

# A Note on Using Partitioning Techniques to Solve Nonlinear Optimization Problems on Parallel Systems

M. Al-Baali<sup>a</sup> and C. Triki<sup>b,c</sup>

<sup>a</sup>Department of Mathematics and Statistics,  
Sultan Qaboos University, Muscat, Oman

<sup>b</sup>Department of Mechanical and Industrial Engineering,  
Sultan Qaboos University, Muscat, Oman

<sup>c</sup>Department of Engineering Innovation,  
University of Salento, Lecce, Italy

**Abstract.** In this paper, we deal with the design of parallel algorithms by using variable partitioning techniques to solve nonlinear optimization problem. We propose an iterative solution method that is very efficient for separable functions, and our scope is to discuss its performance for general functions. Experimental results on an illustrative example have suggested some useful modifications that, even though improve the efficiency of our parallel method, keep some open questions for further investigation.

## 1. Introduction

The main aim of this work is to discuss the efficiency of solving nonlinear optimization problems on parallel systems when using partitioning techniques. Consider the following unconstrained optimization problem

$$\min_{x \in R^n} f(x), \tag{1}$$

that we would like to solve on parallel systems not by, trivially, parallelizing a given sequential method, but rather by designing algorithms specifically intended for parallel computers. We want to show that even though such techniques are very successful for some problems there is still the risk that the convergence of the numerical method happens very slowly for other type of problems.

Our approach consists in developing an algorithm that solves nonlinear optimization problems by partitioning the optimization problem into a given number of subproblems, each of them is solved separately by a sequential method. We try to achieve the partitioning phase such a way that all processors are efficiently utilized in doing useful work during most of the execution time.

This strategy will be very fruitful when solving particular types of problems (e.g., separable functions) because, in this case, a solution of the original problem is obtained as soon as all subproblems are solved on the available processors of the parallel system. For general functions that are characterized by variables dependencies, however, convergence may be very slow vanishing, thus, any speedup deriving from the use of parallel technologies.

The solution of optimization problems on parallel systems or on grids has attracted over the last decades the attention of many researchers [1, 2, 3, 4]. If we ignore those contributions that simply used the automatic parallelization offered by the intelligent compilers embedded within the parallel systems then most of the works can be classified into two different types: fine and coarse grain parallelization. Fine grain parallelization acts directly at a matrix and vector levels to split the workload among the processors [5, 6]. Coarse grain parallelization tries to tackle the structure of the problem in order to split it among the available processors [7, 8, 9, 10]. This last direction was the one chosen by Ferris and Mangasarian whose seminal paper [11] proposes a general paradigm for solving nonlinear optimization problems based on splitting the decision variables among the processors. Their method, called Parallel Variable Distribution, involves, besides a concurrent phase, a synchronization phase based on calculating the affine hull of the generated partial solutions. Their method has proven theoretically to converge under a set of mathematical conditions and their preliminary experimental results have shown the potential for high parallelization efficiency of the method.

After the seminal paper of Ferris and Mangasarian many other methods have been proposed addressing the parallelization of the problem's structure [12, 13, 14, 15]. This paper goes in this same direction proposing a parallel algorithm that distributes the variables of a general problem among the available processors. Then, unlike Ferris and Mangasarian's approach that uses an affine hull for the synchronization step, our iterative method simply restarts, whenever is needed, the same method with a new initial point that became available on some of the processors while running the sequential method independently.

The paper is organized as follows. In Section 2 we describe our proposed parallel algorithm. In Section 3 we provide how to redefine the initial point for the parallel numerical algorithm. In Section 4 we apply the proposed parallel algorithm to a simple but challenging general function. Then we suggest, in Section 5, some modifications and we discuss the numerical results. Finally, we draw in Section 6 some concluding remarks.

## 2. Parallel Algorithm for Nonlinear Optimization

We are interested in solving iteratively unconstrained optimization problems, as defined in (1), by using a numerical method supposing that the initial point  $x^{(1)}$  is given.

For convenience, let the  $n$  variables of  $x$  be partitioned into groups that match the number,  $p$  say, of processors of a parallel system. specifically, let

$$x = (x^{[n_1]}, x^{[n_2]}, \dots, x^{[n_i]}, \dots, x^{[n_p]})$$

so that  $n_1 + n_2 + \dots + n_p = n$ .

