INFORMATICA, 1998, Vol. 9, No. 4, 437–448

Parallel Implementation of a Generalized Conjugate Gradient Algorithm

Jonas KOKO, Aziz MOUKRIM

ISIMA, Université Clermont–Ferrand II
Campus des Cézeaux – BP 125
63173 Aubière cedex, France.
e-mail: koko@sp.isima.fr

Received: February 1998

Abstract. This paper presents a parallel version of a Generalized Conjugate Gradient algorithm proposed by Liu and Story in which the search direction considers the effect of the inexact line search. We describe the implementation of this algorithm on a parallel architecture and analyze the related speedup ratios. Numerical results are given for a shared memory computer (Cray C92).

Key words: parallel algorithms, unconstrained high-dimensional optimization, conjugate gradient methods, parallelism, shared memory computer.

1. Introduction

Conjugate gradient methods have been used successfully for the solution of unconstrained high–dimensional minimization problems. When exact line searches are used, conjugate gradient methods give finite termination for quadratic problems. Since the exact line search can cause excessive computational effort, conjugate gradient algorithms are generally used with an inexact line search, Fletcher and Reeves (1964), Polak and Ribière (1969), Shanno (1985), Touati–Ahmed and Storey (1990).

Liu and Storey (1991) proposed a new conjugate gradient algorithm in which the search direction considers the effect of the inexact line search. They proved that the method is globally convergent when the function to be minimized is twice continuously differentiable with a bounded level set.

In this paper we describe parallel versions of two conjugate gradient algorithms, LSA and LSH, proposed by Liu and Story. Whereas algorithm LSH is based on the computation of the Hessian matrix at each iteration, algorithm LSA uses some form of finite difference approximation in order to avoid the computation and the storage of the Hessian matrix. Numerical results for an implementation on a shared memory computer (Cray C92) are given. Then, we evaluate the advantages of parallel versions by computing the related speedup ratios.

2. Generalized Conjugate Gradient

We are concerned with unconstrained minimization problem

$$(\mathcal{P}) \quad \min_{x\in\mathbb{R}^n} f(x)$$

for a twice continuously differentiable function f with a bounded level set.

Several algorithms for solving $(\mathcal{P})$ use the concept of conjugacy. The classical conjugate gradient algorithms aim is to solve $(\mathcal{P})$ by a sequence of line searches

$$x_{k+1} = x_k + t_k d_k, \quad k = 1, 2, \ldots,$$

where t_k is the step length and d_k, the search direction, is of the form

$$d_k = -g_k + \beta_k d_{k-1},$$

with $g_k = \nabla f(x_k)$. There are various formulas for computing the coefficient β_k; see, e.g., Fletcher and Reeves (1964), Polak and Ribière (1969) or Gilbert and Nocedal (1992).

Liu and Storey (1991) proposed a new conjugate gradient algorithm in which the search direction d_k takes into account the effect of the inexact line search. The search direction of the conjugate gradient algorithm of Liu and Storey is of the form

$$d_k = -\alpha_k g_k + \beta_k\, d_{k-1}, \quad \alpha_k > 0.$$

Coefficients α_k and β_k are given by

$$\alpha_k = \left[v_k(g_k^T g_k) - w_k(g_k^T d_{k-1})\right]/D_k, \tag{1}$$

$$\beta_k = \left[w_k(g_k^T g_k) - u_k(g_k^T d_{k-1})\right]/D_k, \tag{2}$$

$$D_k = u_k v_k - (w_k)^2, \tag{3}$$

where

$$u_k = g_k^T H_k g_k, \quad v_k = d_{k-1}^T H_k d_{k-1}, \quad w_k = g_k^T H_k d_{k-1}. \tag{4}$$

The formal description of the conjugate gradient algorithm LSH of Liu and Storey (1991) which uses the computation of the Hessian matrix is as follows.

Algorithm LSH

Step 1. $k \leftarrow 0$, $d_0 = \nabla f(x_0)$.

