Parallel Algorithms for Large Scale Nonlinear Unconstrained Optimization

Paul Kang Hoh Phua, Weiguo Fan, Daohua Ming

Email: {phuakh, fanweigu, mingdh}@.iscs.nus.edu.sg

Department of Information Systems & Computer Science
National University of Singapore
Lower Kent Ridge Road
Singapore, 119260

Submitted to ***CrayQuest 97 Singapore Competition***

1. Introduction To Optimization

Though scientific and engineering research problems may differ in physical mechanisms and processes, they usually are common in the final step of seeking quantitative solutions from a set of equations which in most cases are nonlinear equations. This leads to the application of the theory of optimization. Optimization problems can be classified as constrained or unconstrained problems. However, as a constrained optimization problem can be transformed into unconstrained case, majority of the recent research works have been focused on unconstrained optimization problems.

Generally, an unconstrained optimization problem can be defined as follows:

$$\min_{x} f(x) \quad (1.1)$$

Here x is a n-dimensional real vector and f(x) is a nonlinear real valued function. We concern ourselves here that n is large and f is highly nonlinear.

1. Conventional methods for nonlinear optimization

1.1 Introduction to QN Methods

Many approaches for solving unconstrained optimization problems have recently been developed by researchers. Among these new techniques, the quasi-Newton (QN) methods are commonly used by practitioners. Assume that at the *k*th iteration, an approximation point x_k and an $n \times n$ matrix H_k are available, then the methods proceed by generating a sequence of approximation points via the equation

$$x_{k+1} = x_k + \alpha_k d_k \quad (1.1)$$

where $\alpha_k > 0$ is the step-size which is calculated to satisfy certain line search conditions and d_k is an n-dimensional real vector representing the search direction. For QN methods, d_k is defined by:

$$d_k = -H_k g_k \quad (1.2)$$

where $g_k = \nabla f(x_k)$ is the gradient vector of f(x) evaluated at point $x = x_k$.

One important feature of QN methods is the choice of the matrix H_k which is usually chosen to satisfy the QN equation

$$H_{k+1} y_k = \lambda_k \delta_k, \quad (1.3)$$

here, $\delta_k = x_{k+1} - x_k$, $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$, and $\lambda_k > 0$ is a parameter.

Some commonly used QN methods are introduced as follows.

1.2 QN updates

One of the best known QN method is the BFGS method proposed independently by Broyden (1970), Fletcher(1970), Goldfarb (1970) and Shanno (1970). The BFGS update is defined by the following equation

$$H_{k+1} = H_k + \frac{1}{\delta_k^T y_k}((\lambda_k + \frac{y_k^T H_k y_k}{\delta_k^T y_k})\delta_k \delta_k^T - \delta_k y_k^T H_k - H_k y_k \delta_k^T \qquad (1.4)$$

where $\lambda_k \equiv 1$.

Based upon non-quadratic models, Biggs (1973) suggested a self-adjustable value for the parameter λ_k and modified the BFGS update as below:

$$H_{k+1} = H_k + \frac{1}{\delta_k^T y_k}((\lambda_k + \frac{y_k^T H_k y_k}{\delta_k^T y_k})\delta_k \delta_k^T - \delta_k y_k^T H_k - H_k y_k \delta_k^T \qquad (1.5)$$

where,

$$\lambda_k = \frac{1}{t_k} \qquad (1.6)$$

$$t_k = \frac{6}{\delta_k^T y_k}(f(x_k) - f(x_{k+1}) + \delta_k g_{k+1}) - 2 \qquad (1.7)$$

The above update is referred as Biggs' BFGS update thereafter.

Davidon(1968) proposed a method that updates H_k by adding only a symmetric rank-one matrix. This symmetric rank-one (SR1) update is defined by

$$H_{k+1} = H_k + \frac{(\delta_k - H_k y_k)(\delta_k - H_k y_k)^T}{(\delta_k - H_k y_k)^T y_k} \qquad (1.8)$$

1.3 Optimization codes

The BFGS methods was first implemented by Shanno and Phua (1976). This code, called MINIMI, is available in the ACM Mathematical Software Library. Incorporating the conjugate gradient algorithm into their earlier implementation of BFGS methods, Shanno and Phua (1980) developed the CONMIN optimization code which is currently available both in ACM Mathematical Software and IMSL libraries.

Based on the above two works, Buckley and Lenir (1985) developed the BBVSCG optimization code which offers users with the variable storage capability. Liu and

Nocedal (1989) developed the LBFGS optimization code which modifies the BFGS methods with limited-memory capabilities. Another implementation of BFGS methods called E04DGE, is also available in the NAG library, NAG (1992).

2. Parallel Quasi-Newton Algorithms

2.1 Introduction

The parallelization of QN methods has been considered by several researchers in the past decades. The first work was done by Satraeter (1973) for the symmetric rank-one method and it was later modified by van Larrhoven (1985). At each step, this algorithm updates the quasi-Newton matrix along *m* independent directions by parallelly evaluating values of the function and the gradient at *m* points. For a positive definite quadratic function, it can be shown that the method converges in one iteration regardless of the initial starting point. Computational results of Larrhoren (1985) for a set of well-known problems with no more than four variables indicates that the algorithm improves the total number of parallel function evaluations by a speedup factor of 2.5 and the total number of iterations by a speedup factor of 2.5. However, the prerequisite that as many processors as needed are available to perform all the parallel computations, may not be feasible for large problems that arise in many applications. In this report, a class of multi-directional parallel QN methods is presented.

We consider here the class of QN updates (see Shanno (1970)) given by

$$H^{*} = H + t\frac{\delta\delta^{T}}{\delta^{T}y} + \frac{[(1-t)\delta - Hy][(1-t)\delta - Hy]^{T}}{[(1-t)\delta - Hy]^{T}y} \qquad (2.1)$$

where t is a scalar parameter. In equation (3.1), the subscripts have been omitted by dropping (k) and replacing ($k+1$) with the subscript (*). Obviously, the BFGS and the SR1 updates defined by (2.4) and (2.8) correspond to $t = \infty$ and $t = 0$ respectively in (3.1). Consequently, we shall denote the SR1 and BFGS updates by $H^*(0)$ and $H^*(\infty)$ respectively. Analogously the Biggs' update given in (2.5) to (2.7) will be denoted by $H^*(t_k)$.

In practice, we have observed from our numerical experiments and results of many others that a particular QN algorithm may be 'effective' in solving certain types of minimization problems (Phua (1993)). For example, we have noticed that when the SR1 update without modifications is applied to solve a practical problem, it usually has higher efficiency than the other QN updates, such as BFGS (Phua and Chew (1992)), if it is able to solve a problem at all. On another hand, it may encounter difficulties sometimes while other QN updates are able to solve the problem. Subsequent

modifications made to this update may force it to solve the problem, but its efficiency will degenerate.

In order to incorporate the relative merits of different updates into the implementation, we propose parallel algorithms based on QN methods such that when these new algorithms are applied to solve practical problems, the parallel mechanisms will be able to explore different search directions generated by various QN updates during the minimization process. In addition, different search strategies will also be employed simultaneously in the process of locating the line minimum along each direction. These algorithms are described as follows.

2.2 PQN (Parallel quasi-Newton) algorithms

The proposed parallel quasi-Newton (PQN) algorithm consists of the following steps.

1. Initialization

Let $k := 0$, and x_0 be the initial guess of the minimum and $H_0 = I$ be the identity matrix. Set $\varepsilon > 0$ as the required accuracy.

2. Compute the function and gradient values at x_k

Let $f_k := f(x_k)$

and $g_k := \nabla f(x_k)$

3. Compute the parallel search directions

Let $m_1 > 0$ be the number of processors available for computing the search directions in parallel. Compute

$$d_k^{(j)} = -H_k(t_j)g_k, \quad j=1,2, \cdots, m_1 \tag{2.2}$$

For example, t_j can be chosen as 0, 1, ∞ in (3.1), or any other appropriate values.