In this case, the initial point  $x^{(1)}$  can be defined as:

$$x^{(1)} = (x^{[n_1,1]}, x^{[n_2,1]}, \dots, x^{[n_i,1]}, \dots, x^{[n_p,1]}). \quad (2)$$

The idea of designing a parallel algorithm consists in solving iteratively, using the available processors,  $p$  independent unconstrained subproblems starting from the initial point  $x^{(1)}$ . Then we check, at each iteration, the termination condition consisting in:

$$\|g(x^{(c)})\| \leq \epsilon, \quad (3)$$

where  $g(x)$  is the gradient of  $f(x)$ ,  $\epsilon$  is a tolerance parameter, and  $c$  is an iteration counter. If condition (3) is satisfied then the algorithm is terminated defining the solution of problem (1). Otherwise, the process is restarted, but with a new initial point  $x^{(c+1)}$  defined by using the solution obtained in iteration  $c$ , as will be described in Section 3. We refer in the sequel to this repetitive process as *outer* iteration of the parallel method and we use the index  $c$  for its identification, whereas the repetitive process inherent in the numerical method is referred to as *inner* iterations and denoted by the index  $k$ . Along the paper, we will specify explicitly the index we are referring to whenever this is not made evident from the context.

Consequently, at each iteration  $c$  we consider solving, on each processor  $p_i$  (with  $i = 1, 2, \dots, p$ ), by using a certain sequential method ([16, 17]) the following subproblem having  $n_i$  variables:

$$\min_{y \in R^{n_i}} f_i(y) = \min_{y \in R^{n_i}} f_i(\bar{x}^{[n_i,c]}). \quad (4)$$

where

$$\bar{x}^{[n_i,c]} = (x^{[n_1,c]}, x^{[n_2,c]}, \dots, x^{[n_{i-1},c]}, y, x^{[n_{i+1},c]}, \dots, x^{[n_p,c]}). \quad (5)$$

that is, all the components of  $\bar{x}$  are known quantities except the partition  $y$  that represents the only set of variables of  $p_i$ 's subproblem specified as

$$y = x^{[n_i]} = (x_{s+1}, x_{s+2}, \dots, x_{s+n_i})^T,$$

where  $n_0 = 0$  and  $s = n_0 + n_1 + n_2 + \dots + n_{i-1}$ .

The starting point for a generic iteration  $c$  is, thus, given by

$$y^{(c)} = x^{[n_i, c]} = (x_{s+1}^{(c)}, x_{s+2}^{(c)}, \dots, x_{s+n_i}^{(c)})^T.$$

After solving subproblem (4), we obtain on processor  $p_i$  the solution  $y^* = x^{[n_i, *]}$  that verifies  $f_i(y^*) = f_i(\bar{x}^{[n_i, *]})$  and by combining all the results deriving from all the processors a new estimate of problem's (1) solution is defined.

It is worth noting that this algorithm is very efficient if the variables assigned to each processor  $p_i$  are independent from the other variables, i.e the original problem can be partitioned into a set of independent subproblems. In this case, the solution of the original problem (1) is obtained as soon as all the processors complete the required tasks of solving the assigned subproblems. This happens, for example, in the case of separable problems in which it is possible to divide as equally as possible the variables among a certain number of processors so that each of them requires almost similar time for solving the assigned subproblem. Subsequently, by combining all the obtained partial solutions we get the desired solution of problem (1). In this case, the time required to solve (1) is equal to the time required to solve the most difficult subproblem plus some data communication overhead.

However, for general problems it is usual that the variables assigned to processor  $p_i$  depend on some other variables handled by another processor  $p_j$  with  $i \neq j$ . In this case the algorithm may converge, as shown in section 4, very slowly since it will be necessary to restart the algorithm by generating a new initial point at each (outer) iteration of our method. The way of generating a new initial solution is illustrated in the next section and some improvement techniques will be implemented in section 5.

### 3. Obtaining a New Starting Point

We assume here that, at a given iteration  $c$ , not all the processors terminated with a solution  $\bar{x}^{[n_i, c]}$  of the assigned subproblem  $i$  which, once combined with the other partial results, do not constitute a valid solution to the original problem (i.e. condition (3) is not satisfied). In this case, we may exploit any solution  $\bar{x}^{[n_j, c]}$  of another subproblem that is available on processor, say  $p_j$ , defined as in (5) except that  $i$  is replaced by  $j$ .

