Parallel Algorithms for the BFGS Update on a $\mathcal{PUMA}$ Machine

Tim Oliver
Centre for Mathematical Software Research
University of Liverpool

September 1991

Abstract

A quasi-Newton algorithm using the BFGS update is one of the most widely used unconstrained numerical optimisation algorithms. We describe three parallel algorithms to perform the BFGS update on a local memory MIMD architecture such as $\mathcal{PUMA}$. These algorithms are distinguished by the way in which Hessian information is stored. Cost models are developed for the algorithms and used to compare their performances.

1 Introduction

An iteration, k say, of the quasi-Newton method for unconstrained optimisation may be stated as follows:

Model Algorithm Let x_k be the current estimate of the minimum of the function $f(x)$. Also, let p_k be the current search direction.

Step 1 *Test for convergence.*
Terminate the algorithm if the convergence conditions are satisfied returning x_k as the solution.

Step 2 *Compute a step length, λ_k, along the search direction p_k.*

Step 3 *Update estimate for the minimum.*
Set $x_{k+1} = x_k + \lambda_k p_k$.

Step 4 *Update Hessian approximation and calculate next search direction p_{k+1}.*
Set $B_{k+1} = B_k + U_k$, and $p_{k+1} = -B_{k+1}^{-1} g_{k+1}$.

Step 5 *Set $k = k + 1$ and go to Step 1.*

Steps 2 and 4 account for most of the cost of the algorithm and so we are interested in ways of parallelizing these steps for a local memory, MIMD architecture such as $\mathcal{PUMA}$. In this paper we examine the two linear algebra calculations of Step 4: the update of the Hessian approximation, and the computation of the next search direction. To simplify the notation for the rest of this paper we shall drop the use of subscripts to indicate the iterations k and $k+1$. For a particular iteration we use the convention that the current values of quantities are represented by their name alone while the updated values are represented by their name superscripted with a $*$, for example the current Hessian is represented by B and the updated Hessian by B^*. By convention, the symbol p is used to denote the search direction vector in a quasi-Newton method; this symbol is also used to represent the number of processors used by an algorithm. However it should be clear from the context which meaning is intended.

Once a new point, x^*, is found the Hessian approximation, B, is updated to reflect the new curvature information obtained:

$$B^* = B + U.$$

The update matrix, U, is chosen so that B^* satisfies the *quasi-Newton condition*:

$$B^* s = y,$$

where s is the change in x ($s = x^* - x$), and y is the change in gradient ($y = g^* - g$) in an iteration. Also updates B^* must possess the property of hereditary symmetry. This requires that if B is symmetric then B^* is symmetric. In most quasi-Newton methods B is positive definite (such methods are often called *variable metric methods*).

The simplest update matrix which satisfies these conditions is the rank one update matrix. From this we obtain the symmetric rank one update:

$$B^* = B + \frac{(y - Bs)(y - Bs)^T}{(y - Bs)^T s}.$$

This update has some drawbacks so in general rank two updates are preferred. The most popular rank two updates are the *Davidon-Fletcher-Powell* (DFP) update:

$$B^* = B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s} + (s^T Bs) ww^T,$$

where

$$w = \frac{y}{y^T s} - \frac{Bs}{s^T Bs},$$

and the *Broyden-Fletcher-Goldfarb-Shanno* (BFGS) update:

$$B^* = B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}.$$

It is generally accepted that the most effective of these is the BFGS update, and so this paper considers the performance of parallel BFGS updates on a $\mathcal{PUMA}$ machine. Further details of this and other updates can be found in Fletcher[12, Chapter 3] and Gill, Murray & Wright[4, Section 4.5.2].

The Hessian approximation is used to calculate the search direction for the next iteration. For this reason the combined cost of the update and calculation of the next search direction should be considered when comparing methods.

The next search direction can be computed in two ways, either using the Hessian approximation itself:

$$B^* p^* = -g^*, \tag{1}$$

or by using the inverse Hessian approximation, H^*:

$$p^* = -H^* g^*. \tag{2}$$

Equation 1 involves the solution of a system of linear equations at a cost of $O(n^3)$, whilst using the inverse Hessian in Equation 2 only requires a matrix vector multiply costing $O(n^2)$. However, if the Hessian matrix is stored as Choleski factors, LL^T, then Equation 1 can also be solved in $O(n^2)$. Such considerations lead to four methods for performing the BFGS update which vary depending on how Hessian information is held: storing the Hessian matrix or inverse Hessian matrix, each one factored or unfactored. See Table 1 for a summary of these four methods.

Updating an unfactored inverse Hessian (we shall call this Method I) was the most popular method in early implementations since it avoided the cost of solving a linear system of equations. But, in addition to the costs of the various update methods an important consideration is maintaining positive definiteness of the matrix. It is claimed that it is easier to recognise and correct an indefinite matrix when the Choleski factors are stored and this has lead to many recent implementations updating a factored Hessian (Method II). The factored inverse Hessian update (Method III) has not received much attention due to doubts about its numerical stability. However Byrd, Schnabel & Schultz[1] and others report no significant numerical differences between any of the methods. We have chosen to examine parallel algorithms for these three methods, neglecting the unfactored Hessian update because of its higher sequential cost.