4. Apply the parallel line search algorithm

Call the line search routine PLS (see below) in parallel along each search direction $d_k^{(j)}$, $j=1,2, \cdots, m_1$. Stop executing this procedure once a line minimum α_k has been found to satisfy the following Wolfe's conditions along any search direction $d_k^{(j)}$:

$$f(x_k + \alpha_k d_k^{(j)}) \le 0.0001 \times \alpha_k g_k^T d_k^{(j)} \tag{2.3}$$

and

$$\nabla f(x_k + \alpha_k d_k^{(j)}) \ge 0.9 \times g_k^T d_k^{(j)} \tag{2.4}$$

Let d_k^* be the search direction that α_k has been found successfully.

In this step, if line minimum points are found from more than one search directions, then d_k^* is chosen to be the search direction that attains the lowest minimum point.

5. Compute the new point

$$x_{k+1} := x_k + \alpha_k d_k^*$$
$$g_{k+1} := \nabla f(x_{k+1})$$

6. Test for convergence

If $\|g_{k+1}\| \le \varepsilon \cdot \max\{1, \|x_{k+1}\|\}$, then stop; otherwise, proceed to Step 7.

7. Compute the new QN updates

Let

$$H_{k+1} := H_{k+1}(\infty)$$

That is to update the approximate inverse Hessian matrix H_k by using the BFGS update defined in (2.5). Repeat the whole process from Step 2.

2.3 Parallel Line Search Routine (PLS)

The parallel line-search procedure works as follows. Let m_2 be the number of parallel processors available for locating the minimum along a particular search direction $d_k^{(j)}$, $j=1,2,\cdots,m_1$. Let α_{max} be the maximum allowable step-size, and denote $\psi(\alpha) = f(x_k + \alpha d_k^{(j)})$. The parallel line search process consists of the following steps:

1. Choose the step sizes

Let

$$0 < \alpha^{(i)} < \alpha_{max},\ i=1,2,\ \cdots,\ m_2$$

where, $\alpha^{(1)} < \alpha^{(2)} < \cdots < \alpha^{(m_2)}$ are m_2 different approximately chosen step sizes. For instance, we may chose $\alpha^{(1)} = 0.5$, $\alpha^{(2)}=1.0$, $\cdots \alpha^{(m_2)} = \alpha_{max}$. Let Φ be the set of these step sizes.

2. Compute the function /gradient values concurrently

For i=1,2, ⋯, m_2, compute concurrently:

$$x_k^{(i)} = x_k + \alpha^{(i)} d_k^{(j)},$$
$$f_k^{(i)} = f(x_k^{(i)}),$$
$$g_k^{(i)} = \nabla f(x_k^{(i)}).$$

3. Test for successful points

Let Φ^* be the set of step sizes $\alpha^{(i)}$ such that for each $\alpha^{(i)} \in \Phi^*$, $\alpha^{(i)}$ satisfies the Wolfe's conditions (3.3)-(3.4). If $\Phi^* \neq \phi$ (empty set) and $\alpha_k^{(i)} \in \Phi^*$ is the stepsize which corresponds to the minimum functional value, that is ,

$$\psi(\alpha_k^{(i)}) = \min_{\alpha^{(i)} \in \Phi^*} f(x_k + \alpha^{(i)} d_k^{(j)})$$

then set $\alpha_k := \alpha_k^{(i)}$ and return to the main PQN routine; otherwise, proceed to Step 4.

4. Choose interpolation points

Let Φ^+ be the set of stepsizes such that for each $\alpha^{(i)} \in \Phi^+$, $\alpha^{(i)}$ satisfies

$$d_k^{(j)T} g_k^{(i)} > 0 .$$

Let $\Phi^- = \Phi - \Phi^+$. Choose $\alpha_1 \in \Phi^-$ such that

$$\psi(\alpha_1) = \min_{\alpha^{(i)} \in \Phi^-} f(x_k + \alpha^{(i)} d_k^{(j)})$$

and choose $\alpha_2 \in \Phi^+$ such that

$$\psi(\alpha_2) = \min_{\alpha^{(i)} \in \Phi^+} f(x_k + \alpha^{(i)} d_k^{(j)})$$

If $\Phi^- = \phi$, then choose $\alpha_1 = 0$. If $\Phi^+ = \phi$, then choose $\alpha_2 = \alpha^{(m)}$, where $\alpha^{(m)} \in \Phi^-$ such that

$$\psi(\alpha^{(m)}) = \min_{\alpha^{(i)} \in \Phi^-} f(x_k + \alpha^{(i)} d_k^{(j)})$$

5. Apply the cubic interpolation technique

Let $\varphi(\alpha)$ be the cubic polynomial passing the two points α_1 and α_2. Let α^* be the minimum of $\varphi(\alpha)$. Compute

$$\varphi(\alpha^*) = f(x_k + \alpha^* d_k^{(j)})$$

and

$$h^* := \nabla\psi(\alpha^*)$$

If α^* satisfies Wolfe's conditions (3.3) -(3.4), then set $\alpha_k = \alpha^*$ and return to the main PQN routine. Otherwise, if $d_k^{(j)T} \cdot \nabla\psi(\alpha^*) > 0$ then replaces α_1 with α^*; if $d_k^{(j)T} \cdot \nabla\psi(\alpha^*) \leq 0$ then replace α_2 with α^*. Repeat Step 5.

3. Vectorization and fine-tuning techniques

3.1 Introduction

To implement the proposed parallel QN methods, we have chosen three updates for computing the following QN directions in step 3 of the PQN routine:

$$d_k^{(1)} = -H_k(0)g_k \tag{3.1}$$

$$d_k^{(2)} = -H_k(\infty)g_k \tag{3.2}$$

$$d_k^{(3)} = -H_k(t_k)g_k \tag{3.3}$$

They are the three search directions generate by SR1, BFGS and Biggs' updates as defined in (2.8), (2.4) and (2.5)-(2.7) respectively. Meanwhile, three line search steps are used for each of the three search directions (m_2 = 3) in the implementation of step 1 of the **PLS** (Parallel Line Search) routine. Here, $\alpha_1 = 0.5$, $\alpha_2 = 1$, and $\alpha_3 = 2$ are used. This particular implementation is referred to as PQN3_3 algorithm.

Originally, the code for implementing our parallel algorithm for nonlinear optimization problems was written in Fortran90 on the CRAY J916 of the NUS computer centre. Since CRAY T94 has only two processors which are insufficient for the implementation of the PQN3_3. Here, the PQ3_3 was converted into the sequential version called PQN33_S. In the sequential version, multi-updates and multi-line-search directions are still applied. However, they are executed on one processor in sequence rather than parallelly on a few processors.

3.2 Numerical experiments on the FORTRAN90 compilers

The Fortran90 installed on Cray T94 is equipped with four switches, namely ***vector, scalar, inline*** and ***task,*** for loop optimization, scalar optimization, subroutine inline, and task parallelization respectively. Each of the four options can be set as an integer from 0 to 3. Usually, value 0 means closing the switch, value 1 means low level usage of the switch, value 2 invokes moderate usage of the switch, and value 3 invokes aggressive usage of the switch. Different combinations of the four switches will be experimented.

To investigate the effects of different options of FORTRAN90 compiles on the computational efficiency of a program, a series of numerical experiments have been conducted with the PQN33_S on CRAY T94.

