EVALUATION OF LARGE-SCALE OPTIMIZATION PROBLEMS ON VECTOR AND PARALLEL ARCHITECTURES*

BRETT M. AVERICK† AND JORGE J. MORÉ‡

Abstract. We examine the importance of problem formulation for the solution of large-scale optimization problems on high-performance architectures. We use limited memory variable metric methods to illustrate performance issues. We show that the performance of these algorithms is drastically affected by application implementation. Model applications are drawn from the MINPACK-2 test problem collection, with numerical results from a super-scalar architecture (IBM RS6000/370), a vector architecture (CRAY-2), and a massively parallel architecture (Intel DELTA).

Key words. optimization, large-scale, limited memory, variable metric, performance evaluation, vector architecture, parallel architecture.

AMS subject classifications. 65Y05, 65Y20, 65K05, 65K10, 90C06, 90C30

1. Introduction. Our aim is to explore performance issues associated with the solution of large-scale optimization problems on high-performance architectures. The solution of these problems, where the number of variables ranges between 10^4 and 10^6, requires computer architectures with fast processors and large memories. We consider three architectures with these features: the IBM RS6000/370 workstation with 16 MW of memory is representative of a super-scalar architecture, the CRAY-2 with 512 MW of memory is representative of vector architectures, and the Intel DELTA (512 mesh-connected i860 processors) with a combined memory of 1024 MW is representative of massively parallel architectures.

We examine the importance of problem formulation for model applications drawn from the MINPACK-2 test problem collection. This collection is representative of large-scale optimization problems arising from applications in elasticity, combustion, lubrication, optimal design, superconductivity, and other fields of interest. Additional information on the MINPACK-2 problems can be found in Averick *et al.* [1].

Performance issues are illustrated with limited memory variable metric algorithms. These methods require minimal storage, lend themselves naturally to vector and parallel implementations, and work well on a wide range of optimization problems. See, for example, the comparisons of Nash and Nocedal [17], and Zou *et al.* [20]. Our implementation of a limited memory variable metric algorithm, VMLM, follows the approach of Liu and Nocedal [12]. In Section 3 we describe the key features of this code and its performance on the model applications of Section 2. The most noticeable aspect of these results is that on all the architectures under consideration, the cost of evaluating functions and gradients dominates the overall solution time. The obvious consequence of this observation is that improvements in performance can be obtained only if the implementation of function and gradient evaluations takes advantage of the architecture.

---

*

† Cooper Neff Technologies, 3 Radnor Corporate Center, Suite 131, Radnor, Pennsylvania, 19087. This work was supported by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CCR-8809615, and by the Army Research Office contract number DAALO3-89-C-0038 with the University of Minnesota Army High Performance Computing Research Center.

‡ Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 60439. Work supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Vectorization issues are treated in Sections 4 and 5. We show that vectorization of the function and gradient evaluation for our applications requires an appropriate partitioning of the element functions. These partitioning techniques are of interest because they are applicable to any partially separable function. In our applications they improve computing times by at least a factor of 10.

Sections 6 and 7 are concerned with distributed memory architectures. Section 6 shows that it is possible to develop algorithms for the function and gradient evaluation that are scalable in the sense that as the number of variables increases and the number of processors increases, the computing time remains reasonably constant. Section 7 shows that the implementation of the VMLM code is scalable in terms of the computing time per iteration. In Section 7 we also compare the performance of the VMLM code on all three architectures by determining when the DELTA is faster than the other two machines.

Performance evaluations of limited memory codes have also been performed by Nocedal [18] and Maier [13]. The issues brought out in these studies differ, in particular, because Nocedal deals only with shared memory architectures, while Maier's evaluation on the SIMD architecture of the CM-200 does not address scalability issues.

We end the paper by discussing possible extensions of this work. In particular, we point out that the performance issues that we have raised are also applicable to truncated Newton methods because for these methods the cost of evaluating functions and gradients is likely to dominate the computing time.

2. Large-Scale Optimization Problems. The optimization problems that we consider arise from the need to minimize a function f of the form

$$f(v) = \int_{\mathcal{D}} \Phi(x, v, \nabla v)\, dx,$$

where $\mathcal{D}$ is some domain in $\mathbb{R}^2$, and Φ is defined by the application. In all cases f is well defined if $v : \mathcal{D} \mapsto \mathbb{R}^p$ belongs to $H^1(\mathcal{D})$, the Hilbert space of functions such that v and $\|\nabla v\|$ belong to $L^2(\mathcal{D})$. This is the proper setting for examining existence and uniqueness questions for the infinite-dimensional problem.

Finite element approximations to these problems are obtained by minimizing f over the space of piecewise linear functions v with values $v_{i,j}$ at $z_{i,j}$, $0 \le i \le n_y + 1$, $0 \le j \le n_x + 1$, where $z_{i,j} \in \mathbb{R}^2$ are the vertices of a triangulation of $\mathcal{D}$ with grid spacings h_x and h_y. The vertices $z_{i,j}$ are chosen to be a regular lattice so that there are n_x and n_y interior grid points in the coordinate directions, respectively.

