Control

and Cybernetics
VOL. 18 (1989) No. 1

Parallel optimization

by

M. BERTOCCHI

Dipartimento di Matematica, Statistica,
Informatica e Applicazioni
Istituto Universitario di Bergamo
Via Salvecchio 19, 24100 Bergamo
24100 Bergamo, ITALY

This work summarizes results and experiences on production of parallel programs for selected optimization algorithms using parallel facilities guaranteed by some available supercomputers.

KEYWORDS: Parallel algorithms, global optimization, local optimization.

1. Introduction

The interest of scientists for supercomputers has grown in the last years as their availability has become a real fact: on the market there is now a great range of such machines with various form of parallelism.

The advent of these machines, especially the vector ones, has had a great influence on scientific and engineering computation [21]. Certainly most of the work has been done in the field of linear algebra because of its extensive use in so many numerical areas.

The field which we are particularly interested in is the one of nonlinear optimization because the real application that need to use the non/linear optimization algorithms usually prefer to avoid them in view of the high computation time often involved.

The paper summarizes experiences we made in parallel optimization in the last years. We found particularly interesting some stochastic algorithms for global optimization [6, 7, 8, 9] and recent algorithms proposed by Dembo [16] and Han [24] in local optimization.

2. The computing environment

The machines which we used in our experiments are the Cray X-MP, the IBM 4381-3 with attached FPS processors (APs), the CRAY X-MP48 and IBM 3090 VF.

The first mentioned IBM machine was a prototype very useful to test the first approach to parallel problems. In fact it had a parallel environment both in hardware (10 array processors FPS 164) and in software (VM/EPEX and APEX/SUM) [18].

The parallelism in this machine could be seen at two levels:
— the first characterized by the possibility to execute in a very fast way operations involving vectors and matrices: this was carried out by the array processors.
— the other characterized by having on the host many virtual machines each connected to one AP, communicating with each other through a shared memory. In such a way there was an opportunity to have many tasks running simultaneously on different APs.

Tools available for implementing the parallel code were a Fortran 77 language, directives to describe parallelism in the code and precompilers, compilers, as well as the linker to make the code ready for the execution. One of the drawbacks was that communication among the APs and the host could be carried out only through an I/0 channel.

The CRAY X-MP machine is characterized by twelve functional units for different operations; some of them, the ones dedicated to vector operations, are supplied by vector registers. Two important hardware characteristics are chaining and overlapping: chaining allows results going out from a vector unit to come in to an other vector unit without the use of registers; overlapping allows the simultaneous use of different units with different data. The most important tool available is a Fortran compiler able to suggest which part of the code can be vectorized. There was a package [15] available on this machine for simulation of multitasking.

CRAY X-MP48 has four CPUs, each one equipped with the vector facilities described above, sharing a common memory. An interesting tool available on this machine to explore parallelism is microtasking [14, 15].

It is well known that vector facility speeds up all the computations involving operations between vector and matrices; besides that microtasking allows multiple processors to work at the DO-loop level where the granularity of the tasks can be small.

In a multiuser environment microtasking takes advantage of the fact that the number of processors available during program's execution may vary; in fact, because the overhead of synchronization is small enough the microtasked job can be dynamically adjusted to the number af available processors.

For this reason microtasking is an interesting tool to use because the user is not requested to know in advance the number of processors he will use and in the same time there will be the possibility to use the maximum computational power available during running time.

The IBM 3090 VF is a powerful machine with vector facilities and a memory hierarchy which allows to speed up the computations. On this machine, using the vector facility, it is possible to do operations with vector more than three times faster with respect to the scalar mode.

One of the most important feature is the compiler that is able to recognize automatically which DO-loops are vectorizable and which are not. There is no chaining as in the CRAY but there are compound operations, like multiply and add or multiply and subtract, which allow fast execution of these operations [12, 13, 22].

The IBM 3090 may be configured in such a way as to present up to six processors sharing a common memory each equipped with a vector facility.

3. First experiences in parallel optimization

First experiences have been carried out using the CRAY X-MP and the IBM 4381 with FPSs on the following algorithms:
— the modified Newton algorithm for local minimization;
— the Price and Boender algorithm for global optimization.

