Adaptive Scheduling of Parallel Computations for SPMD Tasks

Mikhail Panshenskov and Alexander Vakhitov

Saint Petersburg State University, Department of Mathematics and Mechanics,
Universitetsky pr. 28,
198504 Peterhof, Saint Petersburg, Russia

Abstract. A scheduling algorithm is proposed for large-scale, heterogeneous distributed systems working on SPMD tasks with homogeneous input. The new algorithm is based on stochastic optimization using a modified least squares method for the identification of communication and performance parameters. The model of computation involves a server distributing tasks to clients. The goal of the optimization is to reduce execution time by the clients. The costs of getting the task from the server, execution of the task and sending the results back are estimated; and the scheduling is based on adaptive division of work (input for the clients) into blocks.

1 Introduction

Results about load balancing are of high interest in the field of parallel algorithms. In this paper the case of SPMD (Single Program - Multiple Data) without any synchronizations and with *homogeneous* input is considered. For such programs authors propose a method to reduce imbalance caused by different performance and answer time of each processor. Primarily the algorithm proposed can be applied for large-scale heterogeneous computer systems. The method is based on stochastic optimization. The main motivation of using this method is fundamental uncertainty in the system and the need of fast adaptation to the parameter changes [1].

SPMD computing for independent tasks is a subclass of computing models with the following properties: 1. Mechanisms for forming independent tasks by input data and combining the general output by the outputs of subtasks 2. Processing of each task is evaluated by the same program.

Good discussion about problems suitable for SPMD computing is provided in [3]. The results about SPMD computing and adaptive techniques used there only start to appear. Earlier, the comparatively simple strategies were used [4]. Later, there appeared several different problems of adaptive learning class in scheduling field. Firstly, the distributed computing runtime can adopt the number of client computers used by any task in a way that if all the given computers are busy, it can get more, on the other way some will be taken and given to another task. This problem definition is stated in [5,3]. The approach by He, Hsu and Leiserson [5] is generally preferred, something of this sort can be added to the research

O. Gervasi and M. Gavrilova (Eds.): ICCSA 2007, LNCS 4706, Part II, pp. 38–50, 2007.
© Springer-Verlag Berlin Heidelberg 2007

proposed here to generalize the problem solved. Here we address only problem of optimal execution of the one SPMD task; in [5,3] it is assumed that there are several concurrent. However, none of the papers deal with external priority (for example, amount of money give for task solution up to some time moment), trying to give the task only what it asks for during execution. We find that external priority is a step to business use of distributed computing.

Some authors propose to adapt to the imbalance by changing the "relative power" on each step when imbalance is more than some threshold. In our opinion, this is not often needed. If the system performs in average well, then there is no need to change the work given to target systems on each step. However, authors [4] state that their load balancing works well.

Adaptation of distribution to network structure is also researched. There are several methods to identify clusters of well-connected computers to assign tasks to them using this information. The approach proposed here can be extended to dynamical hierarchical structure, when network identification will be of great interest.

In this paper we consider special subclass of SPMD computing: SPMD computing for independent tasks with homogeneous input.

SPMD computing for independent tasks with homogeneous input is SPMD computing for independent tasks when for the given machine every task can be computed in the same time.

SPMD computing for independent tasks with homogeneous input is used for wide class of problems, including

– visualization problems (visualization of Julia fractals) [17]
– mathematical modeling (Monte-Carlo methods for surface reactions) [2]
– information retrieval problems (SETI@Home - search for UFO signals)[10]

The main problem of parallel computations nowadays is how to construct a parallel program correctly. Many programming language extensions and software applications are being developed to reduce the costs of parallel programs construction.

However, after the construction, the program should be run *effectively.* This thesis can have different formal implications. We consider the program evaluation effective if it takes shortest time to finish. Usually computational resources, are not acquired for free, so it is reasonable to set an upper bound on the number of processors which can be used in computations. Scheduling is the mechanism of distribution the tasks between the processors to achieve minimal execution time for the whole parallel program. The result of scheduling is load balancing, which can be measured and improved by modifying scheduling algorithm. The consequences of good balancing in the sense described are increase of system throughput and performance. In this paper we present an algorithm for scheduling of SPMD computation for independent tasks with homogeneous input. Formulation of the cost function used in this paper can be found in the next section.