Lower triangular elements T_L are defined by vertices $z_{i,j}, z_{i+1,j}, z_{i,j+1}$, while upper triangular elements T_U are defined by vertices $z_{i,j}, z_{i-1,j}, z_{i,j-1}$. The values $v_{i,j}$ are obtained by solving the minimization problem

$$\min\left\{\sum \left(f^L_{i,j}(v) + f^U_{i,j}(v)\right) : v \in \mathbb{R}^n\right\},$$

where $f^L_{i,j}$ and $f^U_{i,j}$ are the finite element approximation to the integrals in the elements T_L and T_U, respectively.

Our problems are taken from the collection described in Averick *et al.* [1]. This report contains additional information on these problems; in particular, parameter values are chosen as in this report.

Steady State Combustion (SSC). The study of the steady state in solid fuel ignition models leads to the optimization problem

$$\min\{f_\lambda(v) : v \in H^1_0(\mathcal{D})\},$$

where $f_\lambda : H_0^1(\mathcal{D}) \mapsto \mathbb{R}$ is the functional

$$f_\lambda(v) = \int_{\mathcal{D}} \left\{ \tfrac{1}{2} \|\nabla v(x)\|^2 - \lambda \exp[v(x)] \right\} \, dx,$$

and $\lambda \geq 0$ is a parameter. The finite element approximation is defined by

$$f_{i,j}^L(v) = \frac{h_x h_y}{2} \left\{ \tfrac{1}{2} d_{i,j}^+(v) - \lambda \mu_{i,j}^+(v) \right\}, \qquad f_{i,j}^U(v) = \frac{h_x h_y}{2} \left\{ \tfrac{1}{2} d_{i,j}^-(v) - \lambda \mu_{i,j}^-(v) \right\},$$

where

$$d_{i,j}^\pm(v) = \left(\frac{v_{i\pm 1,j} - v_{i,j}}{h_x} \right)^2 + \left(\frac{v_{i,j\pm 1} - v_{i,j}}{h_y} \right)^2,$$

$$\mu_{i,j}^\pm = \tfrac{1}{3} \left\{ \exp(v_{i,j}) + \exp(v_{i\pm 1,j}) + \exp(v_{i,j\pm 1}) \right\}.$$

We use $\lambda = 2$ for our numerical experiments.

Minimal Surface Area (MSA). The determination of the surface with minimal area and given boundary values in a convex domain $\mathcal{D}$ is an optimization problem of the form

$$\min\{f(v) : v \in K\},$$

where $f : K \mapsto \mathbb{R}$ is the functional

$$f(v) = \int_{\mathcal{D}} \left(1 + \|\nabla v(x)\|^2 \right)^{1/2} \, dx,$$

and the set K is defined by

$$K = \left\{ v \in H^1(\mathcal{D}) : v(x) = v_D(x) \text{ for } x \in \partial\mathcal{D} \right\}$$

for some boundary data function $v_D : \partial\mathcal{D} \mapsto \mathbb{R}$. We consider the Enneper minimal surface defined on $\mathcal{D} = (-\frac{1}{2}, \frac{1}{2}) \times (-\frac{1}{2}, \frac{1}{2})$ by

$$v_D(\xi_1, \xi_2) = u^2 - v^2,$$

where u and v are the unique solutions to the equations

$$\xi_1 = u + uv^2 - \frac{1}{3}u^3, \quad \xi_2 = -v - u^2 v + \frac{1}{3}v^3.$$

The finite element approximation is defined by

$$f_{i,j}^L(v) = \frac{h_x h_y}{2} \left\{ 1 + d_{i,j}^+(v) \right\}^{1/2}, \qquad f_{i,j}^U(v) = \frac{h_x h_y}{2} \left\{ 1 + d_{i,j}^-(v) \right\}^{1/2},$$

where

$$d_{i,j}^\pm(v) = \left(\frac{v_{i\pm 1,j} - v_{i,j}}{h_x} \right)^2 + \left(\frac{v_{i,j\pm 1} - v_{i,j}}{h_y} \right)^2.$$

Optimal Design with Composites (ODC). The optimal design problem requires determining the placement of two elastic materials in the cross-section of a rod

so as to maximize the resulting torsional rigidity. The problem is formulated in terms of a family of problems of the form

$$\min\{f_\lambda(v) : v \in H_0^1(\mathcal{D})\},$$

where $f_\lambda : H_0^1(\mathcal{D}) \mapsto \mathbb{R}$ is the functional

$$f_\lambda(v) = \int_{\mathcal{D}} \left\{ \psi_\lambda\left(\|\nabla v(x)\|\right) + v(x) \right\} dx,$$

and $\psi_\lambda : \mathbb{R} \mapsto \mathbb{R}$ is the piecewise quadratic

$$\psi_\lambda(t) = \begin{cases} \frac{1}{2}\mu_2 t^2, & 0 \le t \le t_1, \\ \mu_2 t_1 (t - \frac{1}{2} t_1), & t_1 \le t \le t_2, \\ \frac{1}{2}\mu_1 (t^2 - t_2^2) + \mu_2 t_1 (t_2 - \frac{1}{2} t_1), & t_2 \le t, \end{cases}$$

with the breakpoints t_1 and t_2 defined by

$$t_1 = \left(2\lambda \frac{\mu_1}{\mu_2}\right)^{\frac{1}{2}}, \qquad t_2 = \left(2\lambda \frac{\mu_2}{\mu_1}\right)^{\frac{1}{2}}.$$

