Testing Parallel Variable Transformation on VPP500

Eiki Yamakawa
College of Business Administration
Takamatsu University
Takamatsu, Kagawa 761-0194, Japan
E-mail: yamakawa@takamatsu-u.ac.jp
and
Masao Fukushima [1]
Department of Applied Mathematics and Physics
Graduate School of Engineering
Kyoto University
Kyoto 606-01, Japan
E-mail: fuku@kuamp.kyoto-u.ac.jp

December 26, 1997

Abstract. This paper studies performance of the parallel variable transformation (PVT) algorithm for unconstrained nonlinear optimization through numerical experiments on a Fujitsu VPP500, one of the most up-to-date vector parallel computer. Special attention is paid to a particular form of the PVT algorithm that is regarded as a generalization of the block Jacobi algorithm that allows overlapping of variables among processors. Implementation strategies on the VPP500 are described in detail and results of numerical experiments are reported.

Key words. Parallel variable transformation, generalized block Jacobi algorithm, unconstrained optimization, nonlinear programming, VPP500

[1] The work of this author was supported in part by the Scientific Research Grant-in-Aid from the Ministry of Education, Science, Sports and Culture, Japan.

1. Introduction

In the last several years, a number of novel parallel algorithms for solving nonlinear optimization problems have been developed. Among others, Mangasarian [8] proposed the parallel gradient distribution (PGD) algorithm, and Ferris and Mangasarian [2] presented the parallel variable distribution (PVD) algorithm, which was further studied by Solodov [13]. More recently, one of the authors [4] presented a general framework for unconstrained nonlinear optimization called the parallel variable transformation (PVT) algorithm that encompasses the above-mentioned parallel algorithms.

The purpose of this paper is two-fold. The first is to study performance of the PVT algorithm for large-scale unconstrained optimization problems. Special attention will be paid to a particular form of the PVT algorithm that is regarded as a generalization of the block Jacobi method that allows overlapping of variables among processors. We will show that this specific PVT algorithm is effective for large-scale nonlinear least squares problems. The second is to describe how the algorithm is implemented on a Fujitsu VPP500, which is one of the most up-to-date vector parallel computer. Although numerical experiments with parallel algorithms have been frequently reported in the literature, most of the recent experiments seem to have been carried out on a Connection Machine CM-2 or CM-5. Since a Fujitsu VPP500's architecture is considerably different from those machines, and since its performance seems very high, we believe that a detailed description of the implementation strategies used in the numerical experiments is of interest to the people in the optimization community.

This paper is organized as follows. In §2, we state the PVT algorithm and its global convergence and rate of convergence results. We also mention the generalized block Jacobi algorithm. In §3, we describe in detail implementation strategies of the block Jacobi algorithm on a VPP500. In §4, we report results of numerical experiments. In §5, we conclude the paper with some remarks.

2. Parallel Variable Transformation

In this section, we state the *parallel variable transformation* (PVT) algorithm for solving the unconstrained minimization problem

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

and briefly review the convergence results established in [4]. Throughout the objective function $f:\Re^n\to\Re$ is assumed to be continuously differentiable and bounded below.

2.1. Algorithm

Each iteration of the PVT algorithm consists of the *parallelization* phase, where multiple candidate solutions for the next iterate are produced in parallel using p processors, and the *synchronization* phase, where the next iterate is computed from the candidate solutions obtained in the parallelization phase. More specifically, the PVT algorithm for solving problem (1) is stated as follows:

Algorithm PVT (Parallel Variable Transformation)
Step 0 (Initialization): Let m_ℓ, $\ell=1,\ldots,p$, be positive integers such that $m_1+\cdots+m_p\geq n$. Choose an initial point $x^{(0)}\in\Re^n$ and let $k:=0$.
Step 1 (Parallelization): For each $\ell\in\{1,\ldots,p\}$, choose an $n\times m_\ell$ matrix $A_\ell^{(k)}$ and find an approximate solution $y_\ell^{(k)}\in\Re^{m_\ell}$ to the minimization problem

$$\min_{y_\ell\in\Re^{m_\ell}} \ \varphi_\ell^{(k)}(y_\ell) \ \equiv \ f(A_\ell^{(k)}y_\ell+x^{(k)}). \tag{2}$$

If $\nabla\varphi_\ell^{(k)}(0)=0$ for all ℓ, then stop. Otherwise, go to Step 2.
Step 2 (Synchronization): Let $B^{(k)}$ be the $n\times(p+1)$ matrix with its columns being $x^{(k)}$ and $A_\ell^{(k)}y_\ell^{(k)}+x^{(k)}$, $\ell=1,\ldots,p$. Find an approximate solution $z^{(k)}=(z_0^{(k)},z_1^{(k)},\ldots,z_p^{(k)})^\top\in\Re^{p+1}$ to the minimization problem

$$\min_{z\in\Re^{p+1}} \ \psi^{(k)}(z) \ \equiv \ f(B^{(k)}z). \tag{3}$$

Let $x^{(k+1)}:=B^{(k)}z^{(k)}$ and $k:=k+1$. Return to Step 1. □

In the algorithm, we need not solve very accurately problems (2) and (3) in Steps 1 and 2, respectively. In fact, it will suffice to compute, for each k, vectors $y_\ell^{(k)}$ satisfying

$$\varphi_\ell^{(k)}(y_\ell^{(k)}) \leq \varphi_\ell^{(k)}(0) - \sigma_\ell^{(k)}(\|\nabla\varphi_\ell^{(k)}(0)\|), \quad \ell = 1,\ldots,p, \tag{4}$$

where $\sigma_\ell^{(k)} : \Re \to \Re$ is a forcing function [12] associated with the function $\varphi_\ell^{(k)}$, and a vector $z^{(k)}$ satisfying

$$\psi^{(k)}(z^{(k)}) \leq \min_{1\leq\ell\leq p} \varphi_\ell^{(k)}(y_\ell^{(k)}). \tag{5}$$

Note that condition (4) will be satisfied by executing a single descent iteration for $\varphi_\ell^{(k)}$ with starting point $y_\ell = 0$ (see [12, Chapter 14] and [8, pp. 1919–1920]). Also condition (5) is trivially satisfied if we simply choose $z^{(k)}$ to be the unit vector such that $B^{(k)}z^{(k)} = A_{\tilde\ell}^{(k)}y_{\tilde\ell}^{(k)} + x^{(k)}$, where $\tilde\ell = \arg\min_{1\leq\ell\leq p} \varphi_\ell^{(k)}(y_\ell^{(k)})$. In other words, we may choose $x^{(k+1)}$ to be the best of the candidate points $A_\ell^{(k)}y_\ell^{(k)} + x^{(k)}$, $\ell = 1,\ldots,p$.

