Algorithm 2.

Algorithm 2. The Nelder–Mead method

1: $simplexSolve(f,X,\varepsilon)$:
2: Begin
3: $\bar{F}=sum(\{f(X_i),i=1,\ldots,N+1\})/(N+1)$

4: $E = sum(\{\left(f(X_i) - \bar{F}\right)^2, i = 1, \ldots, N+1\})/(N+1)$
5: **if** $E < \varepsilon$ **then**
6: **return** X
7: **end if**
8: sort X in such way that $f(X_1) \leq f(X_2) \leq \cdots \leq f(X_{N+1})$
9: $x_0 = sum([X_1, X_2, \ldots, X_N])/N$
10: $X_R = x_0 + \alpha(x_0 - X_{N+1})$
11: **if** $f(X_1) \leq f(X_R) < f(X_N)$ **then**
12: $X_{n+1} = X_R$
13: **goto** 2
14: **end if**
15: **if** $f(X_1) > f(X_R))$ **then**
16: $X_E = x_0 + \gamma(X_R - x_0)$
17: **if** $f(X_E) < f(X_R))$ **then**
18: $X_{N+1} = X_E$
19: **else**
20: $X_{N+1} = X_R$
21: **end if**
22: **goto** 2
23: **end if**
24: $X_C = x_0 + \rho(x_{N+1} - x_0)$
25: **if** $f(X_C) < f(x_{N+1}))$ **then**
26: $X_{N+1} = X_C$
27: **goto** 2
28: **end if**
29: $X_i = X_1 + \sigma(X_i - X_1), \forall i \in \{2, \ldots, N+1\}$
30: **goto** 2
31: End

At each iteration the following four different scenarios can be obtained:

- Reflection – compute the value f_R of the objective function at the point $X_R = x_0 + \alpha(x_0 - x_N)$, i.e. $f(x_1) \leq f_R \leq f(x_{N+1})$. Depending on the value f_R this can be the end of the iteration.

- Expansion – depending on the f_R ($f_R < f(x_1)$), an additional computation of the objective function at the point $X_E = x_0 + \gamma(x_R - x_0)$ is done, meaning the total computation of two objective function values: f_R, f_E.

- Contraction – depending on the f_R ($f_R \geq f(x_1)$), an additional computation of the objective function at point $X_C = x_0 + \rho(x_{N+1} - x_0)$ is done, meaning the total computation of two objective function values: f_R, f_C.

- Compression – compute m objective function values, as well as f_R and f_C i.e. $X_i = X_1 + \sigma(X_i - X_1), \forall i \in \{2, \ldots, N+1\}$. Here m is the number of simplex dimensions.

The first three scenarios require to compute one or two values of the objective function from the set: f_R, f_E, f_C. We can neglect the last scenario,

because it occurs very rarely. For the first three scenarios we propose to compute two or three points simultaneously. Algorithmically this means that we change the order of computations, which let us to parallelise the Nelder-Mead method. In most cases only two of three points will be used. Therefore, some redundant calculations will be performed, however, this modification gives an additional parallelisation of computations.

Thus, two modifications of the sequential (A_1) Nelder-Mead method are defined. For A_2 we compute in parallel two values of the objective function f_R, f_E and for A_3 we compute in parallel all three values f_R, f_E, f_C. As a test case we assume that the first scenario is relatively rare, the extension step is done with probability 2/3 and contraction steps occurs with probability 1/3. Then we get that the algorithmic efficiency of the proposed parallel modifications are equal to $\gamma_2 = 0.75$ and $\gamma_3 = 2/3$, respectively. We note, that these values can be estimated more precisely for specific applications, and one example is given for the computational experiments with the Rosenbrock objective function in Section 6.

5 Experimental results

In this section we present results of the parallel scalability experiments. All numerical tests in this work were performed on the computer cluster "HPC Sauletekis" at the High Performance Computing Center of Vilnius University, Faculty of Physics. We have used up to 8 nodes with Intel® Xeon® processors E5-2670 with 16 cores (2.60 GHz) and 128 GB of RAM per node. Computational nodes are interconnected via the InfiniBand network.