Table 1. Numerical experiments on Fortran90 compile switches

Test No:	switches:	CPU(Sec.)	CPU-1000 (Sec)
0	default	1109.9	109.9
1	Vector0	failed	NA
2	Vector1	980.7	-19.3
3	Vector2	1122.6	122.6
4	Vector3	1122.7	122.7
5	Scalar0	failed	NA
6	Scalar1	995.1	-4.9
7	Scalar2	1122.8	122.8
8	Scalar3	1108.5	108.5
9	Inline0	1122.7	122.7
10	Inline1	1123.6	123.6
11	Inline2	1122.6	122.6
12	Inline3	1124.5	124.5
13	Vector1+Scalar0	failed	NA
14	Vector1+Scalar1	1004.9	4.9
15	Vector1+Scalar2	983.4	-16.6
16	Vector1+Scalar3	1046.9	46.9
17	Vector2+Scalar0	1107.9	107.9
18	Vector2+Scalar1	1107.9	107.9
19	Vector2+Scalar2	1124.5	124.5
20	Vector2+Scalar3	1107.9	107.9
21	Vector1+Inline0	1048.8	48.8
22	Vector1+Inline1	981.8	-18.2
23	Vector1+Inline2	883.8	-116.2
24	Vector1+Inline3	894	-106
25	Vector3+Inline0	1123	123
26	Vector3+Inline1	1065	65
27	Vector3+Inline2	1065.2	65.2
28	Vector3+Inline3	1065.2	65.2

Table 1 shows the effects of different Fortran compiler switches in executing the program PQN33_S. Tests 1 to 4 are conducted by varying the *vector* switch from 0 to 3 while other switches are set at default. Test 1 with vector0 failed to solve certain problems due to the fact that it exceeded the limit of function/gradient evaluations. Among tests 2 to 4, test 2 of *vector1* required the least CPU time.

Tests 5 to 8 are conducted by varying the *scalar* switch from 0 to 3. Also, test 5 of scalar0 failed to solve certain problems for the same reason as test 1. Test 6 achieved the least CPU time compared with tests 7 and 8.

Tests 9 to 12 are conducted by varying the *inline* switch from 0 to 3. No significant differences are found in these experiments. This shows that the compile option of *inline* does not affect the executing of PQN33_S significantly. The inline option shows

its effect in the group of tests 21 to 24 when *vector1* is used and *scalar* is set at default. Test 23 of (*vector1+inline2*) achieved the least CPU time, being 165 seconds faster than test 21 of (*vector1+inline0*).

In tests 13 to 16, we set the *vector* switch at value = 1, and vary the *scalar* switch from 0 to 3. Test 13 failed due to the fact that it exceeded the maximum allowable function/gradient evaluations. It is found that there are little differences among test 14 to 16. Analogously, in tests 17 to 20, we set the *vector* switch at value = 2, and vary the *scalar* switch from 0 to 3. All the four tests are completed. There are also little differences in this group. However, the average CPU time of tests 14 to 16 is 120 seconds less than the average CPU time of tests 17 to 20. This again shows that *vector1* is faster than *vector2* for the original PQN33_S code.

In tests 21 to 24, we set the *vector* switch at value = 1, and vary the *inline* switch from 0 to 3. Among these four tests, test 21 which requires 1048.8 seconds of CPU time is the slowest, and test 23 which requires 883.8 seconds of CPU time is the fastest.

In tests 25 to 28, we set the *vector* switch at value = 3, and vary the *inline* switch from 0 to 3. Test 25 which requires 1123 seconds of CPU time is the slowest, while tests 26 to 28 require almost the same CPU time of 1065 seconds.

Compared the group of tests 21 to 24 and the group of tests 25 to 28, it is once more shown that *vecotr1* is faster than *vector3* for the original PQN33_S code. The reasons behind these findings remain for further investigations.

In conclusion, various switches have different effects on the numerical performance of out program. For the original PQN33_S code, *vector1* is faster than both *vector2* and *vecotr3*. The compiler switch which achieved the least CPU time among our tests is the combination of *vecotr1+inline2* namely test 23.

3.3 Vectorization Techniques

To find out the utilization of CPU time by different routines in the PQN code, **FLOWVIEW** of the FORTRAN tool on the CRAY T94 is used to analyze the CPU time required by each routine. These results are summarized in Table 2.

Table 2. Profile Statistics Report Showing All Active Modules Sorted by Hit Count (before reconstructing)

Module Name	Hit Count	PCT	ACCUM %
MVM	326975	93.44	93.44
CALFG	17172	4.91	98.35
DB2H	4746	1.36	99.71
PARANU4	711	0.2	99.91
M1NN1	145	0.04	99.95
PLINS	59	0.02	99.97
_second	25	0.01	99.98
write	22	0.01	99.99
others	11	0.01	100

It is found that the subroutine MVM uses up to 93.44% of the CPU time in executing the program PQN33_S. The subroutine MVM is used to perform matrix operations for updating the search direction in the optimization process. The computation is performed using default switches in compiling the program in Fortran90. An example is shown as follows:

```
          Subroutine mvm(n, h, x, y)
          double precision h(n*n), x(n), y(n)
          do 10 i=1,n
          y(i) =0.0
10        continue
         do 20 i=1, n
         loc=i*(i-1)/2
           do 30 j=1, i
           y(i)=y(i)+h(loc+j)*x(j)
           if  (i.ne.j)     y(j)=y(j)+h(loc+j)*x(i)
30   continue
20   continue
```

Here, n is the dimension of the optimization problem, the $(n \times n)$ Hessian matrix H is stored with one dimensional array h(m), m = n(n+1)/2. As the Hessian matrix is symmetric, only the left-upper triangle of this matrix is stored for the calculation.

It is found that the subroutine can not be successfully vectorized as the inner loop contains a conditional **if** statement. To optimize the code, the routine is reconstructed as below:

```
          Subroutine mvm(n, h, x, y)
          double precision h(n*n), x(n), y(n)
          do 10 i=1,n
           loc=i*(i-1)/2
           y(i)=h(loc+i)*x(i)
```

```
            do j=1,i-1
                y(i)=y(i)+h(loc+j)*x(j)
            end do
            do j=1,i-1
                y(j)=y(j)+h(loc+j)*x(i)
            end do
10          continue
```

After the reconstruction, the MVM subroutine is inlined directly into the subroutine PARANU4. Again, **FLOWVIEW** of the FORTRAN90 tool is utilized to find out how the CPU time is used by each subroutine of PQN33_S. The following results are obtained.

Table 3. CPU times consumed by routines of PQN33_S (after reconstructing MVM)

Module Name	Hit Count	PCT	ACCUM %
CALFG	62445	27.07	27.07
PARANU4	43624	18.91	45.98
DB2H	28999	12.57	58.55
%SIN%	26214	11.36	69.92
EXPEP%	22176	9.61	79.53
ALOGEP%	21816	9.46	88.99
RTOR%	19857	8.61	97.6
%ALOGEP%	1050	0.46	98.06
%COS%	932	0.4	98.46
%EXPEP%	667	0.29	98.74
COSS%	541	0.23	98.98
%RTO%R%	516	0.22	99.2
M1NN1	446	0.19	99.4
write	324	0.14	99.54
_second	295	0.13	99.66
PLINS	191	0.08	99.83
others	58	0.27	100

Before reconstructing, the subroutines MVM and PARANU4 used up to 93.64% of the CPU time in total. After reconstructing, the routine MVM is manually inlined into PARANU4. As a result, PARANU4 uses only 18.91% of the total CPU time.

In addition to the above reconstruction of MVM, some other works have also been done to fine-tune the code. These include the replacing of nested structures of (***if then else*** …) statements with ***case*** statements.

After fine-tuning the code, we apply the same compiler option, it is found that the total CPU time for the execution of PQN33_S is reduced remarkably from some 1110 seconds to 115 seconds, which represents a speedup factor of 9.7 in average. The detailed results are shown in Table 4.