We consider the problem of minimizing f_λ for a fixed value of $\lambda = 0.008$. In our numerical results we used $\mu_1 = 1$ and $\mu_2 = 2$ so that $t_1^2 = \lambda$, $t_2^2 = 2\lambda$. The finite element approximation is defined by

$$f_{i,j}^L(v) = \frac{h_x h_y}{2} \left[\psi_\lambda\left(e_{i,j}^+(v)\right) + \frac{1}{3}(v_{i,j} + v_{i+1,j} + v_{i,j+1}) \right],$$

$$f_{i,j}^U(v) = \frac{h_x h_y}{2} \left[\psi_\lambda\left(e_{i,j}^-(v)\right) + \frac{1}{3}(v_{i,j} + v_{i-1,j} + v_{i,j-1}) \right],$$

where

$$e_{i,j}^{\pm}(v) = \left\{ \left(\frac{v_{i\pm1,j} - v_{i,j}}{h_x} \right)^2 + \left(\frac{v_{i,j\pm1} - v_{i,j}}{h_y} \right)^2 \right\}^{1/2}.$$

3. Limited memory variable metric algorithms. Given an initial iterate x_0, variable metric methods generate iterates by

$$x_{k+1} = x_k - \alpha_k H_k \nabla f(x_k),$$

where $\alpha_k > 0$ is determined by a line search algorithm, and the matrix H_k is determined by an updating procedure. In a limited memory method the matrix H_k is not stored explicitly; instead, it is defined implicitly in terms of a fixed number of vectors.

In our implementation of the limited memory method, the search parameter α_k is determined by the `csrch` subroutine of Moré and Thuente [15]. The matrix H_{k+1} is defined, as in Liu and Nocedal [12], in terms of a scaling parameter σ_k and information gathered during the previous m iterations. Given

$$y_j = \nabla f(x_{j+1}) - \nabla f(x_j), \quad s_j = x_{j+1} - x_j, \quad \rho_j = y_j^T s_j, \qquad k - m + 1 \le j \le k,$$

the product $q = H_{k+1} w$ for any $w \in \mathbb{R}^n$ is computed with the pseudo-code

$q \leftarrow w$
do $i = \min(k, m), \ldots, 1$
  $l \leftarrow i + \max(0, k - m)$
  $\beta_i \leftarrow (s_l^T q)/\rho_l$
  $q \leftarrow q - \beta_i y_l$
end do
$q \leftarrow \sigma q$
do $i = 1, \ldots, \min(k, m)$
  $l \leftarrow i + \max(0, k - m)$
  $q \leftarrow q + s_l \left(\beta_i - (y_l^T q)/\rho_l\right)$
end do

For most applications, the computational cost of limited memory algorithms is dominated by the cost of evaluating $f(x_k)$ and $\nabla f(x_k)$, where we measure cost in terms of floating-point operations, or *flops*. In addition, our limited memory variable metric algorithm, VMLM, costs $(8m + 1)n$ flops for the computation of $H_k \nabla f(x_k)$, and $11n$ flops for the remainder of the algorithm. Memory requirements are $(2m+3)n$ words of storage.

Table 3.1 shows timing results (in seconds) for the solution of the test problems of Section 2 by VMLM with $m = 5$. The most striking feature of these results is the fraction of computing time required for the function evaluations. On one node of the DELTA the function evaluations require 75% of the solution time, on the CRAY-2 the function evaluations are responsible for over 90% of the overall solution time, and on the IBM RS6000 the functions require about 60% of the solution time. We obtain different ratios because the relative cost (in terms of flops) of evaluating the functions differ drastically from machine to machine. We shall address this point in Section 5. Another reason for the difference in ratios on the CRAY-2 is that the VMLM code vectorizes while the function evaluation code does not.

TABLE 3.1
Initial timings (seconds) for $n = 40,000$

	DELTA (1 node)		CRAY-2		IBM RS6000/370	
Problem	function	algorithm	function	algorithm	function	algorithm
MSA	484	641	125	134	74	132
SSC	307	384	132	136	62	91
ODC	510	636	109	116	66	112

It is worthwhile noting that for our problems the number of VMLM iterations tends to grow like $n^{1/2}$. This can be seen clearly in Figure 3.1, where we have plotted the number of iterations against $n^{1/2}$ for all three test problems. We also note that the linear dependence on $n^{1/2}$ for the number of iterations seems to be typical of two-dimensional grid problems. Another interesting observation on these problems, is that in most cases the line search only requires one function evaluation, with the ratio of function evaluations to number of iterations always less than 1.05.