The idea is to call this value to processor  $p_i$  that will restart the process of solving problem (4) with  $x^{(c)}$  as defined above, except for  $\bar{x}^{[n_j, c]}$  which is replaced

by  $\bar{x}^{[n_j, *]}$ . In addition, in order to improve the initial solution quality it should be also advantageous replacing  $\bar{x}^{[n_i, c]}$  by  $\bar{x}^{[n_i, *]}$  whenever the following condition is verified:

$$f_i(x^{[n_j, c]}) < f_i(x^{[n_i, c]}).$$

#### 4. Illustrative Example

Consider solving the (general) well known Rosenbrock problem

$$\min_{x \in \mathbb{R}^2} f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 \quad (6)$$

starting with  $x^{(1)} = (0, 0)^T$ . It is easy to note that the optimal solution is  $x^* = (1, 1)^T$ .

Assume that two processors,  $p_1$  and  $p_2$ , are available. In this case  $n_1 = n_2 = 1$  and  $x^{(1)} = (x^{[1,1]}, x^{[2,1]})^T = (0, 0)^T$  but for simplicity of notation we will rename the two variables as  $x^{(1)} = (y^{(1)}, z^{(1)})^T = (0, 0)^T$ .

Consequently, subproblem (4) to be solved on  $p_1$  can be written as

$$\min_{y \in \mathbb{R}} f_1(y) = 100(z - y^2)^2 + (1 - y)^2, \quad (7)$$

where  $z = x^{[2,1]}$  is considered to be a known constant here (set initially to 0) and the starting point, for  $c = 1$ , is given by  $y^{(c)} = 0$ .

Similarly, processor  $p_2$  should solve the following subproblem

$$\min_{z \in \mathbb{R}} f_2(z) = 100(z - y^2)^2 + (1 - y)^2, \quad (8)$$

where  $y = x^{[1,1]}$  is a known constant (set initially to 0) and the initial point, for  $c = 1$ , is given by  $z^{(c)} = 0$ .

While, problem (8) can be solved on  $p_2$  exactly in one step yielding easily the solution  $z^* = y^2$ , we need to apply a numerical method to solve subproblem (7). In this paper we have implemented Newton-Raphson method that is known to be very efficient whenever the starting point is close enough to the optimum [18]. By using the index  $k$  that denotes the inner iterations of this iterative method then its termination criterion can be defined by

$$\|y^{(k)} - y^{(k+1)}\| < \epsilon, \quad (9)$$

where  $\epsilon = 10^{-6}$ .

The numerical results for solving  $f_1(y)$  on processor  $p_1$  are reported in Table 1. It includes for each inner iteration  $k$  the value of the variable  $y^{(k)}$ , its corresponding value  $f_1(y^{(k)})$  and the initial value of  $z^{(c)}$  corresponding to the (outer) iteration  $c$  (recall that  $z^{(c)}$  is constant for problem  $f_1$ ).

Table 1: Solution of problem (7) on  $p_1$  for outer iteration  $c = 1$ 

$k$	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	0	1	0
2	1	100	0
3	6.6722130E-01	19.929645	0
4	4.4688493E-01	4.294191	0
5	3.0373406E-01	1.3358733	0
6	2.1664151E-01	8.3392632E-01	0
7	1.7376816E-01	7.7383512E-01	0
8	1.6209454E-01	7.7112126E-01	0
9	1.6126604E-01	7.7110970E-01	0
10	1.6126202E-01	7.7110970E-01	0

From the results of Table 1 it is clear that the Newton-Raphson method has terminated because of the satisfaction of condition (9). However, the whole algorithm cannot be arrested because we checked that condition (3) is not satisfied yet. For this reason we have to restart a new outer iteration by using a new estimate of the initial solution  $x^{(2)} = (0.16126202, 0)^T$  as obtained from Table 1 (and from the exact solution of subproblem (8)). The results are summarized in Table 2 and even in this case condition (3) is not verified and the iterative process should, thus, continue by using  $x^{(3)} = (0.2113389, 0.02600544)^T$  as a new initial solution.