In parallel computing systems scheduling is always present, however the algorithms do not vary greatly. We will observe next different algorithms of scheduling for SPMD computations for independent tasks with homogeneous

input. If accurate information about the task complexity and processors performance is known before runtime, *static* [6] scheduling can be done. *Static* scheduling is a class of scheduling algorithms when distribution of tasks between processors is defined before runtime. This approach is good when we can be sure in the characteristics of the system and task. For instance, if we have run the task several times on the system, and it demonstrates to be highly consistent running this task. The opposite is dynamic scheduling.

Dynamic scheduling is a class of scheduling algorithms when the decisions about the distribution of the tasks between processors are done in runtime using the available information about results of runs of already finished tasks. Processors can change their performance. They can even leave or join the computations. This leads to necessity of dynamic balancing scenarios instead of static balancing.Study about taxonomy of scheduling algorithms for parallel computations is done in [7].

Dynamic (non-cooperative[1]) scheduling algorithms can be divided into distributed and centralized scheduling. Work-sharing and work-stealing [8,9,5] are the only cases of distributed scheduling, differing in the initiator of imbalance reduction.

In case of dynamic distributed scheduling, the tasks are given to systems in equal amounts, and when one processor finishes execution, it either takes new task from overloaded neighbor (work-stealing) or waits until busy neighbor gives it (work-sharing). This can lead to communicational difficulties close to the end of task execution, when many of the processors finish the work which was initially their and try to take a part of work from randomly chosen overloaded processor.

The another class of dynamic scheduling algorithms is centralized scheduling. In this case, there exists special target system (broker), which runs scheduling algorithm, and other target systems get work from the broker. the algorithm proposed in the paper belongs to this class (dynamic centralized scheduling). Centralized scheduling gives different meaning to computers participating in the system, so *Target system* is a computer, which participates in the distributed computing system and executes tasks given by user *Broker* is a computer, to which resource management is delegated. There are two levels of broker activities. Firstly, broker gives resources for particular computation. Secondarily, broker runs scheduling algorithm and manages which atomic tasks are being executed on which target systems.

If the number of target systems is large, it can be reduced introducing the hierarchical topology [9]. In this paper we concentrate on non-hierarchical, one-level topology.

In this paper, we consider only one class of distributed computations and therefore we are interested in the algorithm for second-level broker activity.

In the class of computations considered, usually atomic tasks are very small and grouped into blocks. The dynamic scheduling algorithms class for this class of computations can be divided into subclasses with constant and varying size of

[1] Cooperative scheduling scenarios assume more complicated multi-agent methods, which are not covered in this paper.

block. The size of block can vary int time and between target systems. The algorithm proposed here uses both possibilities. First allows to overcome some communication issues, second allows to perform balanced scheduling better. Comparing to other subclasses of centralized dynamic scheduling, we can say, that these possibilities allow adaptation of the scheduling to performance and communication parameters of every target system.

The paper is organized as follows. Section 2 determines the model of SPMD computations for independent tasks with homogeneous input and cost function. Section 3 is a short excerpt about stochastic optimization methods proposed to use in the field of scheduling for distributed computing. The method based on least squares estimator is a part of scheduling algorithm proposed and it is formalized as a sequence of steps in this section. Section 4 is central: the other part of scheduling algorithm which determines sizes of blocks is described and discussed there. The last, 5th section, contains conclusions and further directions of research.

2 Model of Distributed SPMD Homogeneous Computing and the Cost Function

2.1 Model

The case of SPMD computations for independent tasks with homogeneous input is considered. We make several assumptions about the system, which we find realistic. These are:

1. Tasks are given as blocks of iterations; *Iteration* is an atomic work unit of the SPMD-computation.
2. Communication with client can be done in asynchronous way. It can be regarded as the communication with two threads on the client computer, one for calculation and one for communication.
3. Broker (we can call it server also) costs on communication are not counted; broker is powerful enough to perform communication with arbitrary number of target systems (clients) without any overhead; the time of task execution depends on the times of executions of client applications.
4. The cost of communication and execution of the block of iterations on the target system linearly depends on the number of iterations in a block (*block size*).