As a consequence of the above remarks, the time to solve grid problems with VMLM grows like $n^{3/2}$ on serial architectures. However, for parallel architectures our aim is to avoid the growth in time resulting from increasing problem size, by increasing the number of processors. By developing a scalable (constant time per iteration) algorithm, the time to solve grid problems with VMLM grows like $n^{1/2}$.

In the remainder of the paper we look at several ways to improve the overall solution time of VMLM. Table 3.1 shows that in order to do this we must concentrate

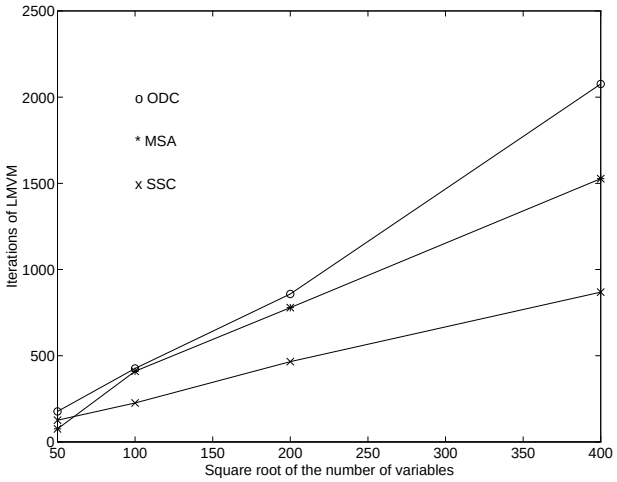

FIG. 3.1. *Number of iterations of VMLM on grid problems as a function of* $n^{1/2}$

on improving the function and gradient evaluations.

4. Evaluating functions and gradients on vector architectures. The results of Section 3 clearly show that improving the performance of the VMLM code on the CRAY-2 requires vectorization of the function and gradient evaluations. In this section we show that this can be done by using partitioning techniques.

The standard way to evaluate the functions and gradients for the optimization problems discussed in Section 2 is to first sweep through the triangular elements with the pseudo-code

```
do j = 1, ..., n_y
   do i = 1, ..., n_x
      Evaluate f^L_{i,j}(v) and ∇f^L_{i,j}(v)
      f(v) ← f(v) + f^L_{i,j}(v)
      ∇f(v) ← ∇f(v) + ∇f^L_{i,j}(v)
   end do
end do
```

followed by similar code for the triangular elements T_U. This code does not vectorize and thus performs poorly on the CRAY-2. The difficulty arises because functions $f^L_{i,j}$ and $f^L_{i+1,j}$ share the variables $v_{i+1,j}$ and thus both contribute to the same gradient element. More specifically, a vector dependency occurs because several elements try to write into the same component of the gradient.

The vector dependency can be eliminated by splitting the functions into groups such that no functions in a group depend on the same variable. This idea was suggested by Moré [14] for treating synchronization issues on shared memory architectures, but can easily be extended to vector architectures.

The *partition* problem of splitting the functions $f^L_{i,j}$ into groups so that functions in a group do not have variables in common is equivalent to partitioning the triangles T_L into groups so that triangles in a group do not intersect. For our test problems, if

$$S = \{(i,j) : 0 \le i \le n_x,\ 0 \le j \le n_y\}\,,$$

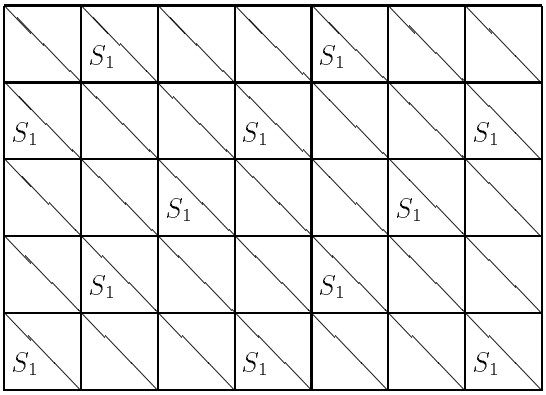

FIG. 4.1. *Triangular elements T_L in S_1*

then the three sets

$$S_1 = \{(i,j) \in S : i = mod(j,3) + 3k,\ k = 0, 1, \ldots\},$$
$$S_2 = \{(i,j) \in S : i = mod(j+1,3) + 3k,\ k = 0, 1, \ldots\},$$
$$S_3 = \{(i,j) \in S : i = mod(j+2,3) + 3k,\ k = 0, 1, \ldots\},$$

define a suitable partition. For example, the triangular elements T_L associated with S_1 are marked in Figure 4.1.

The approach that we have outlined partitions the functions $f_{i,j}^L, f_{i,j}^U$ into six groups. This is the smallest number of groups as long as one of the variables is shared by six functions; in our case, all the variables appear in six functions. A standard red-black partitioning into four groups is therefore not possible. Of course, if we reformulate the optimization problem so that a function is associated with each square, then a standard red-black partitioning would be possible.