Table 4. The total CPU time required by PQN33_S before and after fine-tuning

Compile option	CPU time (sec.) before	CPU time (sec.) after	Speedup factor
default	1109.9	114.7	9.7
vector0	failed	failed	NA
vector1	980.7	112.5	8.7
vector2	1122.6	112.5	10
vector3	1122.7	113.2	9.9
scalar0	failed	3554.9	NA
scalar1	995.1	120.0	8.3
scalar2	1122.8	112.5	10
scalar3	1108.5	120.0	9.2
inline0	1122.7	113.7	9.9
inline1	1123.6	112.8	10
inline2	1122.6	112.8	10
inline3	1124.5	113.7	9.9

It is noted that for compiler option *scalar0*, the computation failed before reconstructing MVM; after reconstructing MVM, the computation completed although it consumed a total of 3554.9 seconds in CPU time.

4. Computational Results

4.1 Test Problems and convergence criteria

To test the performance of the proposed parallel algorithms on CRAY T94, eleven standard functions are chosen. These functions can be found in the papers of Broyden-Toint (1985) and More et. al (1981). Most of these test problems are also included by the MINPACK and CUTE libraries (Bongartz et al 1995). All the functions, except the Wood and Penalty II functions, are tested with variable dimensions of 20, 100, 200, 400, 800, and 1000 respectively. Wood 's function is tested for n = 4. Penalty Function II is tested for n = 20 and 50 respectively. Hence, a total of 57 problems are solved.

Appendices A.1 to A.10 show the optimization process of each of the problems performed by PQN33_S. It can be seen in these appendices that all the optimization problems converge smoothly by utilizing the proposed algorithms. As shown in Appendix A.6, the Broyden-Toint problems of n = 20, 100, 200 and 400 converge within 5 iteration steps, which are the fastest rate of convergence achieved by PQN33_S among all the 57 problems. On other hand, the Broyden-Toint problems of n = 800 and 1000 converge very slowly. They both require more than 1200 iterations to converge, as shown in Appendix A.7. It seems to us that numerical difficulties have been encountered in solving these two problems.

4.2 Analysis of Computational Results

Two serial optimization codes are selected for evaluation purposes. These are the CONMIN, developed by Shanno an Phua (1980), and the E04DGE of the NAG library, see NAG (1992). All codes are evaluated over the same set of 57 test problems. The CONMIN is executed on CRAY T94. As the NAG library is not available on CRAY T94, the E04DGE is executed on the CRAY J916 of the Computer Centre of NUS. In evaluating the performance of each optimization code, the criteria will be based on the total number of iterations, function/gradient evaluations, and the total CPU time required by the code in solving all the test problems.

Computational results obtained by the above optimization codes, namely PQN3_S, CONMIN and E04DGE are summarized in Tables 5 - 7. Table 5 shows the number of iterations and function/gradient evaluations required respectively by PQN33_S, CONMIN and E04DGE for solving these problems. Table 6 provides the differences and the speedups of PQN33_S over CONMIN and E04DGE in terms of total number of iterations and function/gradient evaluations for each problem. Table 7 shows the amount of CPU time used respectively by PQN33_S and CONMIN for each of the test problems. The corresponding speedup factors of PQN33_S over CONMIN are also computed accordingly.

Table 5. Numerical performances of PQN33_S, CONMIN, and E04DGE

Problems:	Number of iterations			Number of function/gradient evaluation		
Rosenbrock	**PQN33_S**	**CONMIN**	**E04DGE**	**PQN33_S**	**CONMIN**	**E04DGE**
n=20	34	34	31	50	50	42
n=100	32	36	28	40	50	50
n=200	34	34	28	46	45	45
n=400	32	34	31	42	45	53
n=800	37	36	28	49	45	54
n=1000	34	35	26	46	46	48
Powell						
n=20	40	51	40	42	52	44
n=100	44	67	64	50	68	73
n=200	45	52	53	49	53	73
n=400	42	69	117	44	70	182
n=800	36	55	64	38	58	102
n=1000	38	49	222	41	53	364
Power						
n=20	41	280	100	42	281	105
n=100	73	867	201	74	868	225
n=200	90	1403	159	91	1404	186
n=400	112	2207	157	113	2208	191
n=800	141	3356	460	142	3357	473
n=1000	151	3857	183	152	3858	209
Watson						
n=20	86	115	709	92	119	715
n=100	154	186	1101	160	195	1138
n=200	147	228	1184	156	233	1257
n=400	221	252	1594	239	265	1749
n=800	265	298	2371	285	313	2656
n=1000	262	failed	2304	278	failed	2678
Broyden-Tridiagonal						
n=20	19	21	31	20	22	40
n=100	21	22	45	22	23	84
n=200	22	25	67	23	26	126
n=400	21	28	90	24	30	176
n=800	22	45	97	28	47	186
n=1000	23	60	98	31	65	187
Subtotal	**2319**	**13802+F**	**11683**	**2509**	**13949+F**	**13511**

Table 5. (continued)

Problems:	Number of iterations			Number of function/gradient evaluation		
Trigometry	**PQN33_S**	**CONMIN**	**E04DGE**	**PQN33_S**	**CONMIN**	**E04DGE**
n=20	37	44	62	43	50	71
n=100	37	42	58	45	52	72
n=200	42	47	65	52	55	79
n=400	39	48	56	47	58	63
n=800	41	58	58	49	67	72
n=1000	42	56	60	50	63	65
Broyden-Toint						
n=20	28	36	61	29	37	72
n=100	41	51	50	42	52	96
n=200	38	47	58	39	48	117
n=400	44	922	85	45	926	174
n=800	1356	1693	1757	1368	1702	2109
n=1000	1714	2061	failed	1722	2071	failed
Wood						
n=4	15	23	37	24	29	44
Hilbert						
n=20	14	30	64	16	31	66
n=100	28	46	168	30	47	172
n=200	27	53	270	30	55	274
n=400	42	62	272	45	64	274
n=800	33	71	279	37	73	285
n=1000	39	70	344	44	72	350
Penalty Function I						
n=20	48	62	498	62	73	729
n=100	49	69	790	59	78	1230
n=200	52	59	140	60	72	147
n=400	48	69	113	58	80	115
n=800	60	58	220	71	71	222
n=1000	47	61	294	55	73	296
Penalty Function II						
n=20	93	307	181	117	359	254
n=50	100	523	113	113	1051	132
Sub-total	**4145**	**6668**	**6153+F**	**4352**	**7409**	**7580+F**
Grand Total:	**6473**	**20470+F**	**17836+F**	**6861**	**21358+F**	**21091+F**