To predict the performance of these algorithms on a $\mathcal{PUMA}$ machine we develop simple cost models. These cost models express the execution time of an algorithm in terms of a set of hardware constants and a set of problem parameters. We define three hardware constants:

	matrix unfactored	matrix factored
B	store full B $B^* = B +$ rank two form Choleski factors 2 triangular solves for p^*	store L where: $B = LL^T$ $J^* = L +$ rank one $Q^* L^{*T} = J^*$ by Jacobi rotations 2 triangular solves for p^*
H	store full H $H^* = H +$ rank two m-v multiply for p^*	store J where: $H = JJ^T$ $J^* = J +$ rank one 2 m-v multiplies for p^*

Table 1: Four methods for implementing the BFGS update

T_f the time taken to perform a REAL32 arithmetic operation. $T_f = 100ns$. All arithmetic operations are included in the total arithmetic cost.

T_s,T_c the total time taken to communicate a message of n REAL32s is given by $T_s + nT_c$. $T_s = 1000ns$, $T_c = 880ns$.

The arithmetic cost simply assumes an average rate of 10 MFlops for numerical operations. Communication rates are more difficult to model and are discussed in [9]. For this paper we assume that the underlying switch network is a three stage cross-bar network as recommended by Hofestädt *et al* [6]. The values for T_s and T_c are derived from [9] assuming a single channel, long message communications model. For simplicity, we permit all components of the communication cost to be overlapped with communication on other links and arithmetic operations; for example we assume that transmitting a vector on one link has the same time cost as transmitting that complete vector on each of the 4 links in parallel. This assumes that the communication engines and memory unit can satisfy these demands. The values of these constants are only approximate; a long message could be partitioned into smaller messages and these transmitted in parallel on multiple channels. This would make better use of the link bandwidth and is modelled with a smaller value for T_c.

In addition to these hardware constants two problem parameters are needed:

n the problem size; in this case the dimensions of the square Hessian matrix.

p the number of processors the algorithm will use.

Algorithms for T8 networks often had two distinct classes of process: a 'master' process and 'slave' processes. The master process was so named because it provided data and accepted results from the slave processes and controlled execution of the parallel algorithm, but did not usually take part in the computation itself. This hierarchy of processes was partly due to the underlying hardware which frequently did not make all four of the links on the master T8 processor available to the user. This prevented the master processor from being included in the grid of slave processors and led to different programs being executed on the two types of processor. On a $\mathcal{PUMA}$ machine we make less distinction between master and slave processes; all p processes are expected to take part in the computation, but one of the processes will still be the source of data and destination for the result. This is possible because the switch network makes it easier for a program to have control of all four links on each processor. Thus, these algorithms use p processes on p processors where one of the processes is the initiator of the algorithm, providing data for all the processes and receiving the result.

Two fundamental communication operations that are used by these algorithms are broadcast and distribute. A broadcast of a vector of m REAL32s from 1 process to $p-1$ others is implemented as follows: the source communicates the vector to four other processes in parallel on its four links. Then each of these five processes (including the source) communicate the data to four more processes in parallel. This continues for $\log_5 p$ steps when all p processes have a copy of the data. The cost for this operation is

$$\log_5 p(T_s + mT_c).$$

The distribution of m REAL32s between p processes (including the source process) consists of $(p-1)/4$ steps in each of which the source process sends m/p different elements to four different processes on the

four links. This has a cost of
$$\frac{p-1}{4}\left(T_s + \frac{m}{p}T_c\right).$$

In the following three sections parallel algorithms for each of the three update methods are described. The paper then finishes with a comparison of the methods in Section 5.

2 Method I: unfactored inverse Hessian update

The unfactored inverse Hessian update may be expressed as:

$$H^* = H + \frac{(s-Hy)s^T + s(s-Hy)^T}{y^Ts} - \frac{(s-Hy)^T yss^T}{(y^Ts)^2}.$$

This is followed by a matrix vector multiplication to find the new search direction:

$$p^* = H^*g^*.$$

Byrd, Schnabel & Schultz[1] present a sequence of operations to perform these two steps which is cheaper than a direct implementation of these equations. We base our algorithm on their method:

Method I (Sequential)

```
1 $t = Hg^*$
2 $z = s - t + p$
3 $\gamma = s^Ty$
4 $\delta = z^Ty$
5 $z' = (z - (\delta/2\gamma)s)/\gamma$
6 $H^* = H + z's^T + s{z'}^T$
7 $\gamma' = s^Tg^*$
8 $\delta' = {z'}^Tg^*$
9 $p^* = t + \gamma'z' + \delta's$
```

Since H is symmetric only its lower or upper triangle need be stored. In this case the total sequential cost of the algorithm is $(4n^2 + 17n - 1)T_f$. If the full matrix is stored the cost of the rank two update is increased and the total cost becomes $(6n^2+15n-1)T_f$. The cost of the method is dominated by the matrix vector multiply of step 1 and rank two update of step 6. Both of these steps involve the inverse Hessian approximation H and so in our parallel algorithm we need to consider the storage of H carefully since the choice of storage will affect the costs of steps 1 and 6.

For a parallel algorithm we will consider storage of both the full matrix H and triangular matrix. The update of step 6 should be cheaper if only a triangle is stored due to the smaller number of elements each processor must update. However, for triangular storage in step 1 a lot of extra communication will be required to give complete rows of H to the processors.