Table 4.1 shows the improvements that can be obtained on the CRAY-2 with the above partitioning. We used $n = 40,000$ variables on all problems and noted the time t_s to evaluate the standard f and ∇f, and the time t_p to evaluate the partitioned f and ∇f.

TABLE 4.1
Function evaluation times (seconds) on the CRAY-2 for $n = 40,000$

Problem	t_s	t_p
MSA	0.31	0.017
SSC	0.66	0.017
ODC	0.34	0.036

The increase in speed obtained by partitioning varies with each problem. For the MSA problem there is a speedup of 18, which is what we expect when comparing vectorized code with scalar code on the CRAY-2. The SSC problem shows a speedup of 39. This improvement is due not only to the partitioning, but also to the replacement of

$$\mu_{i,j}^{\pm} = \tfrac{1}{3}\{\exp(v_{i,j}) + \exp(v_{i\pm1,j}) + \exp(v_{i,j\pm1})\}$$

on all interior elements by

$$\mu_{i,j}^{\pm} = \exp(v_{i,j}).$$

Since each variable is a member of six elements, this modification does not change the SSC function (boundary values are handled separately). The ODC problem showed a speedup of 9. In this case vectorization of the ψ function computation required a modification to the original code that increased the flop count.

The partitioning approach can be extended to general partially separable functions, that is, functions of the form

$$f(v) = \sum_{k=1}^{n_p} f_k(v),$$

where each f_k depends on only a few of the variables v_i. For general partially separable functions, the partitioning problem can be formulated as a graph coloring problem. For details and references, see Coleman and Moré [5]. Other references in this area include Coleman, Garbow, and Moré [4] on software for the partition problem, Goldfarb and Toint [7] on the partition problem for matrices that arise from finite element approximations, Jones and Plassmann [10] on the use of partitioning techniques for the iterative solution of sparse linear systems, and Averick *et al.* [2] on computing large sparse Jacobian matrices using automatic differentiation.

5. Performance evaluation on vector architectures. We have shown that partitioning techniques can be used to improve the performance of the function and gradient evaluation algorithms on vector architectures. We now examine the performance of the VMLM code on the SSC problem on the basis of two measures: the time per iteration (t_{iter}) and the Mflop rate.

We already noted in Section 3 that the VMLM code requires $(8m + 12)n$ operations per iteration, in addition to the flops needed to evaluate $f(x_k)$ and $\nabla f(x_k)$. For the SSC problem, the evaluation of $f(x_k)$ and $\nabla f(x_k)$ requires $38n$ operations and n evaluations of the exponential. The cost of evaluating the exponential function was determined experimentally by computing the times required to evaluate the SSC problem when a multiplication replaces the exponential function. We found that for our compiler options during these tests, an exponential evaluation costs (approximately) 20 multiplications on the IBM RS6000, 40 multiplications on the CRAY-2, and 130 multiplications on the DELTA.

An implementation of the exponential function is likely to require at least 30 floating point operations. See, for example, the implementation described by Cody and Waite [3]. Thus, the number of operations on the CRAY-2 look about right, while the number for the IBM probably reflects an extremely careful implementation. The DELTA is an experimental architecture, so the implementation of the intrinsic functions may not have received sufficient attention.

Table 5.1 presents the time per iteration (t_{iter}) and the Mflop rate on the IBM RS6000 and the CRAY-2. We have included the IBM RS6000 because these results are indicative of the performance that can be obtained with minimal effort on a high-performance workstation. Note, in particular, that the IBM results do not require partitioning of the function and gradient evaluation algorithms. We do not present results for the problem with $n = 2.56\ 10^6$ because it is too large for the 16 MW memory.

TABLE 5.1
Performance evaluation of the VMLM code

	IBM RS6000/370		CRAY-2	
n	t_{iter}	Mflop	t_{iter}	Mflop
10,000	0.06	17.5	0.01	137
40,000	0.26	17.7	0.04	141
160,000	1.03	17.9	0.15	144
640,000	4.10	18.1	0.62	143
2,560,000	...	...	2.36	145

The performance results in Table 5.1 show that there is a slight increase in speed with increasing vector length and that the time per iteration scales linearly with n. The overall Mflop rates are quite satisfactory. For example, Dongarra and van der Vorst [6] mention that an unpreconditioned conjugate gradient iteration on a sytem of linear equations with $n = 10^6$ variables achieves 21.1 Mflops on an IBM RS6000/550 and 149 Mflops on the CRAY-2.

6. Evaluating functions and gradients on distributed memory architectures. Evaluation of the function and the gradient of a partially separable function on shared memory architectures has been discussed by Moré [14]. The issues are more complicated on distributed memory architectures; the discussion below follows the work of Jones and Plassmann [11].

The first issue that must be considered is that the work must be distributed among the processors so as to obtain good load balancing. In our test problems this amounts to distributing the evaluations of $f_{i,j}^L$ and $f_{i,j}^U$ evenly among the processors. Second, the variables need to be distributed among the processors so that the ratio of computation to communication remains high. That is, if a processor is responsible for computing $f_{i,j}^L$, it should own as many variables required by $f_{i,j}^L$ as possible.