2.2. Convergence

Global convergence of the PVT algorithm is ensured under the following assumptions:
(A1) Let $A^{(k)} := [A_1^{(k)} \cdots A_p^{(k)}] \in \Re^{n\times(m_1+\cdots+m_p)}$. There exists a constant $\beta > 0$ independent of k such that, for all k,

$$\|(A^{(k)})^\top x\| \geq \beta\|x\| \qquad \text{for all } \ x \in \Re^n,$$

or equivalently, the sequence $\{A^{(k)}(A^{(k)})^\top\}$ of $n \times n$ matrices is uniformly positive definite
(A2) For each ℓ, the sequence $\{\sigma_\ell^{(k)} \,|\, k = 0,1,2,\ldots\}$ of forcing functions in (4) satisfies the property that, for any $\varepsilon > 0$, there exists a $\delta > 0$ independent of k such that for all k

$$\sigma_\ell^{(k)}(t) < \delta \quad \Longrightarrow \quad t < \varepsilon.$$

Theorem 2.1 ([4]) *Suppose that, at each iteration of the PVT algorithm, conditions* (4) *and* (5) *are satisfied in Steps 1 and 2, respectively. Assume in addition that conditions* (A1) *and* (A2) *are satisfied. Then the PVT algorithm either terminates at a stationary point of problem* (1) *or generates an infinite sequence* $\{x^{(k)}\}$ *such that*

$$\lim_{k\to\infty} \nabla f(x^{(k)}) = 0$$

and every accumulation point of $\{x^{(k)}\}$ *is a stationary point of problem* (1).

The rate of convergence of the algorithm can be established by replacing condition (A2) with the slightly more restrictive condition:
(A2') For each ℓ, condition (4) is satisfied with

$$\sigma_\ell^{(k)}(t) = \delta_\ell t^2 \quad k = 0, 1, 2, \ldots,$$

where δ_ℓ is a positive constant independent of k.

The next theorem indicates that the PVT algorithm is linearly convergent.

Theorem 2.2 ([4]) *Let S^* denote the set of minima of f. Suppose that there exist constants $\rho > 0, c_1 > 0$ and $c_2 > 0$ such that $\operatorname{dist}[x, S^*] \leq \rho$ implies*

$$\|\nabla f(x)\| \geq c_1 \operatorname{dist}[x, S^*] \tag{6}$$

and

$$f(x) - f^* \leq c_2 \operatorname{dist}[x, S^*]^2, \tag{7}$$

where $\operatorname{dist}[x, S^*] \equiv \inf\{\|x - x'\| \,|\, x' \in S^*\}$ and $f^* \equiv \min_{x \in \Re^n} f(x)$. Assume that the PVT algorithm generates a sequence $\{x^{(k)}\}$ satisfying $\lim_{k \to \infty} \operatorname{dist}[x^{(k)}, S^*] = 0$. Moreover suppose that conditions (4) and (5) are satisfied in Steps 1 and 2, respectively, and that conditions (A1) and (A2') are satisfied. Then $\{f(x^{(k)})\}$ converges Q-linearly to f^* and $\{\operatorname{dist}[x^{(k)}, S^*]\}$ converges R-linearly to zero.

Note that (6) holds if f is strongly convex around the minimum of f, in which case S^* reduces to a singleton. Condition (7) is satisfied if ∇f is Lipschitz continuous around S^*.

2.3. Generalized Block Jacobi Algorithm

The PVT algorithm is broad enough to contain a number of existing parallel optimization algorithms. In fact, it was shown in [4] that the parallel gradient distribution (PGD) algorithm of Mangasarian [8], the unconstrained version of the parallel variable distribution (PVD) algorithm proposed by Ferris and Mangasarian [2] and further studied by Solodov [13] belong to the class of the PVT algorithm. The updated conjugate subspaces (UCS) method of Han [5] was also shown to be closely related to the PVT algorithm [4].

In this section, we focus on the special case of the PVT algorithm that was suggested in [4]. This algorithm is regarded as a natural extension of the block Jacobi method. Let J_ℓ, $\ell = 1, \ldots, p$, be sets of indices such that

$$J_1 \cup \cdots \cup J_p = \{1, \ldots, n\}. \tag{8}$$

We note that the sets J_ℓ, $\ell = 1, \ldots, p$, need not be disjoint to each other. Let

$$m_\ell := |J_\ell|, \quad \ell = 1, \ldots, p,$$

and, for each $\ell \in \{1, \ldots, p\}$, let $A_\ell^{(k)}$ be the $n \times m_\ell$ matrix with $\operatorname{rank} A_\ell^{(k)} = m_\ell$ such that, for any $j \in J_\ell$, the jth row of $A_\ell^{(k)}$ is a row of the $m_\ell \times m_\ell$ identity matrix, and the other rows of $A_\ell^{(k)}$ are all zero. It is easy to see that condition (A1) is satisfied with this particular choice of the matrices $A_\ell^{(k)}$.