Parallel algorithms are implemented using MPI with C++ language. The Nelder-Mead method itself is performed on the main process because it takes the negligibly small part of calculations, the main process also serves as a master which distributes the tasks, i.e. simplex points, which are calculated by different groups of processes simultaneously. Each group of processes is implemented as MPI communicator and solves the Schrödinger problem using Wang's algorithm.

Our main goal is to investigate the efficiency of the proposed three level template of workload distribution between processes. First, we have selected three specific benchmarks with different discretizations (4.2), when $M = 4$ discrete approximations of PDEs (4.3) are solved numerically to compute one value of the objective function. The sizes $(J_m \times N_m)$, $m = 1, \ldots, 4$ of discrete problems are given in Table 1.

Table 1. Benchmarks with different sizes $J_m \times N_m$ of the discrete problem (4.3).

	Benchmark 1	Benchmark 2	Benchmark 3
Eq.	Sizes	Sizes	Sizes
1	8000×40000	8000×20000	8000×10000
2	4000×20000	4000×20000	2000×20000
3	2000×20000	4000×10000	2000×10000
4	2000×10000	2000×10000	1000×20000

In the first benchmark the size of one task v_1 is much bigger than sizes of the remaining three tasks. In the second benchmark two changes are done. They make this set of tasks more suited for parallelisation on large number of processes: the size of task v_1 is reduced twice by taking a smaller number of time steps N_1; the size of task v_3 remains the same, but the number of points J_3 is increased twice, therefore the scalability of Wang's algorithm is improved for this task. In the third benchmark the relative sizes of tasks v_m are more homogeneous than in the first benchmark, but this result is achieved by reducing the number of space grid points J_2, J_4, therefore the scalability of Wang's algorithm is decreased for these two tasks, especially for v_4.

First, we exclude the efficiency condition from the load balancing algorithm by taking $E_{min} = 0$ in (3.3). The distribution of processors between tasks are presented in Tables 2–4. We also provide the actual computation time T_p along with T_{Mp} that were predicted by the theoretical complexity model. As we can see from Table 2 the model and experimental times are close to each other and still experimental times are smaller than the model predictions. This result is expected since model times (see Figure 2) were based on benchmarks, that imitate a pessimistic scenario when all nodes were artificially loaded during experiments. The prediction accuracy depends on many parameters such as cluster architecture, network loads during computations.

For comparison purposes we also provide the results obtained by using the two-level parallelisation template. $K = 1$, then the first level of the three-level template is not used.

It is important to note, that in Tables 2-5 we present the CPU time needed to compute one useful point (4.4), i.e., the actual time is divided by $\gamma_k\, k$, which represents the usefulness of computations. Optimal algorithm A_k is selected automatically using the approach that was described above.

As it follows from Table 2, the usage of the first level with $k = 3$ and $P = 128$ processes increases the potential speed-up from 38.75 to 60.44. If $P = 128$ and $k = 1$ then only 70 processes are used. However the result is very similar to the case when $P = 64$ processes are used, which means that these additional resources are used very inefficiently.

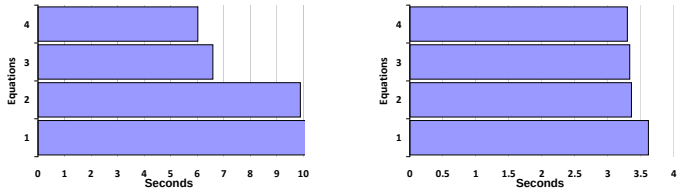

Figure 4. Experimental model times for Benchmark 1 with $p = 16$(left) and $p = 64$(right).

In the Figure 4 the Gantt charts show theoretical model time $t_m(p)$, that is needed to obtain the solutions of different equations. The workload distribution becomes closer to uniform as the number of processes is increased.