Step 2. Line search

Compute the steplength t_k and set $x_{k+1} = x_k + t_k\, d_k$,

$k \leftarrow k + 1$.

Step 3. If $||g_k|| < \varepsilon$ then STOP.

Step 4. Compute u_k, v_k and w_k with (4).

Step 5. If $u_k > 0$, $v_k > 0$, $1 - w_k^2/(u_k v_k) \geqslant 1/(4R)$ and
$(u_k/||g_k||^2)/(v_k/||d_{k-1}||^2) \leqslant R$, $R > 0$, then
Compute α_k, β_k with (1)–(3).
$d_k = -\alpha_k g_k + \beta_k\, d_{k-1}$. Go to Step 2
else $x_0 \leftarrow x_k$. Go to Step 1.

In order to avoid the computation of matrices, Liu and Storey (1991) proposed computing u_k, v_k and w_k in Step 4 of algorithm LSH using finite–difference approximation. We call this version algorithm LSA. It is only different from LSH in Step 4 which is changed as follows.

$$u_k = g_k^T\big(\nabla f(x_k + \gamma_k g_k) - g_k\big)/\gamma_k, \tag{5}$$

$$v_k = d_{k-1}^T\big(\nabla f(x_k + \delta_k d_{k-1}) - g_k\big)/\delta_k, \tag{6}$$

$$w_k = g_k^T\big(\nabla f(x_k + \delta_k d_{k-1}) - g_k\big)/\delta_k, \tag{7}$$

where δ_k and γ_k are suitable small positive numbers. This requires the storage of 5 vectors of length n.

Let ϕ be the real–valued function given by

$$\phi(t) = f(x_k + t d_k), \quad t > 0.$$

Then the first order derivative of ϕ is given by

$$\phi'(t) = \nabla f(x_k + t d_k)^T d_k.$$

The conjugate gradient algorithm of Liu and Storey converges under the line search conditions

$$\phi(t_k) - \phi(0) \leqslant \sigma_1 t_k \nabla f(x_k)^T d_k, \quad 0 < \sigma_1 < 1/2, \tag{8}$$

$$|\phi'(t_k)| \leqslant -\sigma_2 \nabla f(x_k)^T d_k, \quad 0 < \sigma_1 < \sigma_2 < 1. \tag{9}$$

The step length t_k is determined by a line search algorithm. The formal description of a line search algorithm is as follows, Lemarechal (1981).

1. Initialization: $t_l \leftarrow 0$, $t_r \leftarrow +\infty$.
   Choose $t > 0$.
2. Compute $\phi(t)$ and $\phi'(t)$
   If $\phi(t) - \phi(0) \leqslant \sigma_1\, t\, \phi'(0)$ then
   If $|\phi'(t)| \leqslant -\sigma_2 \phi'(0)$ then $t_k \leftarrow t$, STOP.
   Else
   t is too small, $t_l \leftarrow t$.

Compute a new t by interpolation over $(t_l,\ 10t_l)$.

Else

t is too large, $t_r \leftarrow t$.

Compute a new t by interpolation over $(t_l,\ t_r)$.

3. Go to 2.

3. Parallel Implementations

Many studies have been devoted to the construction of unconstrained optimization algorithms for minimizing differentiable nonlinear functions on computers with parallel processors. The main idea in these algorithms is to evaluate simultaneously the function and its gradient at different points. This strategy is well suited for a SIMD (single instruction-multiple data stream) multiprocessor. In the well known Newton or quasi–Newton algorithms, the search direction d_k at the $k\text{th}$ iteration is constructed through the following relation

$$d_k = -H_k \cdot g_k,$$

where H_k is computed using

(i) parallel finite–difference approximation in the case of Newton algorithms Lootsma and Ragsdell (1988), Conforti and Musmanno (1993);

(ii) parallel updating formulas in the quasi–Newton case Laarhoven (1985), Schnabel (1987).