We define cost function as maximal execution time on client. The work of target system is iteration of the cycle (with corresponding time period label in brackets):

1. getting the task from server (c_i)
2. execution of the task (L_i)
3. sending the results back to server (n_i)

The sequence diagram for this process is in the Fig. 1.

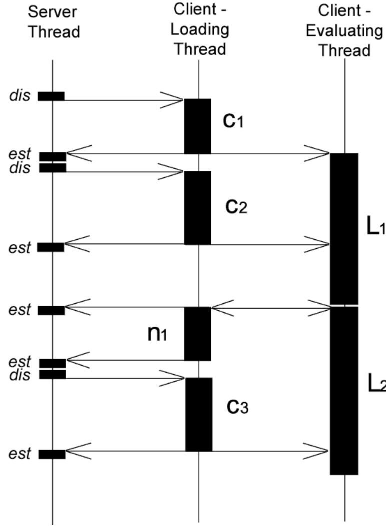

Fig. 1. Algorithm scenario diagram

First time, phase 1 can be done only before the phase 2. Later, phase 1 and 3 can be done in asynchronous way with phase 2. Only the last time the task result is sent back to the server after the total execution finishes.

If the asynchronous task loading is not performed before the previous task execution finishes, then task loading is aborted notifying the server that task need to be loaded again. After, the results are sent back and new task is loaded the next task is executed. This event has corresponding indicator function $\chi_i^{(m)}$.

$$\max_m \{c_1^{(m)} + \sum_{i=1}^{N^{(m)}} L_i^{(m)} + \sum_{i=1}^{N^{(m)}-1} (n_{i-1} + c_{i+1})\chi_i^{(m)} + n_{N^{(m)}}\} \to \min \qquad (1)$$

2.2 Discussion

Many cost functions for distributed computations scheduling and SPMD computations [4] in particular are proposed. The ones discussed in paper [4] can be used for independent tasks with homogeneous input. Some cost functions in work [4] evaluate only imbalance.

The possibility of asynchronous updates is one of the ideas how scheduling can reduce not only final imbalance, but also system execution time on the whole run.

Authors find the cost of communication as a critical parameter for distributed computing, following [6,8]. The set of task which can be solved by the enormously fast distributed meta-computers [10] depends on the communication facilities. Possibility to do asynchronous task and results sending seems to be realistic with contemporary multi-threaded processors and highly-specialized computer architecture.

However, the particular task chosen by authors seems to be realistic and actual. The bounds on the problem allow to go deeper into the problem structure and use more advanced special (learning) methods like discussed in the next section.

3 Tracking Performance and Communication Costs

The problems of identification of true performance and communication cost parameters can be solved in similar way. We assume that the model of every such parameter as a linear function of the block size (b_i) is adequate enough. So, we propose to use modified least squares method, similar to described in [16].

The algorithm proposed for scheduling in this paper consists of two parts. One is parameters estimation, which is executed after every communication with target system. It is executed at the moments labeled as *est* (pic. 1). Another is block size determination, which is executed before sending new task to the target system, at the moments labeled as *dis* (pic. 1).

In this part we consider first algorithm. Briefly introducing the ideas of recursive estimation using least squares, we then formulate the steps of the algorithm.

3.1 Modified Least Squares

Let us assume that there is an observed value y_n:

$$y_n(x_n) = a_n + k_n x_n, \tag{2}$$

where x_n is known and a_n, k_n are unknown. From the formula it can be seen that all the values depend on n. Let a_n, k_n be random with some variance and expectation: $Ea_n = a$, $Ek_n = k$. The problem is to identify a and k and then track the changes in their expectation.