Table 6. Improvements and Speedups of PQN33_S over CONMIN and E04DGE

	In terms of iterations:				In terms of function/gradient evaluations:			
	PQN33_S to CONMIN		PQN33_S to E04DGE		PQN33_S to CONMIN		PQN33_S to E04DGE	
Problems:	**difference**	**speedup**	**difference**	**speedup**	**difference**	**speedup**	**difference**	**speedup**
Rosenbrock								
n=20	0	1.00	3	0.91	0	1.00	-8	0.84
n=100	-4	1.13	4	0.88	-10	1.25	-18	1.25
n=200	0	1.00	6	0.82	1	0.98	-11	0.98
n=400	-2	1.06	1	0.97	-3	1.07	-21	1.26
n=800	1	0.97	9	0.76	4	0.92	-17	1.10
n=1000	-1	1.03	8	0.76	0	1.00	-14	1.04
Powell								
n=20	-11	1.28	0	1.00	-10	1.24	-4	1.05
n=100	-23	1.52	-20	1.45	-18	1.36	-29	1.46
n=200	-7	1.16	-8	1.18	-4	1.08	-28	1.49
n=400	-27	1.64	-75	2.79	-26	1.59	-140	4.14
n=800	-19	1.53	-28	1.78	-20	1.53	-66	2.68
n=1000	-11	1.29	-184	5.84	-12	1.29	-326	8.88
Power								
n=20	-239	6.83	-59	2.44	-239	6.69	-64	2.50
n=100	-794	11.88	-128	2.75	-794	11.73	-152	3.04
n=200	-1313	15.59	-69	1.77	-1313	15.43	-96	2.04
n=400	-2095	19.71	-45	1.40	-2095	19.54	-79	1.69
n=800	-3215	23.80	-319	3.26	-3215	23.64	-332	3.33
n=1000	-3706	25.54	-32	1.21	-3706	25.38	-58	1.38
Watson								
n=20	-29	1.34	-623	8.24	-27	1.29	-629	7.77
n=100	-32	1.21	-947	7.15	-35	1.22	-984	7.11
n=200	-81	1.55	-1037	8.05	-77	1.49	-1110	8.06
n=400	-31	1.14	-1373	7.21	-26	1.11	-1528	7.32
n=800	-33	1.12	-2106	8.95	-28	1.10	-2391	9.32
n=1000	NA	NA	-2042	8.79	NA	NA	-2416	9.63
Broyden-Tridiagonal								
n=20	-2	1.11	-12	1.63	-2	1.10	-21	2.00
n=100	-1	1.05	-24	2.14	-1	1.05	-63	3.82
n=200	-3	1.14	-45	3.05	-3	1.13	-104	5.48
n=400	-7	1.33	-69	4.29	-6	1.25	-155	7.33
n=800	-23	2.05	-75	4.41	-19	1.68	-164	6.64
n=1000	-37	2.61	-75	4.26	-34	2.10	-164	6.03
Sub-total	**-11745**	**133.6**	**-9364**	**-100.1**	**111718**	**131.2**	**-11192**	**120.7**
Average	**-754.7**	**8.62**	**-604.13**	**6.46**	**-756**	**8.47**	**-722.06**	**7.78**

Table 6 (continued)

	In terms of iterations:				In terms of function/gradient evaluations:			
	PQN33_S to CONMIN		PQN33_S to E04DGE		PQN33_S to CONMIN		PQN33_S to E04DGE	
Problems	**difference**	**speedup**	**difference**	**speedup**	**difference**	**speedup**	**difference**	**speedup**
Trigometry								
n=20	-7	1.19	-25	1.68	-7	1.16	-34	1.65
n=100	-5	1.14	-21	1.57	-7	1.16	-35	1.60
n=200	-5	1.12	-23	1.55	-3	1.06	-37	1.52
n=400	-9	1.23	-17	1.44	-11	1.23	-24	1.34
n=800	-17	1.41	-17	1.41	-18	1.37	-31	1.47
n=1000	-14	1.33	-18	1.43	-13	1.26	-23	1.30
Broyden-Toint								
n=20	-8	1.29	-33	2.18	-8	1.28	-44	2.48
n=100	-10	1.24	-9	1.22	-10	1.24	-55	2.29
n=200	-9	1.24	-20	1.53	-9	1.23	-79	3.00
n=400	-878	20.95	-41	1.93	-881	20.58	-130	3.87
n=800	-337	1.25	-401	1.30	-334	1.24	-753	1.54
n=1000	-347	1.20	NA	NA	-349	1.20	NA	NA
Wood								
n=4	-8	1.53	-22	2.47	-5	1.21	-29	1.83
Hilbert								
n=20	-16	2.14	-50	4.57	-15	1.94	-52	4.13
n=100	-18	1.64	-140	6.00	-17	1.57	-144	5.73
n=200	-26	1.96	-243	10.00	-25	1.83	-247	9.13
n=400	-20	1.48	-230	6.48	-19	1.42	-232	6.09
n=800	-38	2.15	-246	8.45	-36	1.97	-252	7.70
n=1000	-31	1.79	-305	8.82	-28	1.64	-311	7.95
Penalty Fun I								
n=20	-14	1.29	-450	10.38	-11	1.18	-681	11.76
n=100	-20	1.41	-741	16.12	-19	1.32	-1181	20.85
n=200	-7	1.13	-88	2.69	-12	1.20	-95	2.45
n=400	-21	1.44	-65	2.35	-22	1.38	-67	1.98
n=800	2	0.97	-160	3.67	0	1.00	-162	3.13
n=1000	-14	1.30	-247	6.26	-18	1.33	-249	5.38
Penalty Fun II								
n=20	-214	3.30	-88	1.95	-242	3.07	-161	2.17
n=50	-423	5.23	-13	1.13	-938	9.30	-32	1.17
Sub-total	**-2514.0**	**63.35**	**-3713**	**108.58**	**-3057**	**65.37**	**-5145**	**113.51**
Grand total	**-14259**	**196.96**	**-13077**	**208.72**	**-14775**	**-196.61**	**-16332**	**234.17**
Average:	**-250.16**	**3.46**	**-229.42**	**3.66**	**-259.21**	**3.45**	**-286.53**	**4.11**

Table 7. CPU time and Speedup factors

Problem	Dimension	PQN33_S	CONMIN	CPU Speedup
Rosenbrock	n=20	0.0055	0.0091	1.65
	n=100	0.0133	0.1105	8.31
	n=200	0.0292	0.3892	13.33
	n=400	0.0683	1.5094	22.10
	n=800	0.2358	6.3136	26.78
	n=1000	0.3172	9.5564	30.13
Powell	n=20	0.0058	0.0132	2.28
	n=100	0.0178	0.2081	11.69
	n=200	0.0387	0.6012	15.53
	n=400	0.0915	3.11	33.99
	n=800	0.2346	9.7404	41.52
	n=1000	0.3621	13.4911	37.26
Power	n=20	0.0057	0.0733	12.86
	n=100	0.0298	2.7414	91.99
	n=200	0.079	16.583	209.91
	n=400	0.2517	100.4295	399.00
	n=800	0.9481	605.3067	638.44
	n=1000	1.4871	1082.5679	727.97
Watson	n=20	0.0558	0.0468	0.84
	n=100	0.4099	0.7051	1.72
	n=200	0.8027	2.966	3.70
	n=400	2.7438	12.3283	4.49
	n=800	6.8846	49.2613	7.16
	n=1000	8.8798	78.4852	8.84
Bro-Tridiagonal	n=20	0.0026	0.0054	2.08
	n=100	0.0079	0.0664	8.41
	n=200	0.0178	0.2831	15.90
	n=400	0.0429	1.2353	28.79
	n=800	0.1352	7.9372	58.71
	n=1000	0.2069	16.5833	80.15
Trigometry	n=20	0.0173	0.016	0.92
	n=100	0.108	0.1669	1.55
	n=200	0.4224	0.6769	1.60
	n=400	1.3043	2.6732	2.05
	n=800	5.3572	12.6359	2.36
	n=1000	8.8715	18.951	2.14
Broyden-Toint	n=20	0.0129	0.0133	1.03
	n=100	0.0793	0.1837	2.32
	n=200	0.1511	0.5898	3.90
	n=400	0.365	43.8602	120.16
	n=800	25.6702	313.2293	12.20
	n=1000	42.7779	592.3554	13.85
Wood	n=4	0.0017	0.0024	1.41
Hilbert	n=20	0.003	0.0085	2.83
	n=100	0.0338	0.1534	4.54
	n=200	0.0849	0.6534	7.70
	n=400	0.3874	2.9269	7.56
	n=800	0.9415	13.1165	13.93
	n=1000	1.6714	20.1001	12.03
Penalty I	n=20	0.0081	0.0119	1.47
	n=100	0.0206	0.2209	10.72
	n=200	0.0468	0.6368	13.61
	n=400	0.1077	2.5616	23.78
	n=800	0.4039	13.5282	33.49
	n=1000	0.4577	16.8636	36.84
Penalty II	n=20	0.0175	0.0829	4.74
	n=50	0.0275	0.4873	17.72
	Total CPU:	**114.7117**	**3079.3634**	**27.07**

4.3 Comparisons, Discussions and Conclusions

4.3.1 Number of iterations and function/gradient evaluations

Table 8. Overall savings and speedups achieved by PQN33_S over CONMIN

	Iteration	Function/gradient evaluations
PQN33_S	6211	6583
CONMIN	20470	21358
Savings	-14259	-14775
Speedup	3.3	3.25
Remarks	56 problems excluding Watson for n = 1000	

Table 9 Overall savings and speedups achieved by PQN33_S over E04DGE

	Iteration	Function/gradient evaluations
PQN33_S	4759	5139
E04DGE	17863	21091
Savings	13077	15952
Speedup	3.75	4.15
Remarks	56 problems excluding Broyden-Toint for n = 1000	

Since CONMIN failed to solve the Watson function for n = 1000 due to the fact that the maximum allowable CPU time was exceeded, only the remaining 56 problems are considered for evaluating PQN33_S and CONMIN algorithms. As shown in Table 8, to solve these 56 test problems, the total numbers of iterations performed by PQN33_S and CONMIN are 6211 and 20470 respectively, being a difference of 14259 iterations, which represents an overall speedup of 3.30. In terms of total function/gradient evaluations, the savings gained by PQN33_S over CONMIN is 14775, which represents an overall speedup of 3.25.