Table 2. The results for Benchmark 1. T_p is the CPU time in seconds required to compute one useful point (4.4).

p	16	32	64
	$k = 1$		
Eq. 1	10	22	50
Eq. 2	3	5	8
Eq. 3	2	3	4
Eq. 4	1	2	2
Total number of p	16	32	64
Model T_{Mp}	11.145	5.784	3.614
T_p	11.003	5.394	3.608
Speed-up	12.679	25.862	38.664

p	96	128	128
	$k = 2$	$k = 3$	$K = 1$
Eq. 1	34	29	56
Eq. 2	8	7	8
Eq. 3	4	4	4
Eq. 4	2	2	2
Total number of p	96	126	70
Model T_{Mp}	2.742	2.272	3.605
T_p	2.719	2.308	3.600
Speed-up	51.307	60.444	38.75

Table 3. The results for Benchmark 2. T_p is the CPU time in seconds required to compute one useful point (4.4).

p	16	32	64	96	128	128
	$k = 1$			$k = 2$		$K = 1$
Eq. 1	9	18	37	26	37	56
Eq. 2	4	8	15	12	15	18
Eq. 3	2	4	8	7	8	8
Eq. 4	1	2	4	3	4	4
Total number of p	16	32	64	96	128	86
Model time	6.59	3.36	2.01	1.65	1.34	1.8
T_p	6.69	3.37	1.98	1.62	1.33	1.86
Speed-up	13.6	27.03	46.03	56.24	68.25	49.03

5.1 The control of efficiency

The reduction of the energy consumption is an important goal, especially when increasment of computation speed-up are small for additional processes. The presented results indicate that in some cases there is a highly inefficient usage of computational resources.

For the purposes of controlling the efficiency of calculations the condition

Table 4. The results for Benchmark 3. T_p is the CPU time in seconds required to compute one useful point (4.4).

p	16	32	64	96	128	128
	$k=1$			$k=2$		$K=1$
Eq. 1	8	16	32	24	32	56
Eq. 2	4	8	16	12	16	31
Eq. 3	2	4	8	6	8	8
Eq. 4	2	4	8	6	8	9
Total number of p	16	32	64	96	128	104
Model time	3.33	1.76	1.05	0.87	0.7	0.9
T_p	3.38	1.76	1.06	0.86	0.7	0.95
Speed-up	14.33	27.55	45.96	56.72	69.08	51.23

(3.3) was introduced in Algorithm 1. This condition guarantees that the efficiency of the numerical solution of each block of tasks will be at least E_{min}. It is important to note, that we are not attempting to generate optimal mappings of processors – we have developed an heuristic that provides the quality of distribution of tasks, that is sufficient for the most practical purposes. The quality of the algorithm is improved when more processors are available.

Next, a more detailed analysis of the Benchmark 1 is provided. In Table 5 the results for $E_{min} > 0$ are presented. Comparing the results in Table 5 with the results in Table 2 we see that for $K = 1$ and $E_{min} = 0.6$ the number of processes for the first equation is decreased by 14, however, the computation times are almost the same as it was in the case of $E_{min} = 0$. Also, for $K = 3$ the efficiency requirement begins to limit the number of processes for $E_{min} = 0.75$ and it decreases further with $E_{min} = 0.8$.

However, even then a three level approach with $K = 3$ is superior to the standard two-level approach in terms of the final speed-up. The results in Table 5 indicate that even for the efficiency limitation $E_{min} = 0.75$ the proposed three-level approach lets to maintain a big number of parallel processes active, this number is equal to $(26 + 7 + 4 + 2) \times 3 = 117$. The speed-up is 56 and the efficiency of the parallel algorithm is $56/117 \approx 0.48$. The last column in Table 5 with $K = 1$ presents the results for the two-level approach (without the first level). A straightforward two-level parallelisation approach would have the limited parallelisation possibility especially for problems of the size $J = 2000$. For such small subproblems it would be possible to utilise only up to 32 processes (Figure 2), the speed-up would be quite limited as well.