The storage schemes for these two algorithms are thus as follows: the inverse Hessian approximation H is stored either in full or just the upper or lower triangle with rows distributed to the processors. If a triangle only is stored then pairs of rows equi-distant from the central row are given to each processor to try and balance the number of elements held by each process. Both algorithms would probably in practice require enough storage for a full matrix since the triangular matrix algorithm needs temporary storage in step 1 for complete rows of the matrix. Distributing the matrix by columns would increase the cost of step 1 by introducing more communications and so is not considered further. Block storage instead of row or column storage would increase the maximum number of processes that could be used to solve a problem of a given size. However the resulting small granularity would lead to poorer performance as the amount of communication compared with computation on each process is increased.

The vectors p, g^*, s, y, t and z are distributed across the processes with each process storing those elements of the vectors for which it also holds the row of the inverse Hessian H. The vectors z' and p^* overwrite z and p. Four temporary distributed vectors are also required for communication of vector slices of g^*, s and z during steps 1 and 6. We assume that at the start of an iteration of the algorithm y holds the old value of g^* and p holds the old value of p^*, but all other vectors are undefined.

Parallel Method I starts with the process which holds the minimum point, x^*, distributing the vectors g^* and s. This has a cost of $(p-1)(T_s + (n/p)T_c)/2$. The distributed vector y is then calculated from: $y = g^* - y$, costing $(n/p)T_f$.

The next step is the calculation of the distributed vector t. For the full matrix this proceeds in p steps as follows: In each step (except the last) the processes pass segments of g^* to a neighbour in a ring of processes. These segments are held in a temporary vector. In parallel with this each process updates its segment of t using the segment of g^* which it input into another temporary vector in the preceding step. With current estimates of the hardware parameters this hides the communication behind computation except in the last iteration. The total cost for the p steps is:

$$\max\left(T_s + \frac{n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

If instead a triangular matrix is stored then complete rows of the inverse Hessian must be gathered to the processes. Each process must output all elements it holds except elements in columns for which it also holds the row. Assuming an even distribution of elements between the processes then the elements a process outputs are distributed equally between the other processes. Each process must also input the same number of elements from the other processes. If each process has n/p rows of the triangular matrix distributed as described previously we approximate the number of elements held by each process as $n(n+1)/2p$. Of these $(n/p+1)n/2p$ do not need to be output. Hence the number of elements that a process must send to each other process is

$$\frac{1}{p-1}\left(\frac{n(n+1)}{2p} - \frac{n(n/p+1)}{2p}\right) = \frac{n(n-p)}{2p^2}.$$

Each process inputs $(p-1)$ messages and outputs $(p-1)$ messages using all four links giving a communication cost of

$$\frac{p-1}{2}\left(T_s + \frac{n(n-p)}{2p^2}T_c\right).$$

The computation of t then takes place as for the full matrix algorithm above except that only the upper triangular elements of the inverse Hessian are updated. It may be possible to arrange for columns of the matrix to be gathered to the processes at each step of the algorithm thus reducing the storage requirement and perhaps hiding that communication behind the computation, however for simplicity this is not considered here.

The calculation of z which follows does not require any communication since each process can calculate those elements of z for which it holds corresponding elements of s, t and p at a cost of $(2n/p)T_f$.

The two inner product calculations for γ and δ in steps 2 and 3 are executed in parallel, but for simplicity we assume for the cost model that they are executed sequentially. The first stage in the inner product calculation is for all the processes to form the partial inner product for the elements of the vectors which they hold. This costs $(2n/p-1)T_f$. This is followed by $\log_5 p$ steps in each of which groups of 5 processes communicate their partial inner products. One of the processes inputs the partial products from the other 4 processes on its 4 links and sums them with its own partial product. In the next step this process then outputs the new partial product to another process. The communications form a tree structure with 4 branches at each node. Each step i, $i = \log_5 p \ldots 1$, of the operation involves processes on level i of the tree outputting their partial products to their parent process at level $i-1$. In the last step one process forms the complete inner product of the distributed vectors. This communication phase has a cost of $(4T_f + T_s + T_c)\log_5 p$ giving a total cost for a single distributed inner product of

$$(2n/p - 1)T_f + (4T_f + T_s + T_c)\log_5 p.$$

We execute the two inner product calculations so that the results end up at the same process, which is the master process. This process then calculates $\delta/2\gamma$ and broadcasts it, together with γ, to all the processes at a cost of $2T_f + 2(T_s + T_c)\log_5 p$. The calculation of z' follows costing $(3n/p)T_f$.

The inverse Hessian update of step 6 uses the same technique of communication hiding as step 1 does, proceeding in p steps as follows: In each step (except the last) the processes pass segments of z' and s to a neighbour in a ring of processes. These segments are held in two temporary vectors. In parallel with this each process updates its elements of the inverse Hessian using the segments of z' and s which it input into another pair of temporary vectors in the preceding step, along with its own segments of z' and s. For full matrix storage each process updates n^2/p^2 elements of H giving a total cost for the step of

$$\max\left(T_s + \frac{2n}{p}T_c, \frac{4n^2}{p^2}T_f\right)(p-1) + \frac{4n^2}{p^2}T_f.$$

If a lower triangle is stored only approximately half the number of elements are updated at each stage giving a cost of

$$\max\left(T_s + \frac{2n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

The next two operations which calculate γ' and δ' have the same cost as the earlier distributed inner products. These operations are followed by the broadcast of γ' and δ' costing $(T_s + 2T_c)\log_5 p$ ready for the calculation of the next search direction p^*. The distributed vector p^* is calculated at a cost of $(4n/p)T_f$. The distributed partitions of p^* are then collected together at the master process in a similar manner to the broadcast algorithm costing $(p-1)(T_s + (n/p)T_c)/4$.