For each ℓ, let $x_{J_\ell} \in \Re^{m_\ell}$ and $x_{\overline{J}_\ell} \in \Re^{n-m_\ell}$ denote the subvectors of $x \in \Re^n$ with those elements which correspond to the index set J_ℓ and its complement in $\{1, \ldots, n\}$, respectively. With the identification

$$y_\ell = x_{J_\ell} - x_{J_\ell}^{(k)}, \quad \ell = 1, \ldots, p,$$

subproblem (2) in the PVT algorithm can then be written as

$$\min_{x_{J_\ell} \in \Re^{m_\ell}} f(x_{J_\ell}, x_{\overline{J}_\ell}^{(k)}), \tag{9}$$

which looks exactly the same as that of the block Jacobi method, except a possible overlap of variables among subproblems.

The transformation matrices $A_\ell^{(k)}$ can be modified using the idea behind the PVD algorithm. Specifically, we may augment the matrices $A_\ell^{(k)}$ by adding arbitrary columns that are generated using, for example, the steepest descent or quasi-Newton directions in some subspaces of appropriate dimension. As a result, it is expected that each processor obtains a better solution in the parallelization phase at the expense of the increase in the number of variables in subproblem (2).

3. Implementation on VPP500

We now implement **Algorithm PVT** with some special choices of the transformation matrix $A_\ell^{(k)}$ on a vector parallel computer VPP500. In this section, we first state the characteristics

of the VPP500 hardware and software, and then describe the design of the specific algorithms.

3.1. Vector Parallel Computer VPP500

Fujitsu VPP500 is a multiprocessor system consisting of the Global System Processor (GSP) and the Vector Parallel Processor (VPP); the former compiles a source program, and the latter executes the object code [6]. The VPP is a parallel supercomputer with several Processor Elements (PE's) for parallel operation and a Control Processor (CP) for coordination of all PE's. Each PE has a scalar unit and a vector unit, whose performance are estimated as 200 MFLOPS and 1.6 GFLOPS (peak), respectively. All PE's and CP are interconnected by a cross-bar network for data communication whose transfer rate is 400 Mbps.

In order to code parallel algorithms, a special programming language VPP FORTRAN 77 is provided on the VPP system [7]. VPP FORTRAN 77 is an extension of FORTRAN 77, in which various compiler directives beginning with `!XOCL` are available to execute a specified procedure on multiple PE's in a parallel fashion and to transmit local data residing in one PE to other PE's. Although the layout of the data in the distributed memory as well as the way of dividing the procedure into PE's has to be explicitly described in the program, the compiler automatically vectorizes the program executed on each PE. Therefore, it seems relatively easy to obtain good performance on the VPP system by using VPP FORTRAN 77.

In the program written in VPP FORTRAN 77, we may use the following directive lines (see [3] for details);

- `!XOCL PROCESSOR` *processor_group_name* (*integer*)
- `!XOCL LOCAL` *variable_name*
- `!XOCL GLOBAL` *variable_name*
- `!XOCL PARALLEL REGION`
- `!XOCL END PARALLEL`
- `!XOCL SPREAD REGION /` (*processor_group_name* (*processor_number*))
- `!XOCL REGION /` (*processor_group_name* (*processor_number*))

- `!XOCL SPREAD DO`
- `!XOCL END SPREAD`
- `!XOCL BROADCAST (`*variable_name*`) (`*flag*`)`

At the beginning of the program, the `PROCESSOR` directive is located to declare the number of available PE's. However, the program is executed on a single PE before the specific directive line `!XOCL PARALLEL REGION` is encountered. If the system encounters the `PARALLEL REGION` directive, then all the PE's start executing the same program on their own copy of data until they find the `END PARALLEL` directive. Within the `PARALLEL REGION`, each PE may execute their own operations. For example, the `SPREAD REGION` directive with `END SPREAD` and an optional `REGION` specifies the statement list executed on individual PE's in a manner similar to the block `IF` structure in FORTRAN 77. This scheme corresponds to the control parallel programming model. On the other hand, an iterative `DO` loop supplemented by the `SPREAD DO` directive is divided among the available PE's. Thus the elementwise array arithmetic can effectively be executed on the VPP, which corresponds to the data parallel programming model.

VPP FORTRAN 77 provides several global functions such as `SUM`, `MAX`, `MIN`, `AND` and `OR` relevant to the `SPREAD DO` structure. These functions adjust the distributed local data with the same variable name in which each PE holds the results by executing a part of the `DO` loop. Since the argument of each global function may be an array variable as well as a scalar variable, most of the data parallel operations can be written as the `SPREAD DO` structure with some global function. On the other hand, the `BROADCAST` directive copies the values of local data in a specific PE to all other PE's independently of the `SPREAD DO` structure. Note that all the PE's are synchronized before executing the `BROADCAST` operation. They are also synchronized at the directive lines `!XOCL SPREAD REGION`, `!XOCL SPREAD DO` and `!XOCL END SPREAD` under normal conditions. However, each PE may execute its own procedure asynchronously, for example, by directing `END SPREAD NOBARRIER` instead of simply directing `END SPREAD`.

Within the `PARALLEL REGION`, every variable in the program is usually held in all PE's individually. The `LOCAL` directive enables us to distribute the elements of an array over the local memories of all PE's. Note that the distributed `LOCAL` array is only available within the `PARALLEL REGION`, and each PE may only access a part of the array residing its own local

memory. On the other hand, the variables declared with the `GLOBAL` directive are held among all the PE's in common. Since a `GLOBAL` scalar variable is located in a specific PE and the elements of the `GLOBAL` array variable are distributed in a specific group of PE's, it takes communication overheads to access these data. Note that the `GLOBAL` variables cannot be manipulated on the vector unit of each PE.

3.2. Implementation Strategies

We now state some strategies to implement the PVT algorithm, particularly the generalized block Jacobi algorithm described in §2.3, on a VPP500. In the following discussion, we assume that the number of subproblems to be solved in the parallelization phase is no greater than the number of PE's, so that each subproblem can be assigned to a single PE exclusively.