In both cases, matrix H_k must be stored. Consequently, the use of these methods is limited to middle–size problems, due to storage requirements. In this section, we propose parallel versions of Liu and Storey algorithms LSH and LSA.

3.1. *Algorithm PLSH*

The parallelization of algorithm LSH is based on the parallel finite–difference approximation of Hessian H_k in order to compute u_k, v_k and w_k defined in (4), without storage of matrices. In algorithm LSH, we need matrix H_k only for the computation of u_k, v_k and w_k through the calculation of some vector y_k in the form

$$y_k = H_k \cdot b_k,$$

where b_k is g_k or d_{k-1}. The best opportunity for parallelism of algorithm LSH is the evaluation of components of matrix H_k in parallel, and then the computation of components of vector $H_k b_k$ in parallel too. The number of available processors, denoted by p, is limited. So a suitable decomposition of the algorithm into tasks must be done. The algorithm LSH can be viewed as composed of p tasks $T_1, ..., T_p$. Each task T_l corresponds to the processing of about n/p rows, denoted by $Rows(T_l)$. To each processor π_l, we assign a task T_l. In order to avoid the computation of H_k two times, $H_k \cdot g_k$ and $H_k \cdot d_{k-1}$ are

done simultaneously. Note that for balancing the processing load between processors, if a processor treats a row i, it also treats row $n - i$ since generally $n \gg p$. In addition, with the scheme described below, we do not need the storage of the Hessian matrix for the computation of the symmetric part.

The component $H_k[i, j]$ is computed by using the following central finite–difference formulas. If $i \neq j$, then

$$H_k[i,j] = \big(f(x_k + \delta_i e^i + \delta_j e^j) - f(x_k + \delta_i e^i) - f(x_k + \delta_j e^j) + f(x_k)\big) \;/\; (\delta_i \delta_j), \tag{10}$$

and,

$$H_k[i,i] = \big(f(x_k + \delta_i e^i) - 2f(x_k) + f(x_k - \delta_i e^i)\big) \;/\; \delta_i^2, \tag{11}$$

where e^i is the ith unit vector and δ_i a suitable, small positive number.

Since the different tasks could update concurrently vector y_k, each processor π_l stocks the part of y_k it computes in a local vector y_k^l. At the end, the different local vectors are summed to obtain $y_k = H_k \cdot b_k$.

Task T_l: For each row i of matrix H_k in $Rows(T_l)$ do
  For each j in $1, ..., i-1$ do
    Let h be the component $H_k[i, j]$ computed as in (10)
    $y_k^l[j] = y_k^l[j] + h \cdot b_k[i]$
    $y_k^l[i] = y_k^l[i] + h \cdot b_k[j]$
  Let h be the component $H_k[i, i]$ computed as in (11)
  $y_k^l[i] = y_k^l[i] + h \cdot b_k[i]$

3.2. *Algorithm PLSA*

For LSA, we propose a parallel version which is similar to PLSH. The central problem is the computation of some inner products

$$s = x^T y$$

at each iteration k to obtain u_k, v_k or w_k. An inner product s can be decomposed as follows

$$s = \sum_{l=0}^{p-1} s_l,$$

where

$$s_l = \sum_{l\frac{n}{p}}^{(l+1)\frac{n}{p}} x[i] * y[i].$$

So, each processor π_l computes a partial inner product s_l. At the completion of the computation of these different s_l, $(l \in 0, ..., p-1)$, they are summed to obtain s. The instantiation of vector y during the computation of u_k, v_k or w_k suppose the computation of some gradient $\nabla f(x)$ which we determine by using the central finite-difference approximation:

$$\nabla f(x)[i] = \big(f(x + \delta_i e_i) - f(x - \delta_i e_i)\big) \ / \ (2\delta_i).$$

3.3. *Parallel Line Search*

For the line search algorithm, only the computation of $\phi'(t)$, in Step 2, is parallelized. The interpolation routine used for computing the step length t_k reduces to one scalar operation. The parallelization of the computation of $\phi'(t)$ is the same as the one used in §3.2 for computing u_k, v_k and w_k given by (5)–(7).