The algorithm proposed is of the form of least-squares estimation with some discounting parameter α. Firstly, we define a discounted cost:

$$J_n(\alpha, \hat{a}_n, \hat{k}_n) = \sum_{l=1}^{n} \alpha^{n-l} \|y_n - \hat{a}_n - \hat{k}_n x_n\|^2 \tag{3}$$

In (3) the values $\hat{a}_n$ and $\hat{k}_n$ are the estimated parameters, y_n can be substituted with (2). The parameters can be estimated as follows:

$$e_{y_n} = \|y_n - a_n - k_n x_n\|^2$$

The algorithm for estimation performs following operations on each step:

$$\begin{pmatrix} \hat{a}_{n+1} \\ \hat{k}_{n+1} \end{pmatrix} = \begin{pmatrix} \hat{a}_n \\ \hat{k}_n \end{pmatrix} + Q_n(e_{y_n}), \tag{4}$$

$$Q_{n+1} = \frac{P_n \begin{pmatrix} \hat{1} \\ \hat{x}_n \end{pmatrix}}{\alpha + P_n(1 + x_n^2)},$$

$$P_{n+1} = \frac{1}{\alpha}\left[P_n - \frac{P_n^2(1+x_{n+1}^2)}{\alpha + P_n(1+x_{n+1}^2)}\right]^{-1},$$
$$P_1 = (1+x_1^2)^{-1}.$$

Parameter α is known as forgetting factor, $\alpha < 1$. It defines how fast the parameter is expected to change. There is an adaptive algorithm for adjusting α, which should be used in this case. Due to insufficient space, we reference here the book of Kushner and Yin [16]. The fact of α adjustment allows us to carry on the difficulties with throughput of the network and performance of the target system changing in time.

4 Determination of The Block Size

The algorithm for block size determination is discussed next. To simplify notation, we remove lower indices from c, n, L, assuming that at every moment there is actual estimate of the coefficients in the linear model for each of these parameters. On every step b_i is the current block size. For a target system, there is an upper bound on the block size $Bmax$. The number of not-started tasks is global updateable variable W. The number of tasks needed to do lowering of the block size for all the target systems is $U = \sum B^{(j)} + Bmax^{(j)}$ where j is a number of particular target system and $B^{(j)}$ is a variable which contains a number of tasks needed for target system j to reach the upper bound of block size increasing it by the algorithm proposed.

Algorithm is presented in 3 phases. In brackets near each phase there is a condition needed to be satisfied to move to the next phase.

1. Block size growth ($b_i < Bmax$)
   (a) $\Delta = L_i - n_{i-1}$
   (b) $b_{i+1} = \frac{\Delta - a_L}{k_L}$
   (c) $B = B + b_i$
   (d) add b_i to stack $Q^{(j)}$
   (e) return b_{i+1}
2. Stable block size (W¿U)
   (a) return $Bmax$
3. Lowering of block size
   (a) take b_{i+1} from $Q^{(j)}$
   (b) return b_{i+1}

5 Using Performance and Communication Parameters in Scheduling

Scheduling algorithm can rely on the parameters $(\hat{c}_n, \hat{n}_n, \hat{L}_n)$ estimated by the algorithm discussed in previous section. The parameters of communication (c_i and n_i) can be measured on the server due to interactivity of TCP/IP protocol. Then, the execution time on client L_i can be measured. So, the estimation of the parameters for all target systems can be done on the server. Here the client-server interaction algorithm is proposed, which uses these estimates.

5.1 Conditions on Block Sizes

As it was said earlier, iterations are grouped in blocks and sent to clients from the server. There are two main questions arising about this algorithm of client-server interaction:

1. How big should be the blocks?
2. How often should server send the blocks to client?

Informally, the answer to these questions has to depend on the communication facilities. The new parameter $s^{(m)}$ is the time period between consequent data sending, that is

$$s^{(m)} = t_s(c_i^{(m)}) - t_s(c_{i-1}^{(m)}) \tag{5}$$

for client m. For networks with high bandwidth, $s^{(m)}$ is small while for low bandwidth it is high. The method to find $s^{(m)}$ is discussed later, by now assume that we have some $s^{(m)}$ for every client m.

Earlier we said that in our model first getting task from the server can only be done before execution. Then, the size of this first block should be small. During the execution of the first block, bigger block can be asynchronously got from the server.