Table 2: Solution of problem (7) on  $p_1$  for outer iteration  $c = 2$ 

$k$	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	1.6126202E-01	7.0348137E-01	0
2	2.3482151E-01	6.7038702E-01	0
3	2.1393910E-01	6.5695530E-01	0
4	2.1137604E-01	6.5680104E-01	0
5	2.1133890E-01	6.5680104E-01	0

Before proceeding, we want to note that the total number of function evaluations, say  $NF$ , required to terminate Newton-Raphson method is 10 in Table 1 and 5 in Table 2. This concept of counting the number of function evaluations is

very important in the nonlinear optimization field. It represents, indeed, an alternative way (with respect to the CPU run time) of measuring the performance of a numerical algorithm without relating it to a specific hardware platform. For this reason, we will use this concept even for defining a termination condition of the whole parallel algorithm. We decide for simplicity, indeed, to stop our parallel algorithm on the basis of the following well known condition, instead of using criterion (3):

$$\|f_1(y^{(c)}) - f_1(y^{(c+1)})\| < \eta, \quad (10)$$

where  $c$  is the outer iterations index and  $\eta$  is chosen to be, for example,  $10^{-8}$ .

By using relation (10) in the solution of our illustrative example, a summary of the results is reported in Table 3. This table, besides reporting as before  $k$ ,  $y^{(k)}$ ,  $f_1(y^{(k)})$  and  $z^{(c)}$ , shows also the number of outer iterations  $c$  and the number of function evaluations  $NF$ . The first two lines of Table 3 can be easily derived from the results of Tables 1 and 2 and the value of  $x^{(3)}$  defined above can be seen in the line corresponding to  $c = 3$ .

Table 3. Outer and inner iterations for solving problem (7) on  $p_1$

$c$	$k$	$NF$	$y^{(c)}$	$f_1(y^{(c)})$	$z^{(c)}$
1	10	10	0.0000000E+00	1	0
2	5	15	1.6126202E-01	7.7110970E-01	0
3	4	19	2.1133890E-01	6.5680104E-01	2.6005440E-02
4	4	23	2.4508301E-01	5.9362041E-01	4.4664126E-02
5	4	27	2.7112272E-01	5.4933041E-01	6.0065686E-02
:	:	:	:	:	:
:	:	:	:	:	:
10	4	47	3.6653980E-01	4.0873867E-01	1.2571033E-01
11	3	50	3.7761509E-01	3.9415559E-01	1.3435143E-01
:	:	:	:	:	:
:	:	:	:	:	:
208	3	641	8.0786871E-01	3.7055832E-02	6.5146273E-01
209	2	643	8.0860185E-01	3.6773699E-02	6.5265184E-01
:	:	:	:	:	:
:	:	:	:	:	:
1892	2	4009	9.9800324E-01	3.9970463E-06	9.9600046E-01
1893	2	4011	9.9800825E-01	3.9770370E-06	9.9601048E-01

These results show that, even though our parallel algorithm succeeds to generate the solution  $x^* = (0.99800825, 0.99601048)$  which is very close to the op-

imum, its performance, expressed in terms of number of function evaluations, remain unsatisfactory. Another positive comment that can be drawn from the results is that as the number outer iterations increases the quality of the initial solution improves and, consequently, the number of inner iterations required to solve  $f_1$  decreases.

## 5. Modified Algorithm

The illustrative example of the previous section highlights the main drawback of solving nonseparable problems on parallel systems, namely the fact that some of the processors remain idle while others are doing useful work. In the previous example, indeed, processor  $p_1$  was heavily utilized whereas  $p_2$  solved its task in one step and remained idle most of the time. In order to remedy to this inconvenience, we propose a modification of our parallel algorithm consisting in calling any useful detail made available on any of the processors to update the initial solution and restart the process. We illustrate this idea by reconsidering the example of section 4.

Suppose that processor  $p_2$ , once idle, calls any detail made so far available on  $p_1$ , i.e.  $y^{(k=2)} = 1$  (see Table 1). Although  $f_1(y^{(k=2)}) = 100$  is much larger than  $f_1(y^{(k=1)}) = 1$ , this will result in getting  $z^{(2)} = 1 * 1 = 1$  on  $p_2$  and in surprisingly obtaining immediately the optimal solution of the original problem. Since this case may happen very rarely in practice, we propose another practical way of modifying our parallel method.