Note, that all previous results represents the analysis based on a single Nelder-Mead iteration. Next, we solve the actual real-world optimisation problem (4.4). The maximum number of processes $P = 128$ the load balancing algorithm has selected $k = 1$. The number of Nelder-Mead method iterations was fixed to 1000. The parallel and sequentional versions gave the same results the minimum value of the error $E_{\infty}^{C} = 0.0806$. The sequentional version of computations took 180286 seconds, the parallel version computations took 2232 seconds. Thus, a speed-up factor of 81.8 was achieved. The selection of

Table 5. The results for Benchmark 1 with $E_{min} > 0$. T_p is the CPU time in seconds required to compute one useful point (4.4).

p	128		
E_{min}	0.75 $k=3$	0.8 $k=3$	0.6 $K=1$
Eq. 1	26	19	42
Eq. 2	7	5	8
Eq. 3	4	3	4
Eq. 4	2	1	2
Total number of p	117	84	56
Model T_{Mp}	2.45	3.17	3.84
T_p	2.49	3.08	3.76
Speed-up	56.03	45.37	37.11

$k = 1$ indicates that the number of processes can be greatly increased – the algorithm has selected $k = 1$ automatically for a given number of processes.

6 The comparison of different Nelder-Mead parallelisation methods

Here we present the analysis of the convergence properties of different modifications of the Nelder-Mead method. As it was mentioned before, the convergence rate of the selected algorithm directly affects the parallelisation efficiency, which is represented by γ_k, where k is the parallelisation degree. In this section we measure γ_k by measuring the experimental parallel efficiency of algorithms.

The detailed analysis of convergence behaviour for different objective functions is out of the scope of this research. However, the objective function from the previous sections is suitable for a narrow class of applications. Thus, to perform a comparison of different parallel versions of Nelder-Mead method we minimise the Rosenbrock objective function that is widely used by researchers in the field of optimisation theory [8, 36].

We show that in the case of the Rosenbrock function the real experimental γ_k values are different than were assumed to be in the experiments of the previous sections. The reason is that the significant number of iterations require to compute only one point F_R.

We compare the results of our parallel modification of the Nelder-Mead method with the state-of-art technique proposed in [22]. As a benchmark we use the Rosenbrock function

$$f(x_1, \ldots, x_d) = \sum_{i=1}^{d-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2,$$

which makes the optimisation problem challenging. It should be noted that the parallel algorithm [22] can achieve the parallelisation degree K that is equal to the optimisation problem dimension d. Thus potentially this algorithm is well suited for parallel computers with a big number of processes.

Table 6. The γ_k values for direct Nelder-Mead parallelisation.

k	$d=3$	$d=6$	$d=7$
2	0.603	0.604	0.606
3	0.584	0.517	0.502

In the Table 6 we compare three cases $d = 3, 6, 7$: $d = 3$ – the minimum, that is needed for parallelilsation with both methods, $d = 7$ – the case that was studied in previous section, $d = 6$ – to show the tendency for smaller d. We provide results obtained when the Rosenbrock function of different dimensions $d = 3, 6, 7$ was minimized by using our parallel modification of the Nelder-Mead method. The values of the efficiency coefficients γ_k are presented. They show that this parallel algorithm is quite stable and it is well-suited to be used in the three-level template solver for small dimension objective functions.

Table 7. The γ_k values for the parallel Nelder-Mead algorithm from (Lee and Wiswall, 2007).

k	$d=3$	$d=6$	$d=7$
2	0.668	0.685	0.714
3	–	0.436	0.454
4	–	0.023	0.104
5	–	0.001	0.002
6	–	–	0.001

Table 7 presents results obtained by using the state-of-the-art parallel Nelder-Mead algorithm from [22]. It follows, that in all investigated cases the parallelisation degree is very limited, since the convergence drops significantly when the parallelisation degree is increased. This method is mainly targeted to solve problems when the dimension of the objective function is big (e.g. problems in financial mathematics, when $d \approx 100$).

7 Conclusions

In this paper, we introduced a three-level parallelisation template which utilises a new model-based load balancing technique which is based on experimental data. This technique was tested for three benchmarks. The experimental results confirmed the good accuracy of the new time prediction model.