Let us denote $t_s(\cdot)$ the starting time and $t_f(\cdot)$ the finishing time of the interval. Then there are obvious rules to follow to asynchronously get tasks:

$$t_s(n_i) = t_f(L_i), \;\; t_f(c_i) \leq t_s(L_{i+1}) \;\; t_f(n_i) = t_s(c_{i+2}) + t_{sch} \tag{6}$$

These conditions mean that the L_i interval should be overlapped by n_{i-1} and c_{i+1} intervals. To make the phase 1 more robust, we can also define r_i "reserve" pseudo-interval which will depend on the error rate (e_{L_i}, see (4)) of the estimate of execution time $\hat{L}_i$. This pseudo-interval will help when estimate is lower than true value of execution time: $L_i = \hat{L}_i - \delta$, $L_i >> \delta > 0$.

Also, server performs calculation of scheduling parameters, that takes some time, that is why in (6) t_{sch} was added, however it is nearly constant.

5.2 Variation of Block Size During Execution

On the first steps with poor knowledge of the system parameters we cannot expect the asynchronous task getting (phase 1 above) to perform good. However, using reliable estimates of the parameters good accuracy can be achieved.

The block size during the run should be comparatively big. There are several reasons for this. One is that to perform phase 1 better, we should have bigger execution periods. Another is iterations affinity, such that several small block perform lower than one big block.

The "asynchronous growth" of block size can be bounded, because we need to do increasing in step-by-step manner, on each step satisfying condition

$$L_i \geq c_{i+1} + n_{i-1} \tag{7}$$

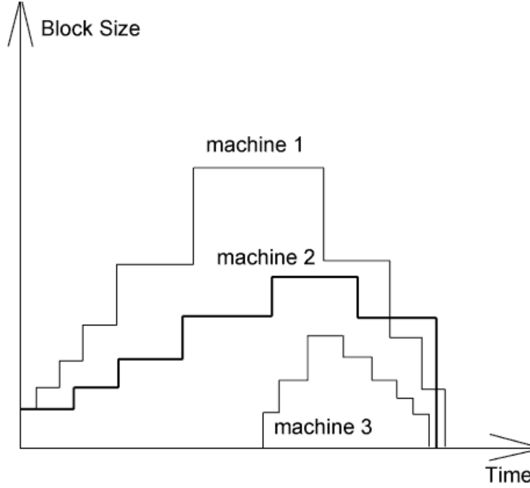

Fig. 2. Block size chart

The parameters in (7) are linear dependent on b_i. This allows us to rewrite conditions (7) as system of linear inequalities:

$$pb_{i-1} + qb_i + rb_{i+1} \leq z \quad i = 1...N \tag{8}$$

Assuming that we have already found reliable coefficients for linear dependencies of parameters on b_i, the plan for increasing b_i while minimizing the functional (1) needs to satisfy the system (8) to make $\chi_i = 0 \ \forall i = 1...$. But if we on every step change the coefficients of dependency of c, n, L on b, then there can appear some misses. The optimality of modified least squares in the sense described in [16] allows us to state that the number of misses will be as low as it is possible. Also, the positive side of block size step-by-step increasing is that we perform estimation of the coefficients on the blocks from smallest to bigger, so while error of prediction tends to zero, block size grows.

So, the algorithm of block size change can be as follows.

Block Size Increase

- Depending on $s^{(m)}$, find minimal b_1 such that (5) is satisfied (probably with some reserve $r^{(m)}$)
- Repeat until the threshold of increasing is reached:
  - find b_i according to the system (8)

The algorithm is illustrated in Fig. 2.

The idea is to increase block size up to the value when desired communication rate does not allow to increase it more. This increase can be done in step-by-step manner, satisfying the conditions above. Desired communication rate, however, can be set manually or investigated via adaptive identification procedure. Then, the average error of prediction (3) can be a good criteria to do this identification.

If the error rate is high in average, we should make shorter blocks and send them to the target system more rapidly.

The last interval of execution according to (1) is $n_{N^{(m)}}$. It should be minimized. It implies that last block $b_{N^{(m)}}$ should be as small as $s^{(m)}$ allows. Then, in some moment the server should start to shorten the blocks. It should be done similar to **Block Size Increase**, but with decreasing instead of increasing. The same system (8) needs to be satisfied.