The algorithm for local minimization is a classical and a well tested on. It is based on the idea that a function can be approximated at each iteration by a quadratic model and, derived from the first order condition, the following system of equations must be satisfied:

$$H(x)p = -\nabla f(x)$$

where $H(x)$ is the Hessian and ∇f is the gradient of the function to be minimized. The steps of the algorithm are as follows:

```
STEP 1. $k = 1$; evaluation of $f(x_k)$, $\nabla f(x_k)$, $H(x_k)$;
STEP 2. Stop if $\|\nabla f(x_k)\| < \delta$, where $\delta$ is the user defined accuracy;
STEP 3. Solve the system $H(x_k)p_k = -\nabla f(x_k)$ with appropriate modification of
        the matrix $H$ in case of singularity;
STEP 4. Set $x_{k+1} = x_k + \alpha_k p_k$;
STEP 5. Compute $\alpha_k$ such that $\Phi(x_{k+1}) < \Phi(x_k)$ where $\Phi(x_{k+1}) = f(x_k + \alpha_k p_k)$
STEP 6. $k = k + 1$, go to step 1.
```

Because the algorithm involves many operations with matrices and vectors we thouhgt it was good for a vector machine. For this reason we ran it on the CRAY machine but we got very poor results as compared to the scalar version [5]. The negative result has quite a simple explanation if one carefully examines the code. The code contains in fact few own subroutines, the significant part of the run times comes from Hatfield routines (linear serch and Cholesky decomposition) which, after a restructuring action, contain no DOs at all.

The two algoritms for global optimization are based on two main steps:

— Choice of the sample —

N points are chosen randomly in the region where the function has to be minimized and the function must be evaluated in such points.

— Search of a better sample —
New points are looked for with the goal of finding those with function values less than the previous ones. This can be carried out in a very simple way without using local minimization (as in the Price algorithm) or through some local minimization and clustering technique (as in the Boender algorithm) [9].

Both algorithms for global minimization are well suited to a multiprocessor environment because they can be easily organized in tasks which can be executed by different processors with some synchronization for exchanging information.

The main steps of the parallel Price algorithm [1,9] are: Initial phase — executed by one processor.

STEP 1. Generate a sample of points using a random number routine through a single processor and compute their function values through different processors.

Interative phase — carried out asynchronously by different processors.

STEP 2. Sort the function values and select the greatest.

STEP 3. Select randomly a subset of the sample and create a new point modifying the centroids of the first n points with the $(n + 1)$ – s t point. Compute the new function value.

STEP 4. If the stop condition is not verified, replace the element with the greatest function value in the sample with the new one. Go to step 2.
The first processor which satisfies the stop criterion interrupts the work of the other processors.

Final phase — executed by one processor.

STEP 5. Determination of global minimum among the points of the sample.

The main steps of the parallel Boender algorithm are:

Initial phase — It is carried out by one processor.

STEP 1. A sample of points is generated through a random number generator and the function values are computed. From each of the points in the sample a local minimization is carried out through a Quasi-Newton method, creating a set C of minima and a set C_1 of points that originate the minima already found.

Iterative phase — carried out by each processor.

STEP 2. Create a new sample of increasing dimension. Apply a clustering technique to the new sample to avoid doing local searches which lead to minima already determined.

If all the points in the sample are clustered, go to step 3 and let the other processors known to stop. Otherwise execute a local search from the unclustered ones and repeat from step 2.

Final phase — executed by only one processor.

STEP 3. Search for the global minimum among the local ones.

For the Price algorithm, the vectorization introduced only little improvements, while the multitasked version was very good. The estimated speed-up with four processors was 3.02 but the maximum number of iterations was more than four times better. The simulation was carried out supposing 4 CPUs [1,6]. Almost the same results were obtained for the Boender algorithm for which we got a speed-up of 2.93 [7].