Among these 56 problems, it is observed that different savings and speedup factors are obtained by PQN33_S over CONMIN for different test problems, as seen in Table 6. The maximum savings gained by PQN33_S happens for the Power function when n = 1000. In this case, a speedup factor of 25.3 is achieved in terms of both total number iterations and total number of function/gradient evaluations, as shown in Figures 1-2.

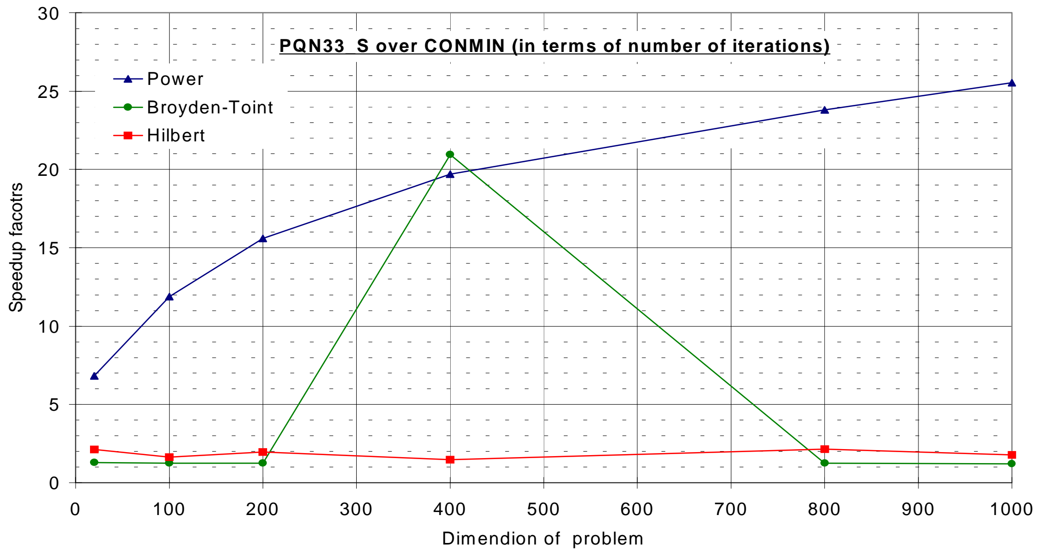


Figure 1 Speedup factors obtained by PQN33_S over CONMIN

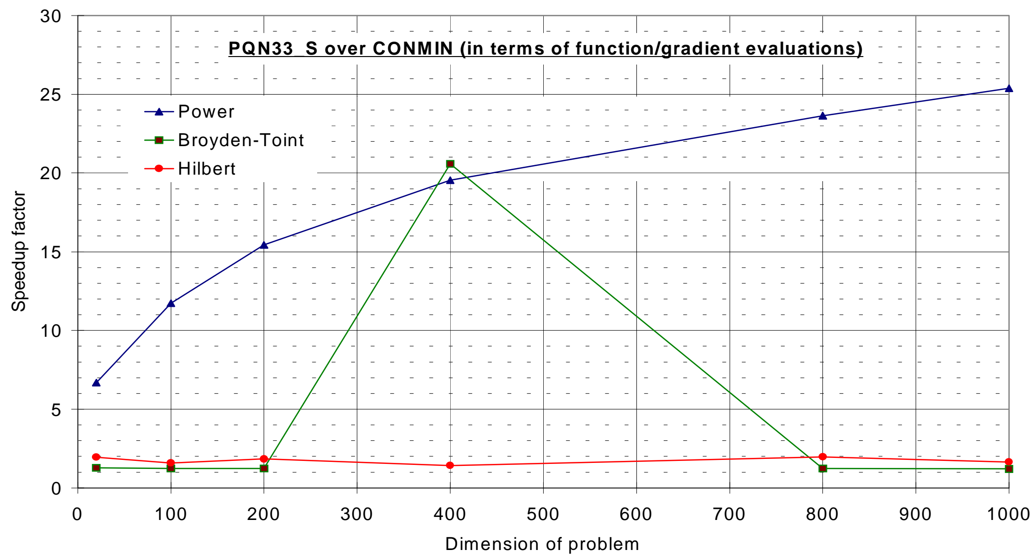


Figure 2 Speedup factors obtained by PQN33_S over CONMIN

E04DGE also failed to solve one problem, namely the Broyden-Toint function of n = 1000, due to internal errors occurred within the package. Therefore, only the remaining 56 problems are available for comparing the performance of PQN33_S and E04DGE. In solving this set of 56 test problems, the total number of iterations performed by PQN33_S and E04DGE are 4759 and 17836 respectively. Hence, a difference in performance of 13077 iterations, which represents an overall speedup factor of 3.75 is obtained by PQN33_S over E04DGE. In terms of total function/gradient evaluations, the savings gained by PQN33_S over E04DGE is 15952, which represents an overall speedup factor of 4.15.