In general, the system (8) provides lower and upper bound for each of b_i. While increasing, we take the value closer to upper, while decreasing - value closer to lower bound is preferable.[2]

Next question arises: How to determine the moment to start the block size decrease procedure?

After first steps of increasing are done, we assume that we can predict the parameters good. Using them, we can find a solution of the system (8) with requirement of the block size reducing adding initial condition $b_{i_{thr}} = B$ and minimization criteria $\sum_{i=i_{thr}}^{N:b_N=b_{min}} b_i \rightarrow min$.

Then, the only unknowns are $b_{i_{thr}}$ and therefore step numbers i_thr and N. We propose to choose i_{thr} closer to the maximal possible with $\sum_{i=1}^{i_{thr}} b_i \rightarrow \max$ and $\sum_{i=i_{thr}}^{N:b_N=b_{min}} b_i \rightarrow min$, while letting the total work of client m to be $W^{(m)} = \sum_{i=1}^{N^{(m)}} b_i^{(m)} \quad \|W^{(k)} - W^{(m)}\| \rightarrow \min \quad \forall k \neq m$ for every pair of different clients k,m.[3]

The decreasing should start in a way that assumes entering the phase with minimal block size in nearly same moments for each target system.

Remember assumed "good" prediction capability of the parameters. In formal definition, it adds some upper bound on the increasing of block size. We should do some fixed number of recurrent estimation iterates to provide reliable results. According to the big numbers law, the expected error is proportional to the inverse of the number of iterates, with some constant which can be once found for typical case.

5.3 Final Planning

When the planning of block decrease for the most of client is done, it is possible to perform planning of the last blocks distribution. Important task of balancing is to make the server to get the results of last block execution from each client in nearly same time. The task of asynchronous task getting from server can be called "effective scheduling" or "effective execution". It gives reward in making the cost functional value lower by some multiplicative factor.

The planning of last block distribution can decrease cost functional value only on some additive term, which however can be significant. Without any planning

[2] Probably, the "closer" term should be defined more formally.

[3] The system to solve seems to be complicated; approximation methods can be used. We still assume server to be capable to deal with these computations. Approximation can be done introducing some "intensity" parameter for every target system and computing the predicted work for the target system using the intensity as a scale in division procedure.

average imbalance is a half of average execution time of the block on average system. If the block is like SETI@Home project work unit, which is computed during a day approximately [10], then this average imbalance is half of a day, and it seems to be significant value to perform planning.

The planning of execution when task execution time on different systems is known and the task set is given is a known and discussed problem [6]. The method from [6] can be used to do the planning. Here we formalize the problem of final planning as follows:

$$\sum_{m=1}^{M} n^{(m)} b^{(m)} - W \geq 0 \tag{9}$$

$$\sum_{m=1}^{M} n^{(m)} b^{(m)} - W \leq \min_{m}\{b^{(m)}\} \tag{10}$$

$$\max_{m}\{n^{(m)} t(b^{(m)}) + r^{(m)}\} \rightarrow \min \tag{11}$$

Here $r^{(m)}$ denotes the remaining task to do before starting the final execution. All the target systems have such remainder due to unsynchronized block execution.

5.4 Note on $s^{(m)}$

The desired communication rate $s^{(m)}$ is defined from client behavior. It should be equal to the lower bound of interval length in seconds such that if client-server interaction will be performed every time interval greater or equal than that, then it can be assumed that the client will answer server in most cases.

This parameter has importance for geographically distributed networks with different communication facilities. Initial values for it can be found in some static table, mapping client IP masks to possible desired communication rates. Then, this data can be tuned and tracked with the system/network changes.

In our formal model, however, there can be found some condition on $s^{(m)}$ derived from conditions (6): $s^{(m)} \geq a_{c^{(m)}} + a_{n^{(m)}}$ for $a_D : D(b) = a_D + k_D b$.