One of the simplest execution models of the algorithm is to repeat the parallelization phase and the synchronization phase alternately until a termination criterion is satisfied; the former is located within the `PARALLEL REGION` and the latter is located outside of it. In fact, the iterate updated in the synchronization phase is automatically transmitted to all the PE's that execute the parallelization phase, because all data in the program are copied to the local memory of each PE when the system encounters the `PARALLEL REGION` directive. In general, however, it is expensive to execute the compiler directive line `!XOCL PARALLEL REGION` frequently, because this instruction makes us prepare the whole system for starting a parallel operation.

In view of this characteristic of VPP FORTRAN 77, we decided to locate the synchronization phase as well as the parallelization phase within the `PARALLEL REGION`. In general, distributed memory systems like a CM-5 provide the message passing primitives which enable us to implement coarse-grained parallel algorithms by using the master-worker model [14]. Although this model is quite natural for the execution of the PVT algorithm, the programs to gather the candidates for the next iterate and to transmit the resulting next iterate are often somewhat complicated. On the other hand, VPP FORTRAN 77 supplies several devices such as global functions, the `BROADCAST` directive and the `GLOBAL` variables for executing various parallel operations as if the VPP is a shared memory system. Then, by applying these devices, each PE can easily use the results obtained in the other PE's without message passing.

In the sequel, we execute the synchronization phase on all PE's simultaneously, which can be realized by describing the procedure for the synchronization phase within the `PARALLEL`

REGION without any compiler directive lines. By doing so, each PE can immediately start the parallelization phase by making use of the next iterate computed there during the synchronization phase. Let each PE solve its own subproblem in the parallelization phase independently by applying an identical optimization algorithm. Then we may code the parallelization phase using an iterative DO loop with a control variable ℓ in the same manner as in the ordinary serial programming language FORTRAN 77 except for the additional SPREAD DO directive. Moreover, the candidate for the next iterate computed in each PE is efficiently transmitted to all other PE's by simply appending a global function to the SPREAD DO structure as mentioned below.

We now state how the VPP system holds the working variables used in the algorithm. Since the PVT algorithm is free from the concept of assigning a separate block of variables involved in the original problem to each PE, all of the array variables are independently stored in each PE as a whole instead of distributing their elements among available PE's. Thus we may declare the variables without any compiler directive such as LOCAL or GLOBAL as if the program is written in ordinary FORTRAN 77. For example, in the synchronization phase, a two-dimensional array corresponding to an $n \times p$ matrix $\overline{B}^{(k)}$ is held in each PE, where $\overline{B}^{(k)}$ is a submatrix of $B^{(k)}$ obtained by eliminating its first column $x^{(k)}$. In the parallelization phase, the ℓth PE stores the candidate solution $A_\ell^{(k)} y_\ell^{(k)} + x^{(k)}$ as the ℓth column of $\overline{B}^{(k)}$ and sets the other columns equal to zero. Then the SPREAD DO structure of the parallelization phase is completed by executing of the global function SUM with $\overline{B}^{(k)}$ as an argument, which makes the value of matrix $\overline{B}^{(k)}$ residing in each PE identical. Moreover, making use of the global function MIN relevant to the SPREAD DO structure enables us to find the index ℓ of the PE that gives the minimum of the objective values $\varphi_\ell^{(k)}(y_\ell^{(k)})$.

Under the above mentioned strategies, the VPP FORTRAN 77 code requires only about 10 additional directive lines compared with its FORTRAN 77 counterpart for serial computation, except for the operation to clear the value of matrix $\overline{B}^{(k)}$. In general, we may easily code a program for executing a coarse-grained parallel algorithm on the VPP system in the following way: First we code parallel operations in a serial fashion by using an iterative DO loop, and then enclose them with the SPREAD DO directive and the END SPREAD directive supplemented by a global function to broadcast the results obtained in each PE.

It is commonly recognized that the main computational burden of a nonlinear optimization

algorithm for large-scale unconstrained minimization is to evaluate function values for determining stepsizes and to evaluate gradient values for generating search directions. We therefore introduce some devices to compute the objective values $f(x)$ and the gradient values $\nabla f(x)$ of the original problem. We now consider the parallelization phase of the generalized block Jacobi method described in §2.3, where the transformation matrices $A_\ell^{(k)}$ involved in the subproblem have smaller number of non-zero rows than n. When the original problem has some characteristic structure such as the minimization of the squared-sum of several functions, the evaluation of the objective function f can often be decomposed into the following two parts: One is evaluated by using the variables corresponding to the non-zero rows of $A_\ell^{(k)}$, and the other independently from them. In such a case, the latter may be executed before the parallelization phase, and only the former needs to be calculated when we determine the stepsize in each iteration. Furthermore, we may partially calculate the elements of $\nabla f(x)$ corresponding to the non-zero rows of $A_\ell^{(k)}$ since $\nabla \varphi_\ell^{(k)}(y_\ell) = (A_\ell^{(k)})^\top \nabla f(A_\ell^{(k)} y_\ell + x^{(k)})$. Note that, in general, we cannot apply this strategy to the PVD algorithm of [2], even if the objective function of the original problem has some particular structure, because the PVD algorithm corresponds to the particular PVT algorithm in which the transformation matrices $A_\ell^{(k)}$ may have n non-zero rows.

4. Numerical Experiments

We have conducted computational experiments on a VPP500 with the PVT algorithm, in particular, with the generalized Jacobi method as described in §2.3. In this section, we first state the characteristics of the test problems that we solved, and then report the numerical results.

4.1. Test Problems

We tested the algorithm on several large-scale nonlinear least-squares problems taken from the literature [9, 10, 11] that constitute a part of the source for the CUTE collection [1]. The details of the test problems are given below. Note that we can vary the problem size arbitrarily by specifying the number n of variables.