We ran the Price algorithm on the IBM machine both in simulation and not. In simulation the results show a very good efficiency in the range of [0.82, 0.89] with the number of processors between 1 and 6 [6]. Using the real system with the FPS processors connected to the IBM machine, we got very poor performance. From the measurements it comes out that the time spent on transferring the data among the APs is too high with respect to the time spent on computation within the APs; that means that the algorithm is very well suited to a multitasked environment but with shared memory and, unless we have very expensive function, no gain is reached using the array processors [1].

We got a very good speed-up in the simulation on the Boender algorithm both in term of function evaluations and of time and efficiency in the range of (.7,.89) [9].

From these first experiences we realized that both for vectorization and for parallelization the algorithms must be well structured [2]; if they are not, it is necessary to restructure them.

4. Vectorized and parallel algorithms for local minimization

The previous experiments show that stochastic algorithms for global optimization are certainly very good for parallelization, because the parallelism is intrinsic to the algorithm in the sense that it is an obvious solution to spread over the various processors the different searches for a local minimum.

We realized that it is important to have good subroutines for local minimization which can take advantage from vectorization and parallelism. Two interesting approaches have recently appeared in the literature:
— one in Dixon and Dembo [16,20] known as the truncated Newton method;
— the other one in Han [24], known as the quasi-Newton method through conjugate subspaces.

We decided to implement the algorithm by ourselves avoiding subroutines from any library, in such a way as to take the full advantage both from vectorization and from parallelism. We briefly describe the algorithms outlining the use of vectorization and of parallelism.

Han's algorithm

The algorithm is characterized by a Quasi-Newton scheme for estimating the

Hessian. The search directions are chosen conjugate with respect to the Hessian and such that they can be computed in parallel.

It is well known that the Quasi-Newton method is an iterative method based on the idea of minimizing the approximation:

$$f(x+d) = q(d) = f(x) + \nabla f(x)^T d + 1/2\, d^T H d$$

where x is an estimate of the solution and H an estimate of the Hessian $\nabla^2 f(x)$. This is carried out by solving the linear system:

$$Hd = -\nabla f(x).$$

Then a new estimate of the solution is computed through the formula $\bar{x} = x + ad$ where the α parameter is determined in a line search that quarantees the decrease of the function. If the solution found is not sufficiently accurate the matrix H is updated to $\bar{H}$. The matrix $\bar{H}$ usually satisfies the following conditions: a) the so called Quasi-Newton equation

$$\bar{H}s = y$$

where $s = \bar{x} - x$ and $y = \nabla f(\bar{x}) - \nabla f(x)$;
b) the symmetry condition:

$$\bar{H} = \bar{H}^T.$$

The above condition does not define uniquely the update; one of the possible further conditions is thet one requiring that the iterative process finds the minimum of a quadratic convex function in a finite number of steps [23, 29]. This approach generates one of the most successful schemes, i.e. the BFGS formula.
The Han's idea is to decompose the computation of d as the sum of other directions which can be computed in parallel.
The following theorem [24] is proved:
„*If the Hessian matrix* $\nabla^2 f(x)$ *is constant positive definite and the search direction subspaces* $S_1,...,S_m$ *are conjugate with respect to it, then the search direction* $d = d_1 + ... + d_m$, *where* d_1 *minimizes*
$q(q_i) = \nabla f(x) + \nabla f(x)^T d_i + 1/2 d_i{}^T \nabla^2 f(x)\, d_i$ *over* S_i,
is the Newton direction $d = -\nabla^2 f^{-1}(x)\, \nabla f(x)$."
From the given theorem we can conclude that if approximation matrix H is close to the Hessian $\nabla^2 f(x)$ and the subspaces S_i are conjugate to H, the direction sum of the directions on the susbspaces S_i can be taken as a suitable direction for the minimization over S.
A further theorem [24] proves a property indicating how to update these subspaces.