While solving subproblem (7) on  $p_1$ , we do not wait till the satisfaction of condition (9) but we stop the inner iterations earlier, i.e. whenever it happens that

$$f_1(y^{(k)}) < f_1(y^{(k=1)}). \quad (11)$$

and we pass this new generated value to the other processor to be used as a new initial value. The whole process then will terminate only when condition (10) is met. To understand how this modification will affect the solution of our example, consider the results of Table 1. By applying the new rule (11), processor  $p_1$  does not need to run 10 inner iterations anymore but will stop at iteration 6, i.e. as soon as  $f_1(y^{(k=6)}) = 0.83392632$  which results to be less than  $f_1(y^{(k=1)}) = 1$ , reducing, thus, the time in which  $p_2$  must remain idle. Hence, both the processors terminate with the point  $x^{(2)} = (0.21664151, 0)^T$ . Using this new estimate, we apply this modification in order to solve again problem (8) and then we continue till obtaining the final estimate of the solution of (8) which yields that of (6). The obtained results are summarized in Table 4.

By comparing these results to those reported in Table 3 we can note how this modification has reduced substantially the number of inner iterations  $k$  required at each outer iteration and, consequently, the number of function evaluations  $NF$  is also reduced from 4011 to 1894.

Table 4. Results of the modified algorithm for solving problem (7) on  $p_1$

$c$	$k$	$NF$	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
1	6	6	2.1664151E-01	8.3392632E-01	0
2	1	7	2.5625824E-01	5.8825087E-01	4.6933546E-02
3	1	8	2.8353419E-01	5.3500097E-01	6.5668290E-02
4	1	9	3.0514270E-01	4.9900761E-01	8.0391645E-02
:	:	:	:	:	:
:	:	:	:	:	:
10	1	14	3.8824373E-01	3.8099867E-01	1.4251556E-01
11	1	15	3.9822456E-01	3.6829531E-01	1.5073320E-01
:	:	:	:	:	:
:	:	:	:	:	:
208	1	212	8.1014949E-01	3.6180865E-02	6.5516895E-01
209	1	213	8.1086987E-01	3.5906568E-02	6.5634220E-01
:	:	:	:	:	:
:	:	:	:	:	:
1889	1	1893	9.9800360E-01	3.9956158E-06	9.9600118E-01
1890	1	1894	9.9800860E-01	3.9756095E-06	9.9601119E-01

Although this last modification has improved our parallel algorithm significantly, the high number of  $NF$  required to solve the problem still remains unsatisfactory. The modified method seems to converge, indeed, slowly since a large number  $NF$ , albeit smaller than before, is still required. A further reduction in the number of outer iterations (and thus in  $NF$ ) can be observed only when larger values of  $\eta$  are used. In this case, however, the substantial reduction of the values of  $c$  and  $NF$  (that can be observed in Table 5 for increasing values of  $\eta$ ) are paid by a deterioration in the quality of the obtained solution for problem (6). It is the responsibility of the decision maker to define the acceptable tradeoff between the solution quality and the method efficiency.

Table 5. Results of the modified algorithm for different values of  $\eta$ 

$\eta$	$c$	$k$	$NF$	$y^{(k)}$	$f_1(y^{(k)})$	$z^{(c)}$
$10^{-8}$	1890	1	1894	9.9800860E-01	3.9756095E-06	9.9601119E-01
$10^{-6}$	989	1	994	9.8044276E-01	3.8347995E-04	9.6116828E-01
$10^{-4}$	244	1	249	8.3378845E-01	2.7725833E-02	6.9420540E-01
$10^{-3}$	59	1	64	6.1549985E-01	1.4882825E-01	3.7569698E-01

## 6. Concluding Remarks

We have proposed in this paper a parallel algorithm based on the variable partitioning technique designed to solve efficiently separable problems, and we wanted to test its usefulness for general problems. We have shown that, despite the proof of convergence demonstrated in [11], such a convergence may happen very slowly vanishing probably the speedup deriving from the use of parallel systems. The main cause of this disadvantage is the fact that some of the processors remain idle while others are involved in carrying out useful computations. We have proposed some modifications that succeeded to reduce the effect of such inefficiency, but the results still remain unsatisfactory. The solution of a simple (but not separable) function has required a large number of function evaluations while it is known that the use of the sequential Newton Method will require only 3 function evaluations [16].