Figure 1 shows the predicted performance of these methods for a variety of problem sizes n and numbers of processors p. The graphs show for each method the predicted MFlop rate for a given n and p. These rates should be compared with the MFlop rate of a single H1 processor to see the speedup achieved. This paper uses a value of 10 MFlops for a single H1. The efficiency of the parallel algorithm on p processors can be judged by comparing its MFlop rate with the maximum possible MFlop rate delivered by that number of processors.

3 Method II: factored Hessian update

The factored Hessian update is the most widespread sequential BFGS update method. It avoids the large cost of solving a linear system of equations to find the next search direction by storing the Choleski factors LL^T of the Hessian approximation B. Methods have been developed to update the factors directly to the factors of the updated Hessian approximation. The papers by Brodlie *et al* [8] and Gill *et al* [10] give a good introduction to the technique whilst Goldfarb [5] presents details of the most efficient factored update algorithms.

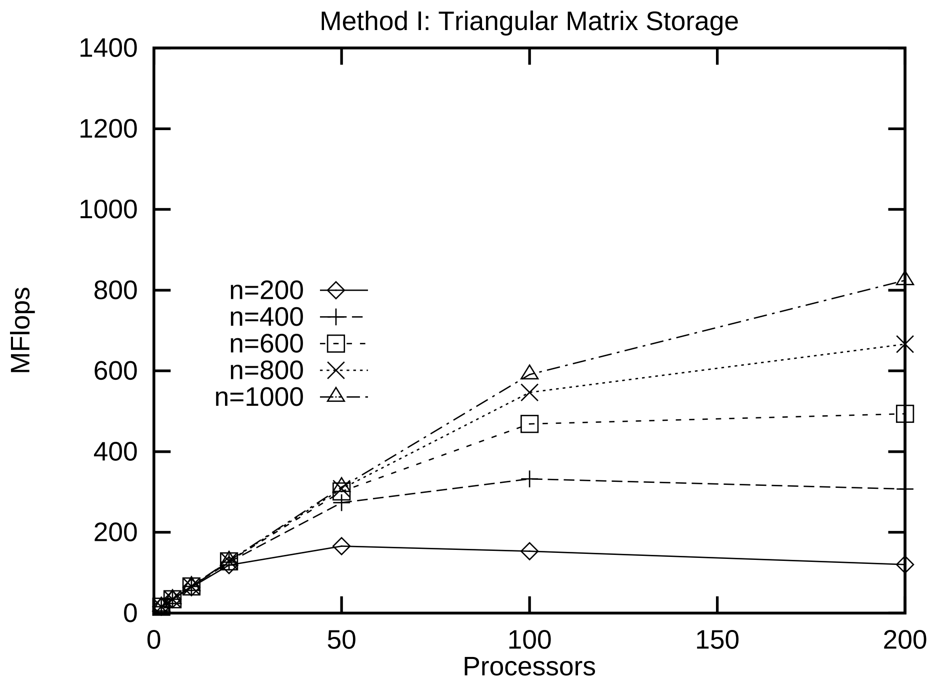
If we store the factors LL^T of B the Hessian update can be expressed as

$$B^* = J^* {J^*}^T, \quad \text{where} \quad J^* = (L + vu^T),$$

with the vectors u and v for the BFGS update given by:

$$\begin{aligned} u &= y - \alpha LL^T s, \quad \text{where} \quad \alpha = \left(\frac{y^T s}{s^T LL^T s}\right)^{1/2}, \\ v &= \frac{L^T s}{\sqrt{y^T s . s^T LL^T s}}. \end{aligned}$$

If we then calculate the factorization $Q^* R^*$ of ${J^*}^T = (R + uv^T)$ the updated Choleski factor L^* is given by ${R^*}^T$. The QR factorization is performed by two sequences of Jacobi rotations. The first $n-1$ Jacobi rotations transform uv^T to the matrix $|u|_2 e_1 v^T$ and R to an upper Hessenberg matrix R^h. The following $n-1$ rotations then transform the matrix $R^h + |u|_2 e_1 v^T$ to the upper triangular matrix R^* which then gives the updated Choleski factor L^*. An implementation of the algorithm does not need to transpose the