Since the test problems are grid problems and the DELTA is a mesh connected machine, the distribution of variables is fairly straightforward. We simply lay a grid of processors over the rectangular finite element mesh, as shown in Figure 6.1, distributing the grid points as evenly as possible. In this figure each processor owns six variables.

Any element that has all of its vertices within one processor is an *interior* element; those elements with vertices in two or more processors are *shared* elements. Any element that has a vertex that lies on the boundary of the grid is a *boundary element*. It is possible to be a shared-boundary element, but these elements are to be considered shared.

Each processor is responsible for computing the function in all of its interior elements and in those shared elements for which the processor contains the right angle vertex. Processors that contain a vertex of a boundary element compute that boundary elements contributions to the function. Lastly, the function contributions of the bottom left and top right elements are computed by the appropriate processor. After each processor has computed its contribution to the function, the results are added to get the global function value.

Evaluating the gradient is less complicated. Each processor is responsible for computing each of the gradient components corresponding to the variables that lie

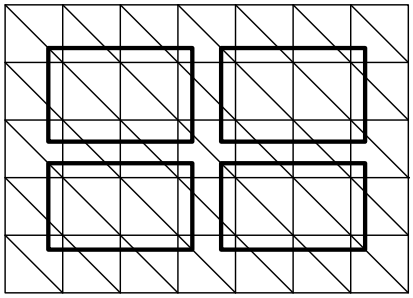

FIG. 6.1. *Partitioning variables into processors*

within the processor. This means that a processor will have to compute the gradient contributions of some shared elements even though the processor does not compute the corresponding function.

Figure 6.2 shows a typical processor with six variables. This processor is responsible for computing the function for all the elements shown. The gridpoints labeled **G**, frequently called *ghostpoints*, are needed to compute the contribution to the function and the component of the gradient for the variables owned by the processor. The ghostpoints must be obtained from the processors directly above, directly below, immediately to the right, and immediately to the left. These data must be communicated to the processor before it can compute the functions or gradient components that depend on the ghostpoints. Communication is handled by the *Chameleon* parallel programming tools developed by Gropp and Smith [8]. These tools can be used to develop code that is portable to a variety of computing environments.

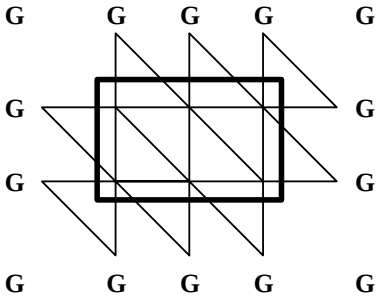

FIG. 6.2. *Processor with ghostpoints*

Let m_x and m_y be the number of grid points in the respective coordinate directions owned by the processors so that $\texttt{nvp} = m_x m_y$ is the number of variables stored in each processor. We store the vector of variables x as an $(m_y+2)\times(m_x+2)$ array and the gradient as an $m_y \times m_x$ array. The extra rows and columns in x allow space for the ghostpoints. This strategy worked well for our problems but may be impractical in terms of storage for three-dimensional problems or for elements of high order.

For a fixed problem size the computing time for evaluating the SSC function and gradient depends on the number of variables `nvp` stored in each processor. Figure 6.3 shows the time for `nvp` set to $5,000, 10,000$, and $20,000$. The computing times decrease as `nvp` decreases since more processors are being used; however, it is not always possible or convenient to use all the DELTA processors, so the results for $\texttt{nvp} = 20,000$

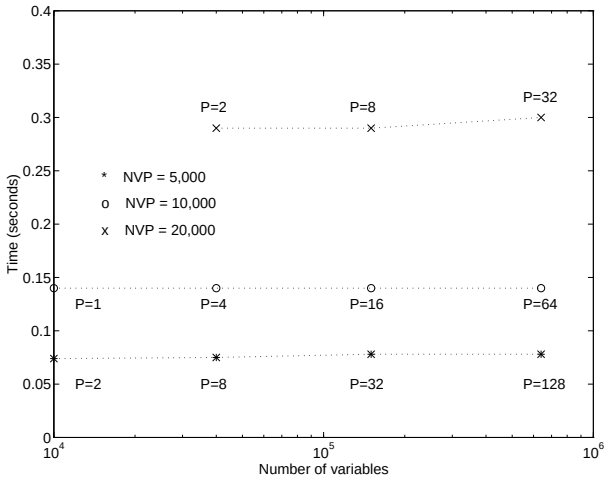

FIG. 6.3. *Evaluation of the SSC function on the DELTA*

are of interest although they do not lead to the fastest computing times.

Figure 6.3 shows that the function and gradient evaluation algorithm is scalable in the sense that as the number of variables increases and the number of processors increases, the computing time remains reasonably constant. Furthermore, the uniformity with which the plots are spaced indicate that the communication time needed to do the evaluation is almost negligible, even for small values of `nvp`.