4. Experimental Results

Computational experiments have been carried out on a shared memory parallel computer Cray C92. The test problems are given in the appendix.

The number of variables varies according to the following values: $128, 256, 800, 1000$.

We have used the line search algorithm given by Gilbert and Lemaréchal (1989), with the initial step length taken to be

$$t_0 = 2\min\{1, (f(x_k) - \tilde{f})/g_k^T d_k\},$$

where $\tilde{f}$ is the estimate of the optimal function value. For all test problems, we set $\tilde{f} = 0$. The line search parameters , in (8)–(9), are $\sigma_1 = 0.0001$ and $\sigma_2 = 0.1$. In all cases, the stopping condition is

$$\| \ g_k \ \| < 10^{-5} \ \mathrm{Max}\{1, \| \ x_k \ \|\}.$$

The global convergence is ensured by setting $R = 10^{10}$.

In order to compare the performances of the two versions PLSH and PLSA, we use the speedup Γ_p defined as $\Gamma_p = C_1/C_p$ where p is the number of processors and C_i the execution time required on i processors. For both algorithms PLSH and PLSA, we present in Table 1 the number of iterations NI, the number of restarts NR and central processor unit CPU with 16 processors.

Numerical experiments of parallel algorithms are satisfactory from the convergence point of view. Both algorithms PLSH and PLSA always find a minimum. As can be seen from Table 1, in terms of CPU time PLSA is very good compared with PLSH even for cases where number of iterations in PLSA is greater than number of iterations in PLSH (problems 5, 8–10). The given approximations in (5), (6) and (7) are suitable to all test problems.

Table 1
Performances of algorithms PLSH and PLSA with 16 processors

Problems		PLSH			PLSA		
No	n	NI	NR	CPU	NI	NR	CPU
1	800	10	0	23.94	7	0	0.13
1	1000	11	0	19.10	10	2	0.17
2	800	10	2	65.77	9	2	0.59
2	1000	14	1	137.48	14	0	1.18
3	800	32	0	378.37	14	0	1.59
3	1000	15	0	229.82	14	0	1.44
4	800	14	0	25.91	14	0	0.26
4	1000	14	0	39.38	14	0	0.31
5	800	56	16	59.99	65	19	0.78
5	1000	52	13	80.71	54	8	0.79
6	800	18	5	19.20	14	3	0.20
6	1000	15	7	21.88	13	5	0.25
7	800	24	1	269.19	14	0	0.26
7	1000	52	0	79.14	14	0	0.31
8	800	28	1	22.13	41	3	0.34
8	1000	35	2	43.94	82	1	0.80
9	800	280	0	322.57	309	5	2.99
9	1000	289	0	529.26	365	7	4.44
10	800	21	0	22.19	39	0	0.39
10	1000	65	0	380.21	21	0	0.26

Due to round-off errors on finite differences approximation, the CPU time for an iteration for a dimension n can be lower than the one required for a dimension $\bar{n} > n$.

Generally the CPU time is an increasing function of the dimension n. However for certain problems, this is not the case as in problems 1 and 7 (for PLSH algorithm). This due to round-off errors introduced by the finite difference approximation of the Hessian matrix. Note that this is never the case for PLSA algorithm (among the 10 test problems).

In Figures 1 and 2, we present the average speedup for PLSH and PLSA related to test problems. The behavior of algorithms PLSH and PLSA is the same for all test problems. The speedup increases in terms of the processors number and vector length. It is noticeable that when the dimension n is small, it is useless to consider many processors. For example, problems with dimension $n = 128$ reach the maximal speedup with $p = 8$ in PLSH as in PLSA. This is was expected since task T_l has been constructed such that it sequentially treats $2 \times 7 = 14$ rows in order to reduce the number of memory locks. Although, problems in which the dimension n is sufficiently large, the behavior of the speedup is almost linear in terms of the processor number.