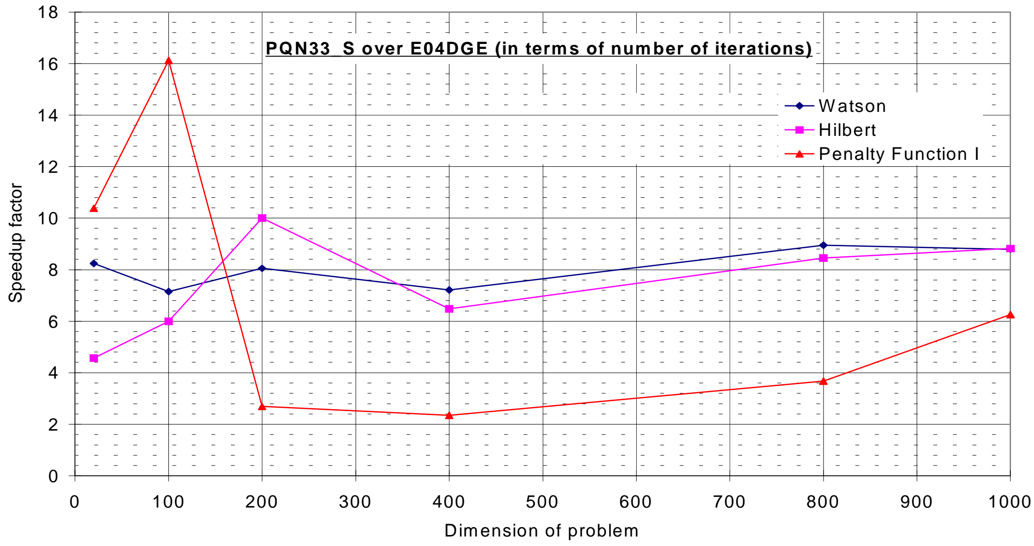


Figure 3 Speedup factors obtained by PQN33_S over E04DGE

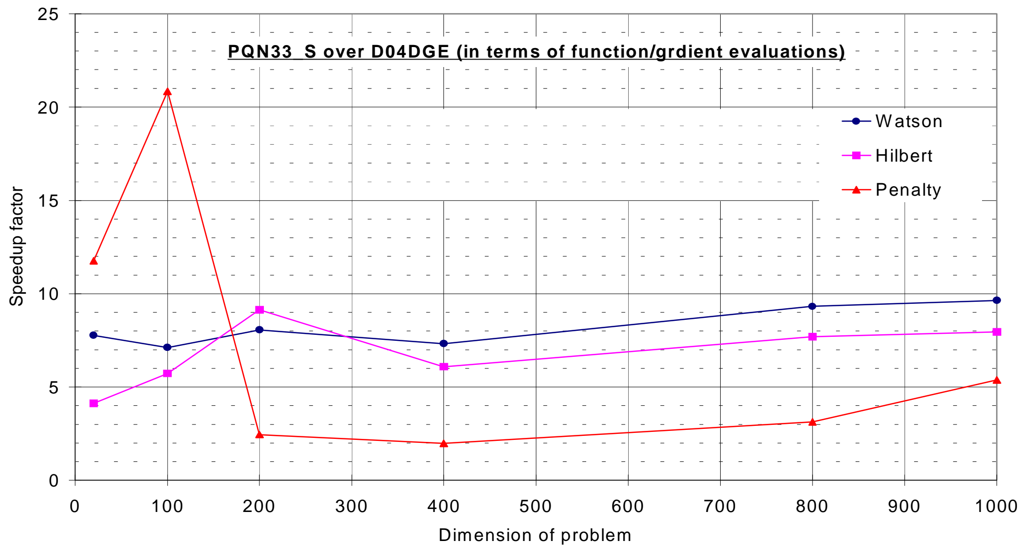


Figure 4 Speedup factors obtained by PQN33_S over E04DGE

Figures 3-4 show the speedup factors of PQN33_S over E04DGE in terms of total number of iterations and total number of function/gradient evaluations for problems of Watson, Hilbert, and Penalty I. For problems of Watson, the speedup factors obtained by PQN33_S are 8 and 7 respectively, based on the total number of iterations and function/gradient evaluations. The highest speedup factor gained by PQN33_S over E04DGE are 16.12 (in terms of total number of iterations) and 20.85 (in terms of function/gradient evaluations) which happens for the case of Penalty I when n = 100.

4.3.2 CPU time

Table 7 shows the amount of CPU time required by PQN33_S and CONMIN optimization codes in solving all the test problems. The speedup factors achieved by PQN33_S over CONMIN (in terms of CPU time) are shown in Figures 5 and 6.

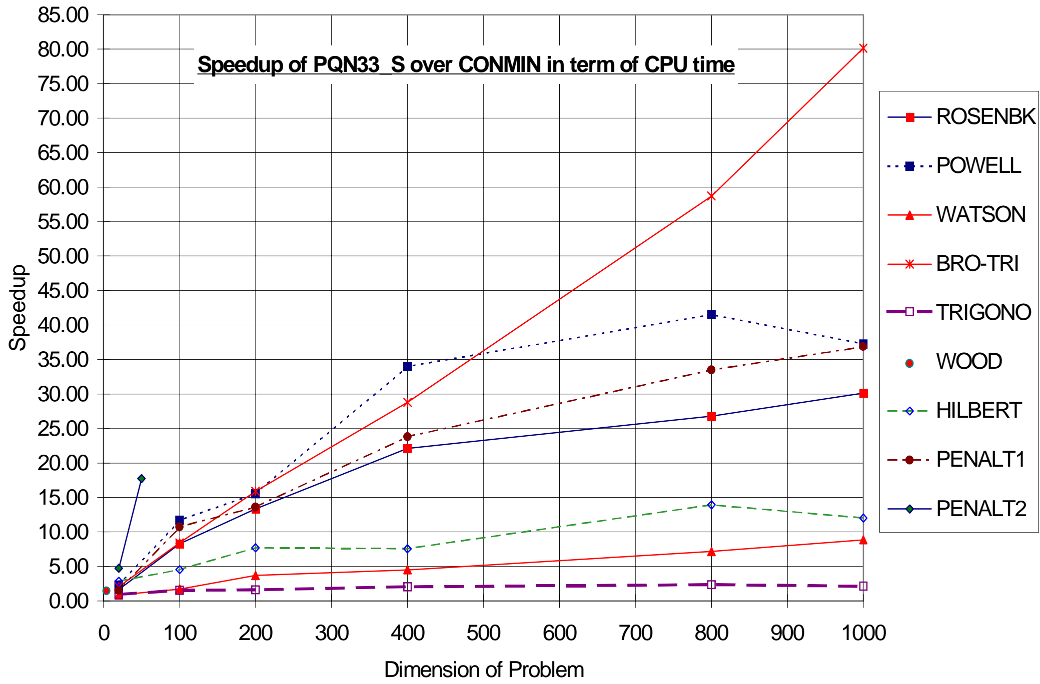


Figure 5 CPU time Speedup factors obtained by PQN33_S over CONMIN

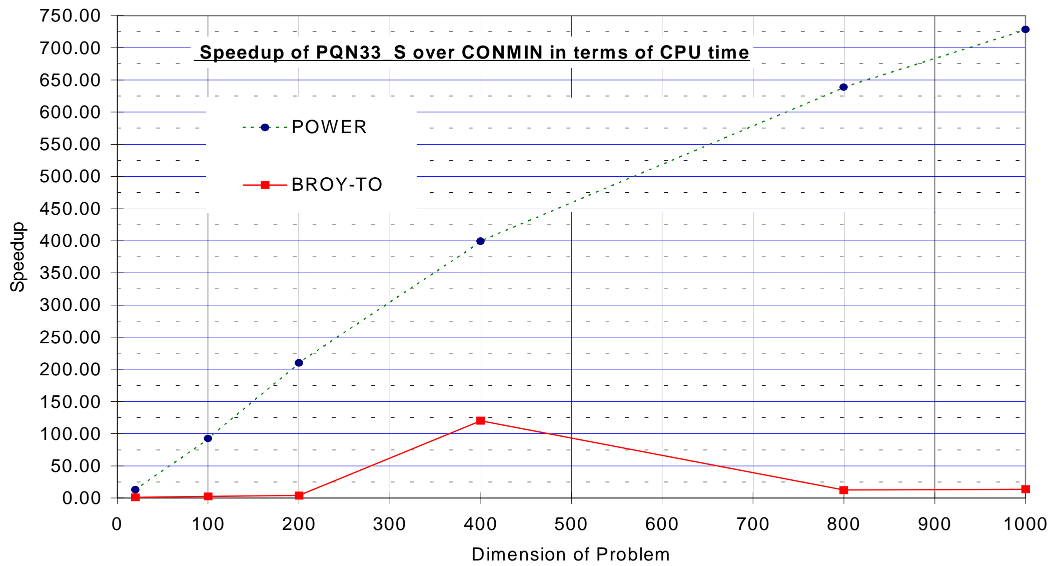
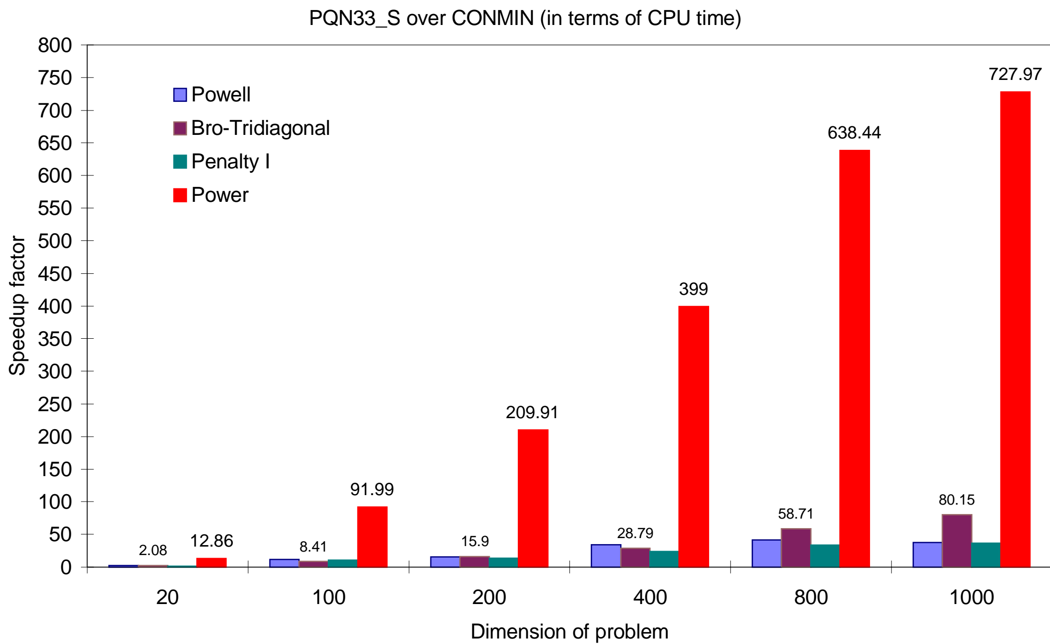