Thus, the possible initial choices of the subspaces satisfying the conditions to be linearly independent and spanning the whole R^n, we choose $P = I$ where P is the $n \times n$ nonsingular matrix $[P_1,...,P_m]$ partitioned in column blocks and P_i is the matrix whose columns form a basis for the subspaces S_i. This is equivalent in BFGS to the choice $H_o = I$.
So the update of the subspaces P_i can be carried out through the update of matrix P_i. As far as the update of matrix P is concerned, remembering that H satisfies the equations:

$$Hs = -\alpha\nabla f(x).$$

we obtain:

$$A = I - sy^T/(y^Ts) - (-\alpha/(f(x)^T sy^T s))^{1/2} s\nabla f(x)^T.$$

and therefore $\bar{P} = AP$,i.e. without using H.
On the other hand the computation of d_i as

$$\min f(x) + \nabla f(x)^T d_i + 1/2 d_i^T H d_i$$

can be performed through:

$$\min f(x) + \nabla f(x)^T P_i W_i + 1/2 W_i^T P_i H P_i W_i$$

by setting $d_i = P_i W_i$. Since it is possible to prove, see [24], that $P_i^T H P_i = I$, w_i can be computed by $w_i = -P_i^T \nabla f(x)$ i.e. without using H.
The algorithm shows two points where it is possible to take advantage from the parallelism:
— one is in the updating of matrix P because it is possible to carry out the computations of blocks of columns simultaneously;
— the other is in the computations of search directions d_i. We didn't take advantage of parallelism in the line search although parallel multisection techniques are available.
We describe in details the steps of the algorithm outlining sequential parts and parallel parts.

STEP 1. Set $k = 1$, $x_k = x_1$ and $P_1 = [P_{11},...,P_{1m}] = I$ where m is the number of processors in the system.

STEP 2. executed in parallel by the m processors —
Compute $w_{ki} = -P^T_{ki} f(x_k)$
and $d_{ki} = P_{ki} W_{ki}$ for $i = 1,...,m$.

STEP 3. Compute the search direction $d_k = d_{k1} + ... + d_{km}$.

STEP 4. Determine a step size through a line search procedure and set $x_k = x_k + \alpha_k d_k$.
Stop if the stop criterion is satisfied, i.e.

$\| g(x_{k+1}) \| < \delta$ where δ is a user-defined accuracy, otherwise goto step 5.

STEP 5. executed in parallel by the m processors —
Compute a subset of column p of P_{k+1} through the formula:

$$\begin{aligned} P_{k+1} &= P_k - (u^T{}_k\, p_k)\, s_k \\ \text{where } u_k &= (y_k / (y^T{}_k s_k) + (-\alpha_k / (s^T g_k S_k^T y_k))^{1/2} g \\ \text{with } y_k &= \nabla f(x_{k+1}) - \nabla f(x_k) \\ s_k &= x_{k+1} - x_k \\ g_k &= \nabla f(x_k) \end{aligned}$$

STEP 6. Set $k = k + 1$; goto step 2.

We can observe that step 2. and step 5. can be easily parallelized in our environment because they are do-loops over the i. Besides, within the steps 2. and 5., computations must be carried out between matrix and vector and this can be done easily using vector facilities. Step 3. can be vectorized too, as well as computation of the new x in step 4. So the only part that does not take advantage of both vectorization and of parallelism is the line search based on quadratic fitting.

At a first instance we decided to take the maximum advantage from the vector facility.

Let T_s be the time needed to run the code without using the vector facility and T_v by the time needed to run the code using the vector facility, we then get [10] the following results with respect to the three test functions:

— 1) Rosenbrock

— 2) Powell

— 3) Wood

using the following dimensionalities:

$- n = 200, 300, 400, 500$:

Test function	n	T_s	T_v
1	200	9.38	0.385
1	300	28.92	1.11
1	400	45.	1.44
1	500	*	3.41
2	200	7.71	0.43
2	300	22.55	1.37
2	400	44.	1.82
2	500	*	2.65
3	200	18.24	1.4
3	300	55.32	3.41
3	400	240.	6.82
3	500	*	12.17

where * means that we avoid doing these computation because they would have taken too much time. It can be seen that tremendous improvement was achieved by using the vector facility.

Using microtasking on those Do-loops which allow it (mainly in computing the directions d_i and in updating of the matrix A) we got the following results. Let S be the speed-up using three processors with vector facility instead of one. Let us call E the efficiency computed as $E = S/p$, where p is of the number of processors. We got, [10], a very good efficiency varying in the range of (0.73,0.8) with an overhead around 10% of time.