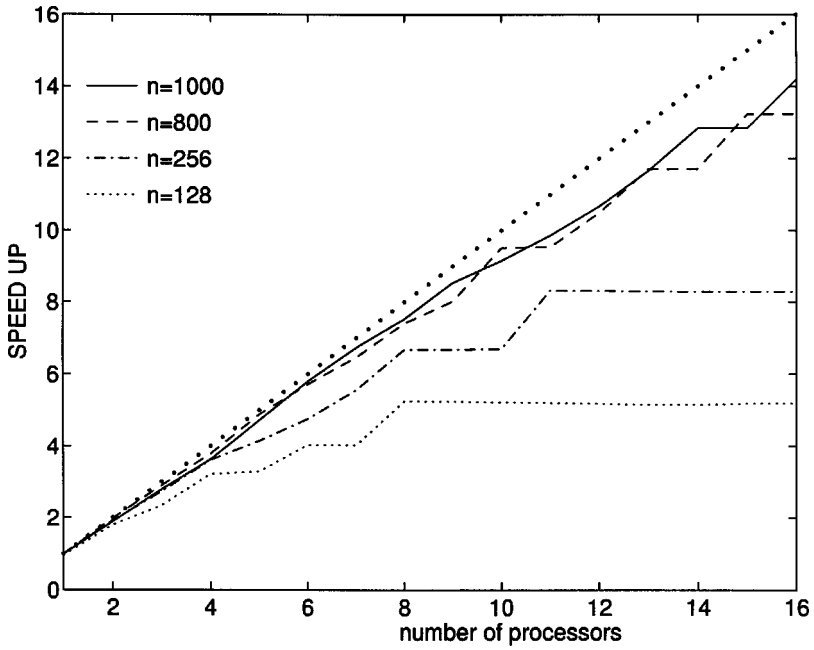

Fig. 1. Average speedup in terms of n for algorithm PLSA.

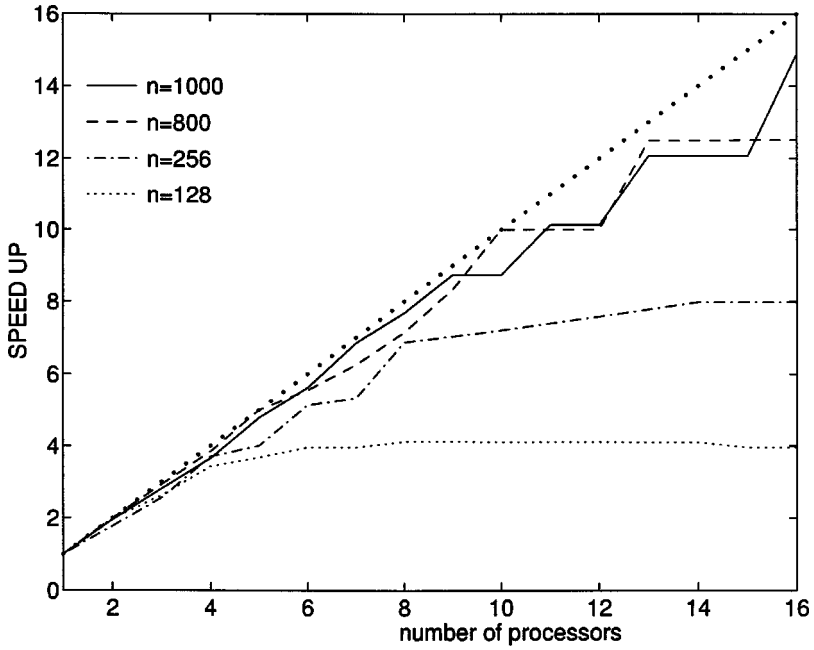

Fig. 2. Average speedup in terms of n for algorithm PLSH.