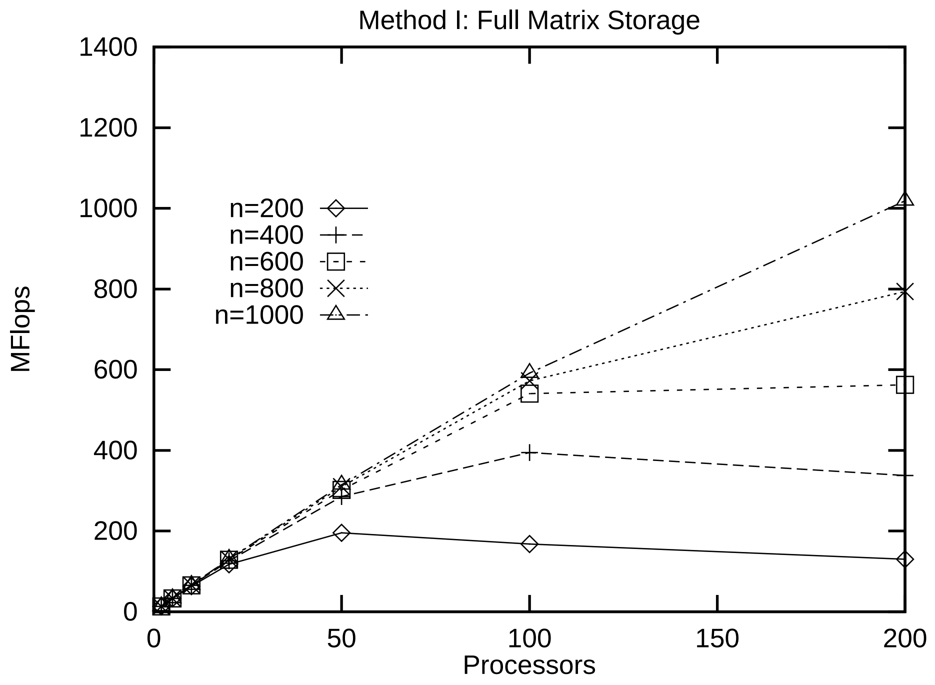


Figure 1: Performance of Method I

matrices L and R^*. Instead, every reference to an element R_{ij} is replaced by a reference to the element L_{ji}. Hence the factored Hessian update method consists of calculating u and v followed by a QR factorization performed by a sequence of Jacobi rotations.

Once the updated Choleski factor L^* is known, the next search direction p^* can be calculated:

$$B^* p^* = -g^* .$$

Since the factors L^* of B^* are stored and not the approximate Hessian itself this equation can be solved by performing two Choleski solves:

$$\begin{aligned} L^* t &= -g^* , \\ {L^*}^T p^* &= t . \end{aligned}$$

The sequential factored Hessian update algorithm which follows is based on Alg A9.4.2. given in [3].

Method II (Sequential)
Calculate u and v

```
1  $\beta = y^T s$
2  $v = L^T s$
3  $\gamma = v^T v$
4  $\alpha = +\sqrt{\frac{\beta}{\gamma}}$
5  $u = y - \alpha L v$
6  $\delta = \frac{1}{\sqrt{\beta\gamma}}$
7  $v = \delta v$
```

QR factorization

```
8  For $i = n-1 \dots 1$
       $J(n, i, u_i, -u_{i+1})$
       $u_i = +\sqrt{u_i^2 + u_{i+1}^2}$
9  For $i = 1 \dots n$
       $L_{i,1} = L_{i,1} + u_1 v_i$
10 For $i = 1 \dots n-1$
       $J(n, i, L_{i,i}, -L_{i,i+1})$
```

Where the Jacobi rotation $J(n, i, a, b)$ is given by:

```
J1. $\alpha = +\sqrt{a^2 + b^2}$
J2. $c = a/\alpha$, $s = b/\alpha$
J3. For $j = i \dots n$
        $\beta = L_{j,i}$
        $\gamma = L_{j,i+1}$
        $L_{j,i} = c\beta - s\gamma$
        $L_{j,i+1} = s\beta + c\gamma$
```

Calculate next search direction: solve $LL^T p^* = g^*$

```
11 $t_1 = {g^*}_1 / L_{1,1}$
12 For $i = 2 \ldots n$
       $t_i = \dfrac{{g^*}_i - \sum_{j=1}^{i-1} L_{i,j} t_j}{L_{i,i}}$
13 ${p^*}_n = t_n / L_{n,n}$
14 For $i = n-1 \ldots 1$
       ${p^*}_i = \dfrac{t_i - \sum_{j=i+1}^{n} L_{j,i} {p^*}_j}{L_{i,i}}$
15 $p^* = -p^*$
```

When calculating the cost of this algorithm we must take careful note of the cost of the square root operations. In the past all floating point operations were implemented in software. Because of this multiplication and division cost significantly more than addition and subtraction, and the square root function itself had a huge cost compared with multiplication. For this reason much effort was expended to design algorithms which required no square root evaluations or at least kept these to a minimum. More recent microprocessor designs incorporate hardware floating point arithmetic units to improve the performance of numerical calculations. With these units the cost of simple arithmetic operations such as add and multiply are comparable. However, even when the square root function has hardware assistance it still has a cost which is larger than the simple operations. In this paper we model the cost of the square root function, T_{sq}, as a multiple of the simple arithmetic operation cost, i.e. $T_{sq} = \alpha T_f$. Using cycle times given in [7] a value of 4 is given to α.

For a sequential implementation of the algorithm, the initial calculation of u and v costs $(2n^2 + 7n + 1)T_f + 2T_{sq}$. The QR factorization, which updates L instead of R, has a cost of $(6n^2 + 18n - 22)T_f + 2(n-1)T_{sq}$. If each negation operation of step 15 costs $1T_f$ then the calculation of the next search direction costs $n(2n+1)T_f$. This gives a total cost for the algorithm of $(10n^2 + 26n - 21)T_f + 2nT_{sq}$.