Dembo and Dixon's algorithm

As we have discussed before in most of the methods used solution of the so called Newton-like equations entails most of the computations. For this reason, when far from the solution of the inital problem there is no need of an accurate solution. Dembo and Steihaug [16] proposed to solve these equations through the conjugate gradient method and when far from the solution of the initial problem to stop the solution of these equations. Further improvements of this method have been proposed by Dixon [20], who introduced the concept of the trust region. We analyse an algorithm, based on these ideas, on a vector machine. One of the most known methods of solving these sets of nonlinear equations in the Newton method which suffers of some drawbacks like lack of global convergence and singularity of the Hessian. Many modifications have been proposed that guarantee overcoming of these problems.
Following Dembo and Dixon we examine the following algorithm:

STEP 1 Compute $f(x_k)$, $g(x_k) = \nabla f(x_k)$, $H(x_k)$
STEP 2 Stop if the convergence criterion is satisfied i.e. $\| g(x_k) \| < \delta$, where δ is a user defined accuracy.
STEP 3 Compute p_k such that:
$$H_k p_k = - g(x_k)$$
where H_k is a good approximation to $H(x_k)$. The solution of the system is carried out through a conjugate gradient algorithm.
STEP 4 Find α_k such that the point
$$x_{k+1} = x_k + \alpha_k p_k$$
satisfies Wolfe and Goldstein-Armijo conditions.
STEP 5 $k = k + 1$; go to step 1.

Because Newton's equations are derived from a quadratic model of the function around a minimum, they give the correct direction when you are very close to the minimum. For this reason when you are far from the minimum it is worthwhile to solve these equation accurately; in this case it is the major iteration that provides for getting closer to the minimum.

Dembo's idea was to truncate the computations in step 3. when the following condition was satisfied:

$$\frac{r^T{}_j\, r_j}{\nabla f(x_k)^T \nabla f(x_k)} < \min\,(0.1/k^2,\ \nabla f(x_k)^T \nabla f(x_k))$$

where r_j is the residual at the j-th iteration of conjugate gradient algorithm in step 3.
Dixon added a criterion to ensure that at the end of each minor iteration the new point satisfies the Wolfe condition and lies within the trust region around the previous point. The radius of the trust region is modified at each major iteration accordingly to the current value of the step size.
Certainly, on a sequential machine and with a large problem to solve the idea to truncate the Newton equations before solution may affect heavily the time for finding the solution of the optimisation problem.
We wanted to verify if this idea is so much effective also on a vector machine.
We tested the algorithm on three clasical test functions:
— Rosenbrock
— Powell
— Dixon
with the following dimensions:
$n = 20,40,80,100,160,200$.
The results [11] show a speed-up of more than three on most of the cases and we have to take into account that these results were obtained automatically, in the sense that the compiler did the most part of vectorization.

5. Conclusions

This work has shown that it is possible to take significant advantage by the parallelisation of the algorithm resulting both from vectorization and from parallelization if the algorithm is well structured. For vectorization the automatic compilers are very good but further restructuring of the code done by the user may increase the performance.

Future experiments would include actual problems where it would be interesting to include in the algorithm also parallelisation in the computation of functions and gradients.

References

[1] Bertocchi M. Analisi e sperimentazione di algoritmi paralleli per l'ottimizzazione globale. Ricerca operativa e informatica. Bielli M.,ed. Milano 1986, F. Angeli, pp. 703-714.

[2] Bertocchi M., Krafft W. A methodology and suggested tools for productioon of parallel programs. *QDMSIA*, (1987) 5.

[3] BERTOCCHI M., KRAFFT W. Structured Fortran 77: a precompiler. *QDMSIA*, (1987) 5.

[4] BERTOCCHI M., KRAFFT W. Introducing parallelism into the CRAY Fortran. *QDMSIA*, (1987) 7.

[5] BERTOCCHI M., KRAFFT W. Application of parallel programming techniques: a modified Newton algorithm for local optimisation. *QDMSIA*, (1987) 8.