On the basis of these comments, the following conclusions are in order:

- while our proposed method is efficient for separable problems, it converges slowly for solving general functions;
- the illustrative example has been purposely chosen general, of small size and presenting the difficulty of having one of the two required processors that remain idle most of the computational time. Our intention was to perform a kind of worst-case analysis on difficult nonseparable functions;
- for well-structured general functions allowing a balanced distribution of the workload among the available processors, we expect better performance of our method. Moreover, tackling large scale problems will increase the workload and, thus, reduce the effect of unavoidable waiting time of idle processors;
- the improvements that can be achieved by our method are clearly problem dependent and the remarkable speedup obtained in solving our illustrative example may result to be more or less significant for other problems.

Thinking about more general improvement techniques tailored for nonseparable functions (such as considering load balancing techniques [19] or tasks scheduling [20]) may be of great benefit to speedup our parallel method.

Carrying out more intensive experimental experience and developing general purpose improvement techniques are beyond the scope of this note. This contribution wanted, from one side, to highlight the challenges to be faced when solving general problems by using partitioning techniques on parallel systems and, from the other side, to open the door in front of worldwide researchers with open questions in the field of parallel nonlinear optimization. This may be our objective in carrying out further investigations on this topic.

## References

- [1] R. B. Schnabel. A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Computing* 21(6), pp. 875-905, 1995
- [2] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: numerical methods*. Athena Scientific, Belmont-USA, 1997
- [3] C. Triki and L. Grandinetti. Computational grids to solve large scale optimization problems with uncertain data. *International Journal of Computing* 1(1), 2002
- [4] A. Migdalas, G. Toraldo and V. Kumar. Nonlinear optimization and parallel computing. *Parallel Computing* 29(4), Pp. 375-391, 2003
- [5] P. Beraldi, L. Grandinetti, R. Musmanno and C. Triki. Parallel algorithms to solve two-stage stochastic linear programs with robustness constraints. *Parallel Computing* 26(13-14), pp. 1889-1908, 2000
- [6] M. D'Apuzzo and M. Marino. Parallel computational issues of an interior point method for solving large bound-constrained quadratic programming problems. *Parallel Computing* 29(4), pp. 467-483, 2003
- [7] J. Blomvall. A multistage stochastic programming algorithm suitable for parallel computing. *Parallel Computing* 29(4), pp. 431-445, 2003
- [8] L. Giraud, A. Haidar and S. Pralet. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing* 36(5-6), pp. 285-296, 2010

- [9] F. Argello, D.B. Heras, M. Bo and J. Lamas-Rodriguez. The split-and-merge method in general purpose computation on GPUs. *Parallel Computing* 38(6-7), pp. 277-288, 2012
- [10] C. Triki. Solving the flood propagation problem with Newton algorithm on parallel systems. To appear on *SQU Journal for Science*
- [11] M. C. Ferris and O. L. Mangasarian. Parallel variable distribution. *SIAM Journal on Optimization*, 4, pp. 815-832, 1994
- [12] O. L. Mangasarian. Parallel gradient distribution in unconstrained optimization. *SIAM Journal on Optimization* 33, pp. 1916-1925, 1995
- [13] D. Rotiroti, C. Triki and L. Grandinetti. Combined MPI/OpenMP implementations for a stochastic programming solver. In *Parallel Computing Advances and Current Issues* (Edited by G. Joubert, A. Murli, F. Peters and M. Vanneschi), Imperial College Press, 2002
- [14] D. Zeng. Parallel iterative methods for nonlinear programming problems. *Advanced Materials Research* 159, pp. 105-110, 2010
- [15] W. Li. A parallel multi-start search algorithm for dynamic traveling salesman problem. *Lecture Notes in Computer Science* 6630, pp. 65-75, 2011
- [16] R. Fletcher. *Practical methods of optimization*. John Wiley & Sons, New York, 1987
- [17] M. Al-Baali and H. Khalfan. An overview of some practical quasi-Newton methods for unconstrained optimization. *SQU Journal for Science*, 12(2), pp. 199-209, 2007
- [18] A. P. Ruszczyński. *Nonlinear Optimization*. Princeton University Press, New Jersey, 2006
- [19] A. Grama and V. Kumar. Load balancing for parallel optimization techniques. *Encyclopedia of Optimization*, pp. 1905-1911, 2008
- [20] O. Sinnen. *Task scheduling for parallel systems*. Wiley-Interscience, New Jersey, 2007