The first consideration in developing the parallel algorithm is the distributed storage of the Choleski factor L. We have again restricted our consideration to storage by rows or columns, excluding block storage. The expensive steps in the algorithm involving L are steps 2, 5, 8, 10, 12 and 14. Of the two matrix-vector multiplications of steps 2 and 5, one involves L and the other L^T so the combined cost is not affected by the choice of row or column storage. However, it may be worthwhile to store L by *both* rows and columns, which is identical to using the same storage method for both L and L^T. This would allow the more time-efficient matrix-vector algorithm to be used for both steps. The disadvantages of this are the almost doubling of the storage requirement to n^2 and the additional time cost of either communicating the transpose of the matrix or updating both L and L^T. Similarly, the Choleski solves which include steps 12 and 14 use both L and L^T so the same argument applies here too.

In the QR factorization each Jacobi rotation operates on 2 rows of R, that is 2 columns of L. The obvious storage method is thus to store L by rows allowing all processes in parallel to update their elements of the 2 columns within a single Jacobi rotation. This requires only the broadcast of c and s at each rotation. Alternatively, if column storage of L is used then each rotation would require the communication of all elements to be updated from one process to another and then at most two processes performing the update. This very poor parallelism could be improved by beginning the next Jacobi rotation as soon as the essential element from the previous rotation had been calculated, however the large number of communications required for each rotation suggest that this method would not perform as well as using row storage.

For this paper we look at the cost of a row storage algorithm for Method II. The data distribution scheme is as follows: The matrix L has its rows distributed in the same manner as used for the triangular matrix in Method I. Only space for a lower Hessenberg matrix is required giving this algorithm a great storage advantage over the other algorithms presented in this paper which require space for a full matrix. The vectors s, y, u, t, g^* and p^* are distributed with a process holding those elements of the vectors for which it also holds the row of L. Eight temporary vectors of size n/p are required on each process for the summation of vectors in step 2. Finally, an n-vector is stored in full on each process to hold partial results of v in step 2 and to store the partial sums from the Choleski solves in steps 12 and 14. We assume that at the start of an iteration of the algorithm y holds the old value of g^*, but all other vectors are undefined.

The parallel algorithm starts by distributing g^* and s from the master process at a cost of $(p-1)(T_s + (n/p)T_c)/2$. Then y is calculated from: $y = g^* - y$, costing $(n/p)T_f$. The distributed inner product of step 1 stores the result β on the master process and costs

$$(2n/p - 1)T_f + (4T_f + T_s + T_c)\log_5 p.$$

To calculate v each process first forms the partial sums $v_i = \sum_{i=1}^{j} L_{j,i}s_j$ for each row, j, of L that it holds. This costs $(n^2/p + n/p)T_f$. The p n-vectors v are then summed to one process and the result vector distributed to the processes. The summation of vectors proceeds in m steps as follows: In each step, i, a process inputs a segment of the summation vector v beginning at index $(n/m)i$ and of size $d = n/m$ from its 4 children in a tree structure into 4 temporary vectors. In parallel with this the process adds the 4 vector segments input in the last step to the processes own segment at index $(n/m)(i-1)$ and the result segment is output to the parent process. This arrangement allows communications of segments to be executed in parallel with the summation. The cost to get the first segment summed at the root process is $(T_s + dT_c + 4dT_f)\log_5 p$. Reading in the remaining segments and adding them costs $\max(T_s + dT_c, 4dT_f)(m-1)$. The value of m should be chosen which give the least cost; for this paper we use $m = p$. The result vector is then distributed back to the processes at a cost of $(p-1)(T_s + (n/p)T_c)/4$.

Step 3 is another distributed inner product leaving the result on the master process. The master process next calculates and broadcasts α for step 4 costing $T_f + T_{sq} + (T_s + T_c)\log_5 p$.

The matrix vector multiply of step 5 is similar to that in step 1 of Method I. The segments of v are cycled round a ring of processes, and at each step a process updates its elements of a temporary result vector. Assuming that at each step half of the full matrix elements involved in the update are zero then the total cost of the matrix vector multiply is

$$\max\left(T_s + \frac{n}{p}T_c, \frac{n^2}{p^2}T_f\right)(p-1) + \frac{n^2}{p^2}T_f.$$

The rest of the calculation of u in step 5 costs $(2n/p)T_f$.

The calculation and broadcast of δ costs $2T_f + T_{sq} + (T_s + T_c)\log_5 p$. In the final step in the initialization phase each process updates those elements of v for which it also holds the row of L at a cost of $(n/p)T_f$.

The second phase of the algorithm is the QR factorization which is dominated by the Jacobi rotations $J(n,i,a,b)$. We parallelize rotation i as follows: For step 8 the process holding u_{i+1} communicates it to the process holding u_i. Then the process holding u_i calculates c and s and broadcasts these to the other processes. For step 10, the initial communication is not required since one process holds both $L_{i,i}$ and $L_{i,i+1}$. Next each process updates columns i and $i+1$ for those rows which it holds. Assuming the updates to L (step J3) are evenly distributed across processes then the time taken to perform all the updates in step 8 or 10 is $(3n^2 + 3n - 6)T_f/p$. To this we add the cost to calculate and broadcast c and s for $n-1$ Jacobi rotations which is $(n-1)(6T_f + T_{sq} + (T_s + 2T_c)\log_5 p)$ for step 10 and an additional $(n-1)(T_s + T_c)$ for step 8 to communicate u_{i+1}. There is no additional cost to update u_i since its new value was calculated within the Jacobi rotation. Step 9 requires u_1 to be broadcast giving a total cost for that step of $(2n/p)T_f + (T_s + T_c)\log_5 p$.