Problem 4.1 ([9]) *Extended Powell singular function.*

Objective function: $f(x) = \sum_{i=1}^{n/4} \{(x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} - x_{4i})^2 + (x_{4i-2} - 2x_{4i-1})^4 + 10(x_{4i-3} - x_{4i})^4\}$

Starting point: $x^{(0)} = (\ 3, -1,\ 0,\ 1, \ldots,\ 3, -1,\ 0,\ 1)$

Minimum value: $f^* = 0 \quad at \quad x^* = (0, \ldots, 0)$

Problem 4.2 ([9]) *Extended Rosenbrock function.*

Objective function: $f(x) = \sum_{i=1}^{n/2} \{100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2\}$

Starting point: $x^{(0)} = (-1.2,\ 1.0, \ldots, -1.2,\ 1.0)$

Minimum value: $f^* = 0 \quad at \quad x^* = (1, \ldots, 1)$

Problem 4.3 ([10]) *Generalized Rosenbrock function.*

Objective function: $f(x) = 1 + \sum_{i=2}^{n} \{100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2\}$

Starting point: $x^{(0)} = (1/(n+1), \ldots, 1/(n+1))$

Minimum value: $f^* = 1 \quad at \quad x^* = (1, \ldots, 1)$

Problem 4.4 ([9]) *Broyden tridiagonal function.*

Objective function: $f(x) = \sum_{i=1}^{n} \{(3 - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1\}^2$

Starting point: $x^{(0)} = (-1, \ldots, -1)$

Minimum value: $f^* = 0$

Problem 4.5 ([9]) *Broyden banded function.*

Objective function: $f(x) = \sum_{i=1}^{n} \left\{x_i(2 + 5x_i^2) + 1 - \sum_{j \in J_i} x_j(1 + x_j)\right\}^2,$

where $J_i = \{j \,|\, j \neq i, \max(1, i-5) \leq j \leq \min(n, i+1)\}$

Starting point: $x^{(0)} = (-1, \ldots, -1)$

Minimum value: $f^* = 0$

Problem 4.6 ([9]) *Trigonometric function.*

Objective function: $f(x) = \sum_{i=1}^{n} \left\{n - \sum_{j=1}^{n} \cos x_j + i\,(1 - \cos x_i) - \sin x_i\right\}^2$

Starting point: $x^{(0)} = (1, \ldots, 1)$

Minimum value: $f^* = 0 \quad at \quad x^* = (2m\pi, \ldots, 2m\pi), \quad m = 0, \pm 1, \pm 2, \ldots$

Problem 4.7 ([11]) *Quartic.*

Objective function: $f(x) = \Big(\sum_{i=1}^{n} i\, x_i^2 \Big)^2$

Starting point: $x^{(0)} = (1, \ldots, 1)$

Minimum value: $f^* = 0 \quad at \quad x^* = (0, \ldots, 0)$

The starting points shown above are standard except the one for Problem 4.6, for which the standard starting point is $(1/n, \ldots, 1/n)$. We used the starting point $(1, \ldots, 1)$ because the standard one becomes close to the optimal solution $(0, \ldots, 0)$ when n is large. The Hessian matrices of the objective functions in Problems 4.1 and 4.2 have block diagonal structure, with block diagonal parts being 4×4 matrices and 2×2 matrices, respectively. The Hessian matrices of the objective functions in Problems 4.3, 4.4 and 4.5 have band structure; in particular, those matrices in the first two problems are tri-diagonal. On the other hand, the Hessian matrices of the objective functions in Problems 4.6 and 4.7 are dense. To apply the generalized block Jacobi method described in §2.3, we need to specify the index sets J_ℓ, $\ell = 1, \ldots, p$, that satisfy (8). The objective functions of the above test problems have the form $f(x) = \sum_i f_i^2(x)$ except Problem 4.7. For Problems 4.1, 4.2, 4.3, 4.4 and 4.5, in particular, each f_i is a simple function involving a fixed small number of variables no matter how the total number n of variables involved in f becomes large. Then, as described in §3.2, the evaluation of the function f in the solution process of the ℓth subproblem in the parallelization phase can effectively be decomposed into two parts; one is evaluated by using x_{J_ℓ} and a few additional variables, and the other is evaluated without using x_{J_ℓ}. For Problems 4.6 and 4.7, we do not have such natural decomposition, because each f_i involves a term of the form $\sum_{j=1}^{n} h(x_j)$. In this case, however, we may still reduce the cost of function evaluations simply by computing the term $\sum_{j \notin J_\ell} h(x_j)$ prior to the parallelization phase.

4.2. Computational Results

The numerical experiments were carried out in double precision arithmetic on the VPP500 at the Kyoto University Data Processing Center. In order to solve subproblems in the parallelization phase and the synchronization phase, we used a quasi-Newton method with BFGS updates.

In the experiments, we stop the iteration of **Algorithm PVT** if the conditions

$$\|\nabla f(x^{(k)})\| \leq \varepsilon_1 \tag{10}$$

and

$$f(x^{(k-1)}) - f(x^{(k)}) \leq \varepsilon_2 \left(1 + |\, f(x^{(k-1)})\,|\right)$$

hold at the beginning of the parallelization phase. Note that condition (10) is equivalent to the condition

$$\|\nabla \varphi_\ell^{(k)}(0)\| \leq \varepsilon_1, \qquad \ell = 1, \ldots, p.$$

In each PE, we always choose $y_\ell = 0$ as an initial point in the quasi-Newton method for solving subproblem (2). Note that $y_\ell = 0$ corresponds to the current major iterate $x^{(k)}$, which is quite natural as a starting point for finding a candidate for the next iterate. Then we terminate the procedure when y_ℓ satisfies the condition

$$\|\nabla \varphi_\ell^{(k)}(y_\ell)\| \leq \max\left\{\varepsilon_1, \frac{\|\nabla \varphi_\ell^{(k)}(0)\|}{10}\right\}. \tag{11}$$

The term $\|\nabla \varphi_\ell^{(k)}(0)\|/10$ is introduced to truncate inner iterations prematurely especially when the current major iterate $x^{(k)}$ is far from the optimal solution. In practice, $y_\ell^{(k)}$ satisfying the termination criterion (11) is regarded as an approximate solution of subproblem (2). On the other hand, we complete the synchronization phase if the conditions

$$\psi^{(k)}(z) \leq \min_{1 \leq \ell \leq p} \varphi_\ell^{(k)}(y_\ell^{(k)}) \tag{12}$$

and

$$\|\nabla \psi^{(k)}(z)\| \leq \max\left\{\varepsilon_1, \frac{\|\nabla \psi^{(k)}(\tilde{z})\|}{10}\right\}$$

hold, where $\tilde{z}$ is an initial point in the quasi-Newton method for solving subproblem (3). Note that condition (12) is essential in ensuring the global convergence of the algorithm (see (5) and Theorem 2.1). Since the choice of a starting point $\tilde{z}$ may affect the performance of the whole algorithm as well as that of the synchronization phase, we need to choose $\tilde{z}$ carefully by taking into account the structure of the transformation matrices $A_\ell^{(k)}$ (see, for example, (14) below).