Comparing the three-level template to the classical two-level scheme, the proposed scheme looks more promising for development of efficient parallel algorithms in the case when a big number is computational resources is available.

The possibilities of the three-level parallelisation template are demonstrated for solving local optimization problems. On the first level a well-known Nelder-Mead algorithm was used as a basic algorithm. We also proposed a family of parallel versions of this method, which increases the parallelisation degree up to the factor of three. The proposed load balancing algorithm chooses the

optimal version of the parallel Nelder-Mead algorithm. It dynamically increases the parallelisation degree on the first level when the speed-up of the second and third levels begins to saturate.

For the considered test problem on the second level M PDEs were solved numerically and on the third parallelisation level Wang's algorithm was used to solve systems of linear equations. It is shown that there exists a limit for the speed-up that can be achieved due to limitations of Wang's algorithm. The proposed approach extends the parallelisation degree allowing to achieve an additional speed-up.

The proposed load balancing algorithm limits the size of computational resources to preserve the efficiency requirement which can be controlled by selecting the parameter E_{min}.

Despite the fact that we introduce the three-level scheme which can be seen as a general approach, it is suitable only if some specific properties are fulfiled. One of the requirements is that two-level parallelisation should be accompanied with some algorithm on higher level that has alternatives with different paralellisation degrees. The the new scheme uses these algorithms to extend the parallelisation degree of the total algorithm.

Additionally to the studied application case, some of other possible applications were mentioned in this paper, however, we do not pretend to prove the usefulness of this technique in other cases. It depends on specific properties of such problems and application of this new three level scheme is not an easy and straightforward task. These topics are a subject of further research, thus, it is out of scope of the current article.

Acknowledgements

Computations were performed using resources at the High Performance Computing Centre "HPC Sauletekis" in Vilnius University Faculty of Physics.

References

[1] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*, **30**:483–485, 1967. https://doi.org/10.1145/1465482.1465560.

[2] J. Bilmes, K. Asanovic, Ch.-W. Chin and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pp. 253–260, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 9781450328401. https://doi.org/10.1145/2591635.2667174.

[3] A. Bugajev, R. Čiegis, R. Kriauzienė, T. Leonavičienė and J. Žilinskas. On the accuracy of some absorbing boundary conditions for the Schrödinger equation. *Mathematical Modelling and Analysis*, **22**(3):408–423, 2017. ISSN 1392-6292. https://doi.org/10.3846/13926292.2017.1306725.

[4] J. Carretero, S. Distefano, D. Petcu, D. Pop, Th. Rauber, G. Rünger and D.E. Singh. Energy-efficient algorithms for ultrascale systems. *Supercomputing frontiers and innovations*, **2**(2):77–104, 2015.

[5] R. Ciegis and M. Baravykaite. Implementation of a black-box global optimization algorithm with a parallel branch and bound template. *Applied Parallel Computing: State of the Art in Scientific Computing*, **4699**:1115–1125, 2007. ISSN 0302-9743. https://doi.org/10.1007/978-3-540-75755-9_129.

[6] R. Čiegis and G. Šilko. A scheme for partitioning regular graphs. In R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski and K. Wiatr(Eds.), *Proc. 4th International Conference on Parallel Processing and Applied Mathematics (PPAM2001, Naleczsow, Poland, September9-12, 2001)*, volume 2328 of *Lecture Notes in Computer Science*, pp. 404–409, Berlin, Germany, 2002. Springer.

[7] A. Datta, A. Kaur, T. Lauer and S. Chabbouh. Exploiting multi–core and many–core parallelism for subspace clustering. *International Journal of Applied Mathematics and Computer Science*, **29**(1):81–91, 2019. https://doi.org/10.2478/amcs-2019-0006.

[8] I. Fajfar, Á. Bűrmen and J. Puhan. The Nelder–Mead simplex algorithm with perturbed centroid for high-dimensional function optimization. *Optimization Letters*, **13**(5):1011–1025, 2019. https://doi.org/10.1007/s11590-018-1306-2.

[9] D.E. Finkel. DIRECT optimization algorithm user guide. *Center for Research in Scientific Computation, North Carolina State University*, **2**:1–14, 2003.