7. Performance evaluation on parallel architectures. We have noted that the evaluation of the function and gradient on a distributed memory architecture requires that the vector x of variables and the gradient vector g be distributed among the processors. Implementation of the VMLM code also requires that the $2m$ vectors y_i and s_i be distributed among the processors. All other information is local to each processor. With this strategy processors are required to store vectors of length proportional to `nvp`.

Global communication is required to compute the quantities $g^T s$ and $y^T s$. The line search needs $g^T s$, and $y^T s$ is used to determine the scaling parameter. Global communication is also needed to compute $s_i^T q$ and $y_i^T q$ during the computation of the direction d (see the algorithm in Section 3). Thus, each iteration of the VMLM code requires $2(m+1)$ global dot products. These computations are performed with the *Chameleon* package of Gropp and Smith [8].

The line search routine requires very few flops except for function and gradient evaluations, which we have already shown to be amenable to parallel implementation. As a result each processor performs exactly the same line search in our VMLM implementation, and parallelism is obtained through the parallelism in the function and gradient evaluations.

Table 7.1 presents the performance data for the VMLM code with `nvp` = 5000. The time per iteration (t_{iter}) is virtually constant, indicating scalability of performance. There is an increase in the time per iteration for the largest problem; it is generally believed that at the time these results were obtained, the DELTA had a faulty communication channel. Results were also obtained for `nvp` = $10,000$ and `nvp` = $20,000$. The results scale almost linearly with those shown in Table 7.1. For

TABLE 7.1
Performance evaluation of VMLM on the DELTA ($\texttt{nvp} = 5,000$)

n	p	t_{iter}	Mflop/p	Mflops
10,000	2	0.12	9.3	18
40,000	8	0.13	8.8	71
160,000	32	0.13	8.3	267
640,000	128	0.14	8.1	1044
2,560,000	512	0.18	6.3	3233

example, the average time per iteration increases from 0.14 for $\texttt{nvp} = 5,000$ to 0.26 for $\texttt{nvp} = 10,000$, and to 0.50 for $\texttt{nvp} = 20,000$.

Figure 7.1 compares the results in Table 7.1 with those presented in Table 5.1 for the IBM RS6000 and the CRAY-2. We show DELTA results for $\texttt{nvp} = 5,000$ and $\texttt{nvp} = 10,000$. Initially both sequential machines are faster than the DELTA because of processor speed, but this situation is reversed as the problem size increases and the DELTA adds processors. First consider the case when $\texttt{nvp} = 5,000$. The DELTA (4 processors) becomes faster than the IBM RS6000 at $n = 20,000$, while the DELTA (28 processors) becomes faster than the CRAY-2 at $n = 140,000$. The crossing point for other values of nvp can be determined from this data because computing times scale linearly with n and nvp. For example, with $\texttt{nvp} = 10,000$, the DELTA (28 processors) becomes faster than the CRAY-2 at $n = 280,000$.

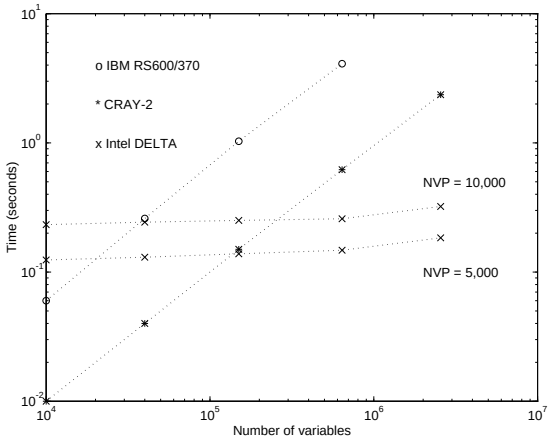

FIG. 7.1. *Time (seconds) per iteration of VMLM as a function of n*

8. Concluding remarks. The performance issues that we have raised apply to truncated Newton methods because for these methods the cost of evaluating the function and gradient is likely to dominate the computing time. Nash and Nodedal [17] mention that a typical cost of the TN code of Nash [16] is $325n$ flops per iteration. This cost is certainly higher than for limited memory variable metric methods, but we still expect it to be a small fraction of the total computing cost for many applications. The cost of the truncated Newton method of Schlick and Fogelson [19]) depends on

TABLE 8.1
Evaluation of $\nabla^2 f(x)v$ (seconds) for the SSC problem on the CRAY-2

n	t_s	t_p
10,000	.18	.004
40,000	.70	.015
160,000	2.79	.061
640,000	11.20	.244

the preconditioner, so the situation is less clear with this code.

Although we have concentrated on issues involving on the evaluation of the function and gradient, many of these issues also apply to the evaluation of the Hessian-vector product $\nabla^2 f(x)v$. This product is often required by, for example, truncated Newton methods. In most codes, this product is approximated by differences of gradient values, but it is certainly possible and desirable to use the actual Hessian-vector product, since this eliminates the dependence of the code on a difference parameter.