We first tested the generalized block Jacobi method on the test problems described in the previous subsection. The number n of variables was fixed at 1000. As mentioned in §2.3, the generalized block Jacobi method is a special case of the PVT algorithm in which $n \times m_\ell$ matrices $A_\ell^{(k)}$ consist of all the rows of the $m_\ell \times m_\ell$ identity matrix and additional $n - m_\ell$ zero rows. In particular, we choose $A_\ell^{(k)}$ in the following manner: Let n_ℓ, $\ell = 0, 1, \ldots, p$, be indices such that

$$0 = n_0 < n_1 < \cdots < n_p = n.$$

Moreover, we choose an integer m such that

$$0 \leq m \leq n_\ell - n_{\ell-1}$$

for all $\ell = 1, \ldots, p$, and set

$$m_\ell = n_\ell - n_{\ell-1} + m.$$

Then, for all k, we let the matrices $A_\ell^{(k)}$ be

$$A_\ell^{(k)} = \begin{pmatrix} \mathbf{0}_{n_{\ell-1}} \\ I_{m_\ell} \\ \mathbf{0}_{n-n_\ell-m} \end{pmatrix}, \quad \ell = 1, \ldots, p-1, \qquad \text{and} \quad A_p^{(k)} = \begin{pmatrix} I_{m_{p-}} \\ \mathbf{0}_{n-m_p} \\ I_{m_{p+}} \end{pmatrix}, \tag{13}$$

where I_{m_ℓ} is the $m_\ell \times m_\ell$ identity matrix, $I_{m_{p+}}$ and $I_{m_{p-}}$ are submatrices of I_{m_p} which consist of the first $n_p - n_{p-1}$ rows and the remaining m rows of I_{m_p}, respectively, and $\mathbf{0}_q$ is the $q \times m_\ell$ zero matrix for $q = n_{\ell-1}$, $n - n_\ell - m$ or $n - m_p$. In other words, we solve subproblem (9) described in §2.3 with index sets

$$J_\ell = \{n_{\ell-1} + 1, \ldots, n_\ell, n_\ell + 1, \ldots, n_\ell + m\}, \quad \ell = 1, \ldots, p-1,$$

and

$$J_p = \{1, \ldots, m, n_{p-1} + 1, \ldots, n\}$$

in the parallelization phase. Then the ℓth PE is responsible for updating its own $n_\ell - n_{\ell-1}$ variables and m additional variables which are primarily managed by the adjacent $(\ell+1)$th PE. This strategy is expected to be effective for solving large-scale problems in which an element in a block of the variables depends upon only a small number of elements belonging to the blocks contiguous to them. In the numerical experiments, we set

$$n_\ell - n_{\ell-1} = n/p \equiv \overline{n}$$

for all $\ell = 1, \ldots, p$, which means that each PE is primarily responsible for the same number of variables. On the other hand, m out of $m_\ell\,(= \overline{n} + m)$ variables are assigned to at most two PE's. Then we varied the ratio of m to $\overline{n}$ from 0% to 40%. In each case, we examined the performance of the algorithm under various choices of the number p of the subproblems to be solved in the parallelization phase. These p subproblems were solved on p PE's in parallel when $p > 1$. On the other hand, the column $p = 1$ in each table below gives the results obtained by applying the quasi-Newton method to the whole problem on a single PE. Specifically, for $p = 1$, the only transformation matrix $A_1^{(k)}$ given by (13) reduces to the $n \times n$ identity matrix I_n . Note that the parameter p not only specifies the number of PE's used but also affects the algorithmic scheme itself.

In view of the condition (12) in the termination criterion of the synchronization phase, a simple choice of the starting point $\tilde{z}$ would be the unit vector such that $\tilde{z}_\ell = 1$ for $\ell = \tilde{\ell}$, and $\tilde{z}_\ell = 0$ for all $\ell \neq \tilde{\ell}$. In the experiments where p is even, we chose the starting point $\tilde{z}$ in the quasi-Newton method for solving subproblems in the synchronization phase as

$$\begin{aligned} \tilde{z}_0 &= 1 - p/2, \\ \tilde{z}_\ell &= \begin{cases} 1, & \ell = \tilde{\ell},\ \tilde{\ell} \pm 2, \tilde{\ell} \pm 4, \ldots \\ 0, & \ell = \quad \tilde{\ell} \pm 1, \tilde{\ell} \pm 3, \ldots \end{cases}, \qquad 1 \leq \ell \leq p, \end{aligned} \tag{14}$$

where $\tilde{\ell} = \mathrm{argmin}_{1 \leq \ell \leq p} \varphi_\ell^{(k)}(y_\ell^{(k)})$. Then the jth element of the vector $B^{(k)}\tilde{z}$ is given by $x_j^{(k)} + y_j^{(k)}$ if $j \in J_\ell$ for some $\ell = \tilde{\ell}, \tilde{\ell} \pm 2, \tilde{\ell} \pm 4, \ldots$, with $1 \leq \ell \leq p$, and by $x_j^{(k)}$ otherwise. We use the starting point given by (14), because we wish to exploit the information obtained in each PE as much as possible.