5. Conclusion

Liu and Storey have proposed a new Conjugate Gradient algorithm in which the search direction considers the effect of inexact line search. We have proposed a parallel version which is based on the parallel finite–difference approximation of some gradient and Hessian matrix without storage. Computational experiments have been carried out on a shared memory parallel computer Cray C92. We have noticed that problems in which the dimension is sufficiently large, the behavior of the speedup is almost linear in numerical experiments have also shown that PLSA algorithm is less sensitive to round-off errors that PSLH algorithm.

Appendix: Test Problems

Problem 1: Extended Beale function

$$f(x) = \sum_{i=1}^{n/2} \Big[\big(1.5 - x_{2i-1}(1 - x_{2i}^3)\big)^2 + \big(2.25 - x_{2i-1}(1 - x_{2i}^2)\big)^2 + \big(2.625 - x_{2i-1}(1 - x_{2i}^3)\big)^2 \Big], \quad n = 2, 4, 6, \ldots$$

with $x^0 = (1, 1, \ldots, 1)^T$.

Problem 2: Brown function.

$$f(x) = \left[\sum_{i=1}^{n/2} (x_{2i-1} - 3) \right]^2 + 0.0001 \sum_{i=1}^{n/2} \Big[(x_{2i-1} - 3)^2 - (x_{2i-1} - x_{2i}) + \exp\big(20(x_{2i-1} - x_{2i})\big) \Big], \quad n = 2, 4, 6, \ldots$$

with $x^0 = (0, -1, 0, -1, \ldots, 0, -1)^T$.

Problem 3: Cragg and Levy function

$$f(x) = \sum_{i \in J} \left[(e^{x_i} - x_{i+1})^4 + 100(x_{i+1} - x_{i+2})^6 + \tan^4(x_{i+2} - x_{i+3}) + x_i^8 + (x_{i+3} - 1)^2 \right],$$

where n is a multiple of 4 and $J = \{1, 5, 9, \ldots, n-3\}$, $x^0 = (1, 2, 2, \ldots, 2)$.

Problem 4: Generalized tridiagonal function

$$f(x) = 1 + \sum_{1}^{n} \big| \, (3 - 2x_i)x_i - x_{i-1} - x_{i+1} + 1 \, \big|^{7/3},$$

with $x_0 = x_{n+1} = 0$ and $x^0 = (-1, -1, \ldots, -1)$.

Problem 5: Penalty 1 function

$$f(x) = 10^{-5} \sum_{i=1}^{n} (x_i - 1)^2 + \left(\sum_{i=1}^{n} x_i^2 - 0.25 \right)^2, \quad n = 1, 2, \ldots$$

with $x_i^0 = i, \quad i = 1, 2, \ldots, n$.

Problem 6: Penalty 2 function

$$f(x) = \sum_{i=1}^{n} (x_i - 1)^2 + 10^{-3} \left(\sum_{i=1}^{n} x_i^2 - 0.25 \right)^2, \quad n = 1, 2, \ldots$$

with $x_i^0 = i, \quad i = 1, 2, \ldots, n$.

Problem 7: Extended Powell function

$$\begin{aligned} f(x) = \sum_{i=1}^{n/4} \Big[& (x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} - x_{4i})^2 \\ & + (x_{4i-2} - 2x_{4i-1})^4 + 10(x_{4i-3} - x_{4i})^4 \Big], \quad n = 4, 8, \ldots \end{aligned}$$

with $x^0 = (3, -1, 0, 3, 3, -1, 0, 3, \ldots, 3, -1, 0, 3)^T$.

Problem 8: Extended Rosenbrock function

$$f(x) = \sum_{i=1}^{n/2} \big[100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2 \big], \quad n = 2, 4, 6, \ldots$$

with

$$\begin{cases} x_{2i}^0 = 1.0 \\ \\ x_{2i-1}^0 = -1.2 + 0.4i/n \end{cases} \qquad i = 1, 2, \ldots, n/2.$$

Problem 9: Tridiagonal function

$$f(x) = \sum_{i=2}^{n} \Big[i \big(2x_i - x_{i-1} \big)^2 \Big]$$

with $x^0 = (1, 1, \ldots, 1)^T$.