Figure 6 CPU time Speedup factors obtained by PQN33_S over CONMIN

Two figures are used due to the differences in scale. To highlight the results, four functions, namely Powell, Power, Broyden-Trigonometry, and Penalty Function I, are chosen for further investigations. Figure 7 shows the speedup factors gained by PQN33_S over CONMIN for the four problems. In terms of total CPU time required, the average speedup factors obtained by PQN33_S over CONMIN for the four problems is 183, and the maximum speedup factor is 728.

Figure 7 Speedup factors obtained by PQN33_S over CONMIN



4.3.3 Conclusion

We have proposed here multi-directional parallel algorithms for solving large-scale unconstrained optimization problems based on quasi-Newton methods. Numerical results obtained from a broad class of test problems show that the average speedup factor achieved by our new algorithms is more than **300%** (both in terms of total number of iterations and function/gradient evaluations) when they are compared with some well-known existing optimization codes. The maximum speedup factor obtained by our new algorithms can be as high as **25** times for some test problems.

In terms of total CPU time required for solving all the problems, the average speedup factor obtained by our new algorithms is **27** over the well-known CONMIN package tested with a wide range of 56 test problems. In this case, the maximum speedup factor gained by the new PQN33_S optimization codes is **728**. These significant results are obtained due to fact that we have successfully parallelized and vectorized the optimization codes.

In general, it is noticed that as the size and complexity of the problem increase, greater improvements and savings could be realized by our new algorithms. Our results also indicate that parallel algorithms are efficient and robust in solving large-scale nonlinear optimization problems, especially when they are applied in conjunction with vectorization techniques.

5. Acknowledgment

The authors would like to extend their gratitude to the National Supercomputing Research Center (NSRC) for allowing the utilization of CRAY T94 supercomputer facilities. Their kind assistance for making this project possible is also greatly appreciated.

6. References

1. Biggs M. C., "A note on minimization algorithms which make use of non-quadratic properties of the objective function", J. Inst. Maths Applics., (1973), pp.337-338.
2. Brent R. P., *Algorithms for Minimization without Derivatives*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.
3. Broyden C. G., "The convergence of a class of double-rank minimization algorithms", J. Inst. Maths. Applics., (1970), pp.76-90.
4. Broyden C.G., "Quasi-Newton methods and their application to function minimization", Math. Comp., (1967), pp.368-381.
5. Byrd R.H., Schnabel R.B. and Shultz G. A., "Parallel quasi-Newton methods for unconstrained optimization", Mathematical Programming, (1988), pp.273-306.
6. Davidon W.C., "Variable metric method for minimization", Computer Journal (1968), pp.406-410.
7. Fletcher R., "A new approach to variable metric algorithm", Computer Journal, 13, (1970), pp.392-399.
8. Liu D. C. and Nocedal J. "On the Limited Momory BFGS method ofr Large Scale Optimization", Mathematical Programming, 45, (1989), pp. 503-528
9. Luksan L., "Computational experience with known variable metric updates", Journal of Optimization Theory and Applications, (1994), pp. 27-47.
10. More J.J., Garbow B.S. and Hillstrom K.E., "Testing unconstrained optimization software", ACM Transaction on Mathematical Software, 7, (1981), pp. 17-41.
11. Nag (1992): "Fortran Library Reference manual (Mark14), Numerical Algorithms Group 3", Nag Limited, Oxford OX2-8DR, (1992), United Kingdom.
12. Oren S. S., "Self-scaling variable metric algorithm Part II", Management Science, 20, (1974), pp. 863-874.
13. Oren S.S., "On the selection of parameters in self scaling variable metric algorithms", Math. Prog., (1974), pp. 351-367

14. Oren S.S. and Luenberger D.G., “Self-scaling variable metric (SSVM) algorithms, Part I: Criteria and sufficient conditions for scaling a class of algorithms”, Management Science, (1974), pp.845-874.
15. Oren S.S. and Spedicato E., “Optimal conditioning of self-scaling variable metric algorithms” Math. Prog., (1976), pp. 70-90.
16. Phua K. H., “Switching Algorithms for quasi-Newton Methods”, FSU-SCRI-93T-151,Spuercomputer Computation Research Institute, Florida State University, 1993, 28 pp.
17. Phua K.H., “Multi-Directional Parallel Algorithms for Unconstrained Optimization”, Optimization, (1996), pp. 107-125..
18. Phua K.H. and Chew S.B., “Symmetric rank-one update and quasi-Newton methods”, in: K.H. Phua et al., eds., Optimization Techniques and Applications, World Scientific, Singapore, (1992), pp. 52-63.
19. Phua K.H. and Setiono R., “Multi-step, Multi-directional parallel algorithms for unconstrained optimization (1993)”, in: Phua K.H. et al., eds., Optimization Techniques and Applications, World Scientific, Singapore, (1992), pp. 481-487.
20. Phua K.H. and Zeng Y. L., “Parallel Quasi-Newton Method For large-Scale Optimization”, Technical Report, Dept. of Information Systems & Computer Science, National University of Singapore, (1995).
21. Phua K.H. and Fan W. G., “Parallel Implementation of PQN Algorithm”, Technical Report, Dept. of Information Systems & Computer Science, National University of Singapore, (1996).
22. Powell M.J.D, “How bad are the BFGS and DFP methods when the objective function is quadratic”, Mathematical Programming, (1986), pp. 34-47.
23. Schnabel R.B., “Concurrent function evaluations in local and global optimization”, Computer Methods in Applied Mechanics and Engineering, (1987), pp. 537-552.
24. Shanno D.F., “Conditioning of quasi-Newton methods for function minimization”, Mathematics of Computation, (1970), pp.647-656.
25. Shanno D. F. and Phua K. H. “Algorithm 500- Minimization of Unconstrained Multivariate Functions”, ACM Transactions on Mathematical Software, 2(1), (1976).
26. Shanno D.F. and Phua K.H., “Matrix conditioning and nonlinear optimization”, Mathematical Programming, (1978), pp. 149-160.
27. Shanno D.F and Phua, K.H. “Remark on algorithm 500, a variable method subroutine for unconstrained nonlinear optimization”, ACM Trans. Math. Software, (1980), pp. 618-622.
28. Toint Ph.L., “Some numerical results using a sparse matrix updating formula in unconstrained optimization”, Mathematics of Computation, (1985), pp. 68-81.
29. van Laarhoven P.J.M., “Parallel variable metric algorithms for unconstrained optimization”, Mathematical Programming, (1985), pp. 68-81.