Tables 1, 2 and 3 show the performance of the generalized block Jacobi method under the tolerance parameters $\varepsilon_1 = 10^{-3}$ and $\varepsilon_2 = 10^{-6}$, in which the ratio $m/\overline{n}$ was set equal to 0%, 20% and 40%, respectively. In the tables, the number of function evaluations means the cumulative number of those in the parallelization phase on each PE and those in the synchronization phase. Each table reveals that the PVT algorithm on $p\,(> 1)$ PE's is generally more efficient than the quasi-Newton method on a single PE. Moreover, the CPU time decreases as the number p increases except for some problems like Problem 4.1 which can be solved on a single PE relatively fast. From these tables, we may recognize that, for each p, the CPU time gradually increases as the ratio of m to $\overline{n}\,(= n/p)$ becomes large. This is because subproblems

Table 1: Performance of the generalized block Jacobi algorithm when $m = 0$.

Problem	CPU time[sec] (Number of function evaluations)							
	$p = 1$		$p = 2$		$p = 4$		$p = 8$	
4.1	1.336	(167)	0.091	(113)	0.095	(326)	0.096	(579)
4.2	19.57	(3116)	3.082	(4082)	0.797	(6102)	61.19	(166265)
4.3	43.33	(5747)	6.815	(4821)	3.216	(8222)	3.153	(26486)
4.4	5.737	(962)	0.898	(1019)	0.399	(1778)	0.278	(2320)
4.5	27.22	(3641)	13.75	(7210)	7.625	(11487)	1.260	(2771)
4.6	15.07	(1945)	0.356	(423)	0.337	(956)	0.447	(1780)
4.7	51.67	(9365)	0.996	(2555)	0.533	(3368)	0.353	(3387)

in the parallelization phase become large owing to the overlap of the variables. However, the difference in the CPU time with respect to the ratio $m/\overline{n}$ decreases as the number p grows. On the other hand, the algorithm is sometimes unstable when the ratio $m/\overline{n}$ is small, as in the case of $p = 8$ and $m = 0$ for Problem 4.2. Thus the generalized block Jacobi method is effective with moderate overlapping of the variables.

We then made additional experiments in which quasi-Newton iterations for solving subproblems in both the parallelization phase and the synchronization phase were executed at most one iteration. In the experiments, we fixed m at 0. Moreover, we chose the initial point $\tilde{z}$ in the synchronization phase as the unit vector with components $\tilde{z}_\ell = 1$ for $\ell = \tilde{\ell}$, and $\tilde{z}_\ell = 0$ for all $\ell \neq \tilde{\ell}$, where $\tilde{\ell} = \arg\min_{1 \leq \ell \leq p} \varphi_\ell^{(k)}(y_\ell^{(k)})$, so that the resulting $z^{(k)}$ readily satisfies condition (5). Note that, with these choices, the PVT algorithm proceeds in the same manner as the PGD algorithm proposed in [8]. The results are shown in Table 4. The symbol $(-)$ in the table indicates that the algorithm failed to solve the problem within 10000 iterations. Since each iteration of the PGD algorithm consists of the simple procedure that compounds the subvectors of the negative gradient at the current iterate, we may anticipate that the PGD algorithm consumes a large number of iterations. In fact, we see from the tables that the generalized block Jacobi method outperforms the PGD algorithm for the majority of the test problems.

Table 2: Performance of the generalized block Jacobi algorithm when $m/\overline{n} = 0.2$.

Problem	CPU time[sec] (Number of function evaluations)							
	$p = 1$		$p = 2$		$p = 4$		$p = 8$	
4.1	1.336	(167)	0.107	(123)	0.092	(295)	0.137	(896)
4.2	19.57	(3116)	4.715	(5123)	0.912	(7245)	0.460	(7328)
4.3	43.33	(5747)	15.58	(13539)	3.656	(12138)	1.583	(10800)
4.4	5.737	(962)	0.665	(624)	0.276	(976)	0.226	(2225)
4.5	27.22	(3641)	11.93	(6861)	7.818	(14185)	4.474	(16179)
4.6	15.07	(1945)	0.837	(1048)	0.397	(903)	0.577	(1601)
4.7	51.67	(9365)	3.793	(6479)	0.796	(5564)	0.430	(5335)

Finally we compared the generalized block Jacobi method with the unconstrained version of the PVD algorithm originally proposed in [2]. We executed the PVD algorithm as a special case of the PVT algorithm by letting, for each ℓ, $m_\ell = \overline{n} + p - 1$ and

$$A_\ell^{(k)} = \begin{pmatrix} -g_1^{(k)} & & & & & & & \\ & \ddots & & & & & & \\ & & -g_{\ell-1}^{(k)} & & & & & \\ & & & I_{\overline{n}} & & & & \\ & & & & -g_{\ell+1}^{(k)} & & & \\ & & & & & \ddots & & \\ & & & & & & -g_p^{(k)} \end{pmatrix}, \tag{15}$$

where $g_\ell^{(k)} \in \Re^{\overline{n}}$ is the subvector of $\nabla f(x^{(k)})$ with elements $\partial f(x^{(k)})/\partial x_j$, $j = n_{\ell-1}+1, \ldots, n_\ell$. We note that each PE generates a candidate for the next iterate by updating not only its own $\overline{n}$ variables but also all other variables. We chose the starting point $\tilde{z}$ in the quasi-Newton method for solving subproblems in the synchronization phase as the unit vector such that $\tilde{z}_\ell = 1$ for $\ell = \tilde{\ell}$, and $\tilde{z}_\ell = 0$ for all $\ell \neq \tilde{\ell}$. Moreover we set $z_0 = 1 - \sum_{\ell=1}^{p} z_\ell$ in each iteration of the synchronization phase as suggested in [2]. The results are shown in Table 5. Since (15) also implies $A_1^{(k)} = I_n$ for $p = 1$, the results given in the column $p = 1$ of Table 5 are identical

Table 3: Performance of the generalized block Jacobi algorithm when $m/\overline{n} = 0.4$.