[10] M. Frigo and S.G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, **93**(2):216–231, 2005. https://doi.org/10.1109/JPROC.2004.840301.

[11] M. Furuichi and D. Nishiura. Iterative load-balancing method with multigrid level relaxation for particle simulation with short-range interactions. *Computer Physics Communications*, **219**:135–148, 2017. ISSN 0010-4655. https://doi.org/10.1016/j.cpc.2017.05.015.

[12] F. Glover. Tabu search–part I. *ORSA Journal on Computing*, **1**(3):190–206, 1989. https://doi.org/10.1287/ijoc.1.3.190.

[13] F. Glover. Tabu search–part II. *ORSA Journal on Computing*, **2**(1):4–32, 1990. https://doi.org/10.1287/ijoc.2.1.4.

[14] I. Huismann, J. Stiller and J. Frohlich. Two-level parallelization of a fluid mechanics algorithm exploiting hardware heterogeneity. *Computers & Fluids*, **117**:114–124, 2015. ISSN 0045-7930. https://doi.org/10.1016/j.compfluid.2015.05.012.

[15] E.-J. Im and K. Yelick. Optimizing sparse matrix computations for register reuse in SPARSITY. In V.N. Alexandrov, J.J. Dongarra, B.A. Juliano, R.S. Renner and C.J.K. Tan(Eds.), *Computational Science – ICCS 2001*, volume 2073, pp. 127–136, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-45545-0_22.

[16] S. Imam and V. Sarkar. Load balancing prioritized tasks via work-stealing. *Euro-Par 2015: Parallel Processing*, **9233**:222–234, 2015. ISSN 0302-9743. https://doi.org/10.1007/978-3-662-48096-0_18.

[17] C.T. Kelley. *Iterative methods for optimization*. Society for Industrial and Applied Mathematics, Philadelphia, 1999. https://doi.org/10.1137/1.9781611970920.

[18] S. Kirkpatrick, C.D. Gelatt and M.P. Vecchi. Optimization by simulated annealing. *Science*, **220**(4598):671–680, 1983. https://doi.org/10.1126/science.220.4598.671.

[19] K. Klein and J. Neira. Nelder-Mead simplex optimization routine for large-scale problems: A distributed memory implementation. *Computational Economics*, **43**(4):447–461, Apr 2014. ISSN 1572-9974. https://doi.org/10.1007/s10614-013-9377-8.

[20] A. Lastovetsky and R.R. Manumachu. New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters. *IEEE Transactions on Parallel and Distributed Systems*, **28**(4):1119–1133, 2017. ISSN 1045-9219. https://doi.org/10.1109/tpds.2016.2608824.

[21] A. Lastovetsky, L. Szustak and R. Wyrzykowski. Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing. *IEEE Transactions on Parallel and Distributed Systems*, **28**(3):787–797, 2017. ISSN 1045-9219. https://doi.org/10.1109/tpds.2016.2599527.

[22] D. Lee and M. Wiswall. A parallel implementation of the simplex function minimization routine. *Computational Economics*, **30**(2):171–187, 2007. https://doi.org/10.1007/s10614-007-9094-2.

[23] A. Llanes, J.M. Cecilia, A. Sanchez, J.M. Garcia, M. Amos and M. Ujaldon. Dynamic load balancing on heterogeneous clusters for parallel ant colony optimization. *Cluster Computing-the Journal of Networks Software Tools and Applications*, **19**(1):1–11, 2016. ISSN 1386-7857. https://doi.org/10.1007/s10586-016-0534-4.

[24] K.I.M. McKinnon. Convergence of the Nelder–Mead simplex method to a nonstationary point. *SIAM Journal on Optimization*, **9**(1):148–158, 1998. https://doi.org/10.1137/S1052623496303482.

[25] J.A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, **7**(4):308–313, 1965. https://doi.org/10.1093/comjnl/7.4.308.

[26] B.S. Nemalikanti, P. Sindhura, P.K. Tumalla and S. Vemuru. Achieving green computing through algorithmic efficiency. *i-Manager's Journal on Information Technology*, **1**(1):39, 2011.