Problem 10: Extended Wood function

$$f(x) = \sum_{i=1}^{n/4} \Big[100(x_{4i-2} - x_{4i-3}^2)^2 + (1 - x_{4i-3})^2 + 90(x_{4i} - x_{4i-1}^2)^2 + (1 - x_{4i-1})^2 + 10(x_{4i-2} + x_{4i} - 2)^2 + 0.1(x_{4i-2} - x_{4i})^2 \Big], \quad n = 4, 8, \ldots$$

with $x^0 = (-3, -1, -3, -1, \ldots, -3, -1)^T$.

References

Conforti, A., and R. Musmanno (1993). A parallel asynchronous Newton algorithm for unconstrained optimization. *Journal of Optimization Theory and Applications*, **77**(2), 305–323.

Dayde, M. (1989). Parallel algorithms for nonlinear programming problems. *Journal of Optimization Theory and Applications*, **61**(1), 23–46.

Fletcher, R., and C.M.Reeves (1964). Function minimization by conjugate gradients. *Computer Journal*, **7**, 149–154.

Gilbert, J.C., and C. Lemarechal (1989). The modules M1QN2, N1QN2, M1QN3 and N1QN3. *INRIA Technical Report*, Rocquencourt. 12pp. (in French).

Gilbert, J.C., and J. Nocedal (1992). Global properties of conjugate gradient methods for optimization. *SIAM Journal on Optimization*, **2**(1), 21–42.

Laarhoven, P.J.M. (1985). Parallel variable metric algorithms for unconstrained optimization. *Mathematical Programming*, **33**, 68–81.

Lemaréchal, C. (1980). A view of line searches. In A. Auslender, W. Oettli and J. Stoer (Eds.), *Lecture Notes in Control and Information Sciences*, Vol. 30, Springer–Verlag. pp 59–78.

Liu, Y., and C. Storey (1991). Efficient generalized conjugate gradient algorithms, part 1 : theory. *Journal of Optimization Theory and Applications*, **69**(1), 129–137.

Lootsma, F.A., and K.M. Ragsdell (1988). State of the art in parallel nonlinear optimization. *Parallel Computing*, **6**, 133–155.

Minoux, M. (1981). *Programmation Mathématique: Tome 1.* Dunod, Paris (in French).

Polak, E., and G. Ribière (1969). Note sur la convergence des méthodes de directions conjuguées, *RAIRO–RO*, **16**, 35–43.

Schnabel, R.B. (1987). Concurrent function evaluations in local and global optimization. *Computer Methods in Applied Mechanics and Engineering*, **64**, 537–552.

Shanno, D.F. (1985). Globally convergent conjugate gradient algorithms. *Mathematical Programming*, **33**, 61–67.

Touati-Ahmed, D., and Storey C. (1990). Efficient hybrid conjugate gradient techniques. *Journal of Optimization Theory and Applications*, **64**(2), 379–397.

J. Koko is Assistant Professor in Applied Mathematics at ISIMA, University of Clermont-Ferrand II (France). His research interests are conjugate gradient methods, augemnted larangian, nonlinear orthotropic elasticity.

A. Moukrim received his PhD in computer science from University of Clermont-Ferrand 2 in 1995. He is currently with LIMOS, University of Clermont-Ferrand 2. His research interests include parallel and distributed processing with a focus on multiprocessing scheduling and interconnection networks.

Lygiagreti apibendrinto jungtinių gradientų algoritmo realizacija

Jonas KOKO, Aziz MOUKRIM

Aprašyta apibendrintų jungtinių gradientų algoritmo, pasiūlyto Liu ir Storey, lygiagreti versija. Šiame algoritme nusileidimo kryptis parenkama atsižvelgiant į tai, kad vienmatė paieška nebus tiksli. Aprašyta algoritmo realizacija, skirta lygiagrečiai Cray tipo architektūrai. Analizuotas pasiekiamas pagreitinimas, pateikti eksperimentų rezultatai.