5.5 Discussion

Several issues are of interest why talking about heterogeneous distributed computing. Firstly, what if a target system wants to participate in the computations which already started? Our system provides the model of inclusion of the target system as follows: it starts from minimal possible block size and runs the increasing procedure together with identification of the parameters, maybe with smaller steps, up to reaching the decreasing threshold found satisfying the condition that decreasing threshold should lead to the nearly same time of entering the phase with lowest possible block size.

Another question is what to do if the system leaves the computation before finish. It seems that the most frightening is the case with biggest blocks in the middle of execution. When target system stops reacting the server tasks (after each block execution it should initiate the returning of results), it should be

removed from the execution and it's block should be rescheduled. However, the middle is not the end, and there is enough possibility to change the scheduling (i_{thr}) flexibly.

6 Conclusions

In this paper authors tried to present realistic model of SPMD computations in homogeneous input case. The adaptive procedure discussed is complicated and needs further explanations; however, it's features (on-line system identification, primarily, which is similar to *non-clairvoyance* in [5]) seem to be highly applicable to the field.

Authors participate in the Saint-Petersburg State University project about GRID computations, supported by Intel. Implementation of the proposed technique is one of the goals of the project. We find that reliable and robust implementation is as important as theoretical optimality results in distributed computing, where practice is very close to theory.

References

1. Granichin, O., Polyak, B.: Randomized algorithms of estimation and optimization under almost arbitrary noise. M. Nauka (2003)
2. Nakano, A.: High performance computing and simulations (Spring '07). Available online: `http://cacs.usc.edu/education/cs653.html`
3. Weissman, J.: Prophet: automated scheduling of SPMD programs in workstation networks. Concurrency: Practice and Experience 11(6), 301–321 (1999)
4. Cermele, M., Colajanni, M., Necci, G.: Dynamic load balancing of distributed SPMD computations with explicit message-passing. In: Proc. of the IEEE Workshop on Heterogeneous Computing, pp. 2–16 (1997)
5. He, Y., Hsu, W., Leiserson, C.: Provably efficient adaptive scheduling for parallel jobs. In: The Proc. of the 12th Workshop on Job Scheduling Strategies for Parallel Processing (2006)
6. Ichikawa, S., Yamashita, S.: Static load balancing of parallel PDE solver for distributed computing environment. In: Proc. ISCA 13th Int'l. Conf. Parallel and Distributed Computing Systems (PDCS-2000), pp. 399–405 (2000)
7. Casavant, T., Kuhl, J.: A taxonomy of scheduling in general-purpose distributed computing systems. IEEE Trans. on Software Engineering 14(2), 141–154 (1988)
8. Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. 35th Annual IEEE Conf. on Foundations of Computer Science (FOCS'94), Santa Fe, New Mexico. IEEE Computer Society Press, Los Alamitos (1994)
9. Neill, D., Wierman, A.: On the benefits of work stealing in shared memory multiprocessors. report, `http://www.cs.cmu.edu/acw/15740/paper.pdf`
10. Anderson, C., et al.: SETI@Home: an experiment in public-resource computing. Comm. of the ACM 45(11), 56–61 (2002)
11. Jaillet, C., Krajecki, M.: Constructing optimal Golomb rulers in parallel. In: Proc. 6th European Workshop on OpenMP, pp. 29–34 (2004)

12. Warren, M., Salmon, J.: A parallel hashed oct-tree n-body algorithm. Supercomputing, 12–21 (1993)
13. Lof, H.: Iterative and adaptive PDE solvers for shared memory architectures. Acta universitalis: digital comprehensive summaries of Uppsala dissertations from the faculty of science and technology (2006)
14. Hamidzadeh, B., Lilja, D.: Dynamic scheduling strategies for shared memory multiprocessors. In: International Conference on Distributed Computing Systems, pp. 208–215 (1996)
15. Autonomic computing: IBM's perspective on the state of information technology, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf
16. Kushner, H., Yin, G.: Stochastic approximation and recursive algorithms and applications, 2nd edn. Springer, Heidelberg (2003)
17. Estkover, C.A. (ed.): Chaos and Fractals: A Computer Graphical Journey, p. 468. Elsevier Science, Amsterdam (1998)