The MINPACK-2 test problems include code for the evaluation of $\nabla^2 f(x)v$. These codes raise the same issues as the gradient evaluation codes. In particular, on vector architectures these codes have the same vector dependencies as the gradient evaluation codes. These dependencies can be eliminated by the same partitioning techniques discussed in Section 4. Table 8.1 shows the improvements that can be obtained with these partitioning techniques. We used the SSC problem with $n = 40,000$ variables and noted the time t_s to evaluate the standard $\nabla^2 f(x)v$, and the time t_p to evaluate the partitioned $\nabla^2 f(x)v$.

Note that the computing time scales linearly with n. Moreover, note that Table 4.1 shows that the cost of evaluating the function and the gradient is about the same as the cost of evaluating the Hessian-vector product. This is typical for our applications, and should hold in all cases. For a discussion of this point, in connection with automatic differentiation, see Iri [9].

As a final remark we note that the evaluation of Hessian-vector products on distributed memory architectures can also be done with the techniques discussed in Section 6, and that results were obtained for the SSC problem that were similar to those shown in Figure 6.3.

Acknowledgments. We thank Bill Gropp, Mark Jones, Barry Smith, and Paul Plassmann for sharing their parallel programming expertise, and Peter Tang for his observations on algorithms for evaluating intrinsic functions. Michelle Hribar deserves a special thanks for providing much of the initial programming for this project.

REFERENCES

[1] B. M. AVERICK, R. G. CARTER, J. J. MORÉ, AND G.-L. XUE, *The MINPACK-2 test problem collection*, Preprint MCS-P153-0692, Argonne National Laboratory, Argonne, Illinois, 1992.

[2] B. M. AVERICK, J. J. MORÉ, C. H. BISCHOF, A. CARLE, AND A. GRIEWANK, *Computing large sparse Jacobian matrices using automatic differentiation*, Preprint MCS-P348-0193, Argonne National Laboratory, Argonne, Illinois, 1993.

[3] W. J. CODY AND W. WAITE, *Software Manual for the Elementary Functions*, Prentice Hall, 1980.

[4] T. F. COLEMAN, B. S. GARBOW, AND J. J. MORÉ, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.

[5] T. F. Coleman and J. J. Moré, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 20 (1983), pp. 187–209.

[6] J. J. Dongarra and H. A. van der Vorst, *Performance of various computers using standard sparse linear equation solving techniques*, in Computer Benchmarks, J. J. Dongarra and H. A. van der Vorst, eds., North-Holland, 1994, pp. 177–191.

[7] D. Goldfarb and P. L. Toint, *Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations*, Math. Comp., 43 (1984), pp. 69–88.

[8] W. Gropp and B. Smith, *Users manual for the Chameleon parallel programming tools*, Report ANL-93/23, Argonne National Laboratory, Argonne, Illinois, 1993.

[9] M. Iri, *History of automatic differentiation and rounding error estimation*, in Automatic Differentiation of Algorithms, A. Griewank and G. F. Corliss, eds., Society for Industrial and Applied Mathematics, 1992, pp. 3–16.

[10] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Preprint MCS-P277-1191, Argonne National Laboratory, Argonne, Illinois, 1991.

[11] ———, *Computation of equilibrium vortex structures for type-II superconductors*, Internat. J. Supercomputing Applications, 7 (1993). To appear.

[12] D. C. Liu and J. Nocedal, *On the limited memory BFGS method for large scale optimization*, Math. Programming, 45 (1989), pp. 503–528.

[13] R. S. Maier, *Large-scale minimization on the CM-200*, Optimization Methods and Software, 1 (1992), pp. 55–69.

[14] J. J. Moré, *On the performance of algorithms for large-scale bound constrained problems*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., Society for Industrial and Applied Mathematics, 1991, pp. 31–45.

[15] J. J. Moré and D. J. Thuente, *Line search algorithms with guaranteed sufficient decrease*, Preprint MCS-P330-1092, Argonne National Laboratory, Argonne, Illinois, 1992.

[16] S. G. Nash, *Preconditioning of truncated Newton methods*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 599–616.

[17] S. G. Nash and J. Nocedal, *A numerical study of the limited memory BFGS method and the truncated Newton method for large scale optimization*, SIAM J. Optimization, 1 (1991), pp. 358–372.

[18] J. Nocedal, *The performance of several algorithms for large-scale unconstrained optimization*, in Large-Scale Numerical Optimization, T. F. Coleman and Y. Li, eds., Society for Industrial and Applied Mathematics, 1991, pp. 138–151.

[19] T. Schlick and A. Fogelson, *TNPACK – A truncated Newton minimization package for large-scale problems: I. Algorithms and usage*, ACM Trans. Math. Software, 18 (1992), pp. 46–70.

[20] X. Zou, I. M. Navon, M. Berger, K. H. Phua, T. Schlick, and F. X. Le Dimet, *Numerical experience with limited-memory quasi-Newton and truncated Newton methods*, SIAM J. Optimization, 3 (1993), pp. 582–608.