[6] BERTOCCHI M., KRAFFT W. Application of parallel programming techniques: the Price algorithm for global optimisation. *QDMSIA*, (1987) 9.

[7] BERTOCCHI M., KRAFFT W. Application of parallel programming techniques: the Boender algorithm for global optimisation. *QDMSIA*, (1987) 10.

[8] BERTOCCHI M., KRAFFT W. A global optimisation algorithm for parallel machines. Optimization techniques and applications. Teo, Paul, Chew, Wang eds. National University of Singapore 1987.

[9] BERTOCCHI M., A parallel algorithm for global optimization. *QDMSIA*, (1987) 19.

[10] BERTOCCHI M., GNUDI A., MOHSENINIA M. Numerical experiments on an unconstrained parallel algorithm for optimization based on conjugate subspace. *QDMSIA*, (1988) 11.

[11] BERTOCCHI M., Numerical experiences of the truncated Newton algorithm for local minimization on a vector machine.

[12] BRATTEN C., CLARK R., DORN P., GRANT R., IBM 3090 Engineering/Scientific Performance. *Tec. Bull.* GG66-0245, 1986.

[13] CLARK R.S., WILSON T.L. Vector system performance of the IBM 3090. *IBM Systems Journal*, **25** (1986) 1.

[14] Cray Corporation. Microtasking User's guide.

[15] Cray Corporation, Cray X-MP multitasking programmer's references. Manual SN-0222. 1986

[16] DEMBO R., STEIHAUG T. Truncated Newton methods for large scale optimisation. *Mathematical Programming*, **26** (1983), 190-212.

[17] DENNIS J.E., SCHABEL R. Numerical methods for unconstrained optimization and nonlinear equations. Prentice Hall, New Jersey, 1983.

[18] DICHIO P., ZECCA V. IBM ECSEC facilities: user's guide. G513-4080, December 1985.

[19] DIXON L.C.W., PATEL K.D., DUCKSBURY P.G. Experience in running optimization algorithms on parallel processing systems. *Tec. Rep.* no 138. Hatfield Polytechnic, 1983.

[20] DIXON L.C.W., PRICE R.C. The truncated Newton method for sparse unconstrained optimisation using automatic differenation. *Tec. Rep.* N.O.C. no 170, Hatfield 1986.

[21] DUFF I.S. The influence of vector and parallel processors on numerical analysis. AERE-R12329, September 1986.

[22] GIBSON D.H., RAIN D.W., WALSH H.F. Engineering and scientific processing on the 3090. *IBM Systems Journal*, **25** (1986) 1.

[23] GILL P.E., MURRAY W., WRIGHT M.H. Practical optimization. London, Academic Press, 1981.

[24] HAN S.P. Optimization by updated conjugate subspaces. DAMTP 1985/NA9, University of Illinois, 1985.

[25] HESTENES M.R. Conjugate — direction method in optimization. Berlin, Springer-Verlag, 1980.

[26] RALL L.B. Global optimisation using automatic differentiation and interval iteration. Mathematics Research Centre, University of Wisconsin, 1985.

[27] RITTER . Parrallel automatic differentiation. Proc. of Parallel Optimization. Madison, 1987.

[28] SCHNABEL R.B. Concurrent function evaluations in local and global optimization. *Tec. Rep.* CS-CU-345-86, University of Colorado, Boulder 1986.

[29] SPEDICATO E. Algoritmi per la minimizzazione di funzioni non lineari non vincolate. *Quaderni IAC, Serie 3, no 15, Roma, 1975.*

Received, April 1988.

Optymalizacja równoległa

W pracy podsumowano wyniki i doświadczenia zgromadzone w trakcie badań nad tworzeniem programów równoległych dla wybranych algorytmów optymalizacji używających dostępnego obecnie sprzętu komputerowego umożliwiającego liczenie równoległe.

Параллельная оптимизация

В работе обобщены результаты и опыт накопленный во время исследований по разработке параллельных программ для избранных алгоритмов оптимизадии, использующих доступное в настоящее время компьютерное оборудование, позволяющие проводить параллельные вычисления.