[27] B. Perez, E. Stafford, J. Bosque and R. Beivide. Energy efficiency of load balancing for data-parallel applications in heterogeneous systems. *Journal of Supercomputing*, **73**(1):330–342, 2017. ISSN 0920-8542. https://doi.org/10.1007/s11227-016-1864-y.

[28] M. Radziunas, R. Čiegis and A. Mirinavičius. On compact high order finite difference schemes for linear Schrödinger problem on non-uniform meshes. *International Journal of Numerical Analysis and Modelling*, **11**(2):303–314, 2014.

[29] J.A. Rico-Gallego and J.C. Diaz-Martin. τ-Lop: Modeling performance of shared memory MPI. *Parallel Computing*, **46**:14–31, 2015. ISSN 0167-8191. https://doi.org/10.1016/j.parco.2015.02.006.

[30] J.A. Rico-Gallego, A.L. Lastovetsky and J.C. Diaz-Martin. Model-based estimation of the communication cost of hybrid data-parallel applications on heterogeneous clusters. *Ieee Transactions on Parallel and Distributed Systems*, **28**(11):3215–3228, 2017. ISSN 1045-9219. https://doi.org/10.1109/tpds.2017.2715809.

[31] R.D. Righi, R.D. Gomes, V.F. Rodrigues, C.A. da Costa, A.M. Alberti, L.L. Pilla and P.O.A. Navaux. MigPF: Towards on self-organizing process rescheduling of bulk-synchronous parallel applications. *Future Generation Computer Systems-the International Journal of Escience*, **78**:272–286, 2018. ISSN 0167-739X. https://doi.org/10.1016/j.future.2016.05.004.

[32] B. Saha. Green computing: Current research trends. *International Journal of Computer Sciences and Engineering*, **6**, 05 2018. https://doi.org/10.26438/ijcse/v6i3.467469.

[33] A. Sharma and M. Kaur. An efficient task scheduling of multiprocessor using genetic algorithm based on task height. *Journal of Information Technology & Software Engineering*, **5**(2):1000151, 2015. https://doi.org/10.4172/2165-7866.1000151.

[34] R. Singh. Task scheduling in parallel systems using genetic algorithm. *International Journal of Computer Applications*, **108**(16):34–40, 2014. https://doi.org/10.5120/18999-0470.

[35] W. Spendley, G.R. Hext and F.R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, **4**(4):441–461, 1962. https://doi.org/10.1080/00401706.1962.10490033.

[36] L. Stripinis, R. Paulavičius and J. Žilinskas. Improved scheme for selection of potentially optimal hyper-rectangles in DIRECT. *Optimization Letters*, **12**(7):1699–1712, Oct 2018. ISSN 1862-4480. https://doi.org/10.1007/s11590-017-1228-4.

[37] J. Szeftel. Design of absorbing boundary conditions for Schrödinger equations in R^d. *SIAM Journal on Numerical Analysis*, **42**:1527–1551, 2004. https://doi.org/10.1137/S0036142902418345.

[38] R. Vuduc, J.W. Demmel and K.A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, **16**:521–530, 2005. https://doi.org/10.1088/1742-6596/16/1/071.

[39] H.H. Wang. A parallel method for tridiagonal equations. *ACM Transactions on Mathematical Software (TOMS)*, **7**(2):170–183, 1981.

[40] Th. Weise. *Global optimization algorithms. Theory and application.* Self-Published Thomas Weise, 2009. Available from Internet: `http://www.it-weise.de`.

[41] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing, 1998. SC98. IEEE/ACM Conference on*, pp. 38–38. IEEE, 1998.

[42] A. Zlotnik and I. Zlotnik. Remarks on discrete and semi-discrete transparent boundary conditions for solving the time-dependent Schrödinger equation on the half-axis. *Russian Journal of Numerical Analysis and Mathematical Modelling*, **31**(1):51–64, 2016. https://doi.org/10.1515/rnam-2016-0005.