Problem	CPU time[sec] (Number of function evaluations)							
	$p = 1$		$p = 2$		$p = 4$		$p = 8$	
4.1	1.336	(167)	0.134	(125)	0.082	(225)	0.110	(793)
4.2	19.57	(3116)	3.461	(2227)	1.223	(7528)	0.596	(10315)
4.3	43.33	(5747)	13.36	(9136)	3.726	(9152)	2.266	(12779)
4.4	5.737	(962)	0.891	(758)	0.328	(1110)	0.238	(1974)
4.5	27.22	(3641)	21.01	(8534)	8.934	(16285)	4.765	(16136)
4.6	15.07	(1945)	1.079	(1144)	0.363	(794)	0.608	(1983)
4.7	51.67	(9365)	4.435	(6598)	1.008	(6622)	0.520	(6857)

to those in the previous tables. It can be seen from the tables that the generalized block Jacobi method tends to be less efficient than the PVD algorithm for small p, but tends to be more efficient for larger p. This is because the cost of function evaluations in the parallelization phase of the PVD algorithm is difficult to reduce even if the number of available parallel processors is large (see the comment at the end of §3.2), although the solution of subproblem (2) in the PVD algorithm may generally provides a good candidate for the next iterate.

5. Conclusion

We have proposed implementation strategies of the PVT algorithm on a vector parallel computer VPP500. In particular, we have demonstrated through numerical experiments that the generalized block Jacobi method, which is a special case of the PVT algorithm, is effective for solving large-scale nonlinear least squares problems. The PVT algorithm can easily be implemented on a VPP500 by simply appending several compiler directive lines together with some global functions to the program written in the ordinary serial programming language FORTRAN 77.

The PVT algorithm constitutes a framework for parallel unconstrained optimization. It is one of the future topics to extend the PVT algorithm to problems with constraints. Finally

Table 4: Performance of the PGD algorithm.

Problem	CPU time[sec] (Number of function evaluations)							
	$p = 1$		$p = 2$		$p = 4$		$p = 8$	
4.1	1.336	(167)	0.312	(469)	0.199	(464)	0.291	(810)
4.2	19.57	(3116)	14.51	(22808)	25.90	(51147)	-	(-)
4.3	43.33	(5747)	-	(-)	-	(-)	-	(-)
4.4	5.737	(962)	2.390	(2810)	2.583	(4286)	4.038	(8006)
4.5	27.22	(3641)	8.784	(3095)	94.14	(48401)	2.660	(1790)
4.6	15.07	(1945)	0.141	(185)	0.241	(468)	1.060	(2461)
4.7	51.67	(9365)	0.159	(258)	0.177	(459)	0.227	(788)

we note that the imbalance of the computational costs on the PE's in the parallelization phase will deteriorate the parallel efficiency of the algorithm. In order to overcome this difficulty, it should be helpful to develop an asynchronous variant of the algorithm. This is another important topic for future research.

References

[1] I. Bongartz, A.R. Conn, N. Gould and Ph.L. Toint, "CUTE: Constrained and unconstrained testing environment," Research Report RC18860, IBM T.J. Watson Research Center, Yorktown, 1993.

[2] M.C. Ferris and O.L. Mangasarian, "Parallel variable distribution," *SIAM J. on Optimization*, vol. 4, pp. 102–126, 1994.

[3] Fujitsu Limited, *Handbook for UXP/V VPP Programming* (in Japanese), Fujitsu Limited: Tokyo, 1997.

[4] M. Fukushima, "Parallel variable transformation in unconstrained optimization," *SIAM J. on Optimization*, to appear.

Table 5: Performance of the PVD algorithm.

Problem	CPU time[sec] (Number of function evaluations)							
	$p = 1$		$p = 2$		$p = 4$		$p = 8$	
4.1	1.336	(167)	0.182	(233)	0.450	(2038)	1.153	(10530)
4.2	19.57	(3116)	1.148	(1747)	1.245	(8585)	4.899	(63552)
4.3	43.33	(5747)	5.902	(4646)	4.216	(9398)	4.306	(25698)
4.4	5.737	(962)	0.584	(694)	0.443	(1299)	0.616	(3670)
4.5	27.22	(3641)	7.620	(2178)	1.857	(831)	2.221	(2470)
4.6	15.07	(1945)	0.322	(450)	0.367	(837)	0.941	(4271)
4.7	51.67	(9365)	2.764	(4944)	0.882	(3024)	0.942	(6688)

[5] S.-P. Han, "Optimization by updated conjugate subspaces," in D.F. Griffiths and G.A. Watson (eds.), *Numerical Analysis: Pitman Research Notes in Mathematics Series 140*, Longman Scientific & Technical: Burnt Mill, England, 1986, pp. 82–97.

[6] A. Hirano, "An introduction to the VPP for MSP users (in Japanese)," *Public Information of KUDPC*, vol. 28, pp. 63–75, 1995.

[7] A. Hirano, "An introduction to parallel programming (in Japanese)," *Public Information of KUDPC*, vol. 28, pp. 116–136, 1995.

[8] O.L. Mangasarian, "Parallel gradient distribution in unconstrained optimization," *SIAM J. on Control and Optimization*, vol. 33, pp. 1916–1925, 1995.

[9] J.J. Moré, B.S. Garbow and K.E. Hillstrom, "Testing unconstrained optimization software," *ACM Trans. Math. Software*, vol. 7, pp. 17–41, 1981.

[10] S.G. Nash, "Newton-type minimization via the Lanczos method," *SIAM J. on Numerical Analysis*, vol. 21, pp.770–788, 1984.

[11] S.S. Oren, "Self-scaling variable metric (SSVM) algorithms Part II: Implementation and experiments," *Management Science*, vol. 20, pp. 863–874, 1974.

[12] J.M. Ortega and W.C. Rheinboldt, *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press: New York, N.Y., 1970.

[13] M.V. Solodov, "New inexact parallel variable distribution algorithms," *Computational Optimization and Applications*, vol. 7, pp. 165–182, 1997.

[14] E. Yamakawa and M. Fukushima, "A block-parallel conjugate gradient method for separable quadratic programming problems," *J. Operations Research Society of Japan*, vol. 39, pp. 407–427, 1996.