The calculation of the next search direction p^* in the final phase of the algorithm involves two parallel Choleski solves; one using the matrix L and the next using L^T. These require two different algorithms to solve because L is distributed. For solving $Lt = g^*$ in steps 11 and 12 each process maintains a partial sum $L_{i,j}t_j$ for each row i of L that it holds and this is updated at each iteration as a new element of t is calculated. At iteration i in step 12 we proceed as follows: the process holding row i calculates t_i from its partial sum. This new element is then broadcast and all processes update their partial sums. Assuming that the partial sum updates take similar times on each process the total cost for steps 11 and 12 is

$$((n-1)^2/p + 2n - 1)T_f + (n-1)(T_s + T_c)\log_5 p.$$

In practice we would expect the broadcast of t_i to be overlapped with updating the partial sums which will give some improvement in performance.

The second Choleski solve involves L^T and hence is similar to a Choleski solve with L where L is distributed by columns instead of by rows. For this situation each process maintains a partial sum for every row of L^T. These partial sums hold the values $L_{j,i}{p^*}_j$ for every row i of L^T and for those j for which a process holds row j of L. At the start of iteration i all processes take part in a summation of their partial sums for row i of L^T with the result finishing on the process holding row i of L^T. This process then calculates ${p^*}_i$ and updates its partial sums before the next iteration begins. Since only one process in an iteration updates its partial sums the cost of this parallel algorithm as it stands is the cost of the sequential algorithm *plus* the cost of the distributed summations. In practice the algorithm should be coded such that as soon as the process has updated its sum $i-1$ the next iteration begins with the summation of partial sums $i-1$. In this way many processes may be updating their partial sums at a time as well as communicating for the summation of distributed partial sums. Unfortunately, this appears to be difficult to code since the time taken to perform the partial updates varies as the iterations proceed making efficient synchronisation between the communicating and updating processes difficult to achieve.

It is also very difficult to propose a cost model for this algorithm other than the worst case sequential one. A workable suggestion that will give a rough idea of the cost of this algorithm can be obtained by comparing the cost of performing an update in an iteration with the cost of the summing the partial sums and calculating a new element of p^*. The worst case for an update involves a process updating $n-1$ partial summations at a cost of $2(n-1)T_f$. In the following $p-1$ iterations, since the rows of L are distributed cyclically, this process does not need to perform any other updates. However it does need to take part in the summation of the partial sums for these iterations. It seems reasonable therefore to compare the worst update cost with the cost of the following $p-1$ summations and calculations of ${p^*}_j, j = i-1 \ldots i-p$ which cost $(p-1)((T_s + T_c + 4T_f)\log_5 p + 4T_f)$. With the current values for the hardware parameters the cost of the update is hidden by the cost of communications on medium to large networks ($p \geq 32$). So we assume that the partial sum updates execute in parallel with the communications and finish sooner allowing us to neglect the update cost in the cost model. This gives an estimated cost for steps 13 and 14 of

$$(2n-1)T_f + (n-1)(T_s + T_c + 4T_f)\log_5 p.$$

Finally we negate the vector p^* and gather the full vector to the master process at a cost $(n/p)T_f + (p-1)(T_s + (n/p)T_c)/4$.

Figure 2 shows the predicted performance of this method using the same values for problem parameters as for Method I.

4 Method III: factored inverse Hessian update

This method has not received as much attention as other methods, but has been investigated by Davidon[2] and Powell[11]. As in Method II we express the inverse Hessian update in product form

$$H^* = (I + uv^T)H(I + uv^T)^T.$$

If we then store the factors JJ^T of H the update of the factors is given by

$$J^* = (I + uv^T)J, \quad \text{where,} \quad \begin{array}{l} u = \frac{s}{s^T y}, \\ v = +\sqrt{\frac{s^T y}{s^T H^{-1} s}}H^{-1}s - y. \end{array}$$

This can be simplified since $H^{-1}s = -\lambda g$ and $sH^{-1}s = -\lambda sg$ where λ is the step length used in this iteration of the quasi-Newton algorithm. Once the factors have been updated a new search direction is calculated by performing two matrix vector multiplications:

$$p^* = -J^* {J^*}^T g^*.$$

The sequential algorithm is thus given by:

Method III (Sequential)
Calculate u and v

```
1  $\alpha = s^T y$
```

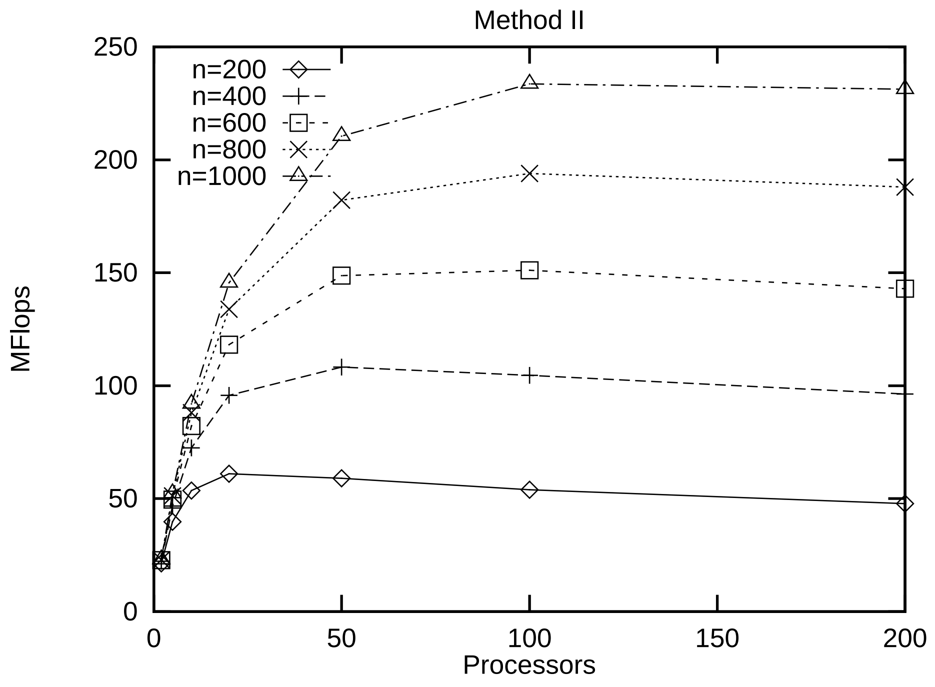


Figure 2: Performance of Method II

2 $\beta = s^T g$

3 $u = s/\alpha$

4 $\gamma = \sqrt{\frac{-\lambda\alpha}{\beta}}$

5 $v = -\gamma g - y$

Update J

6 $t^T = v^T J$

7 $J^* = J + ut^T$

Calculate next search direction

8 $t = {J^*}^T g^*$

9 $p^* = -J^* t$

The total cost for the sequential algorithm is $(8n^2 + 5n + 2)T_f + T_{sq}$.

The cost of a parallel implementation of this algorithm again depends on the storage scheme selected for the matrix J. The costly steps in this algorithm are steps 6, 7, 8 and 9. For steps 6 and 8 column storage is preferable allowing the same method to be used as is used in step 1 of Method I. Step 9, however, would be cheaper to perform if the matrix were stored by rows. The cost of the rank one update in step 7 is not affected by the selection of row or column storage. For the least cost, we choose to distribute columns of J to the processes, either cyclically or in block columns. The storage for the vectors is as follows: s, y, u, v, t and g^* are distributed to the processes whilst p^* is stored in full on each process. Eight temporary vectors of size n/p are required on each process for the summation of vectors in step 9. We assume that at the start of an iteration of the algorithm g holds the value of g^* from the previous iteration.

The first phase of the algorithm, starts with the distribution of s and g^* costing $(p-1)(T_s + (n/p)T_c)/2$. Then y can be calculated at cost $(n/p)T_f$ from $y = g^* - g$. The two inner products follow each costing

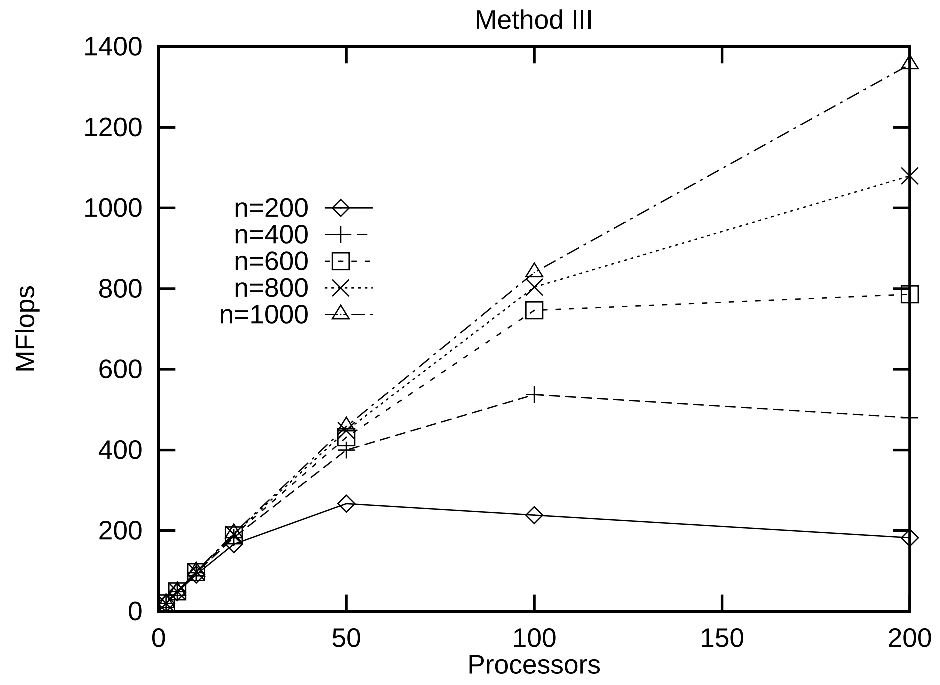


Figure 3: Performance of Method III

$(2n/p - 1)T_f + (4T_f + T_s + T_c)\log_5 p$. Next, $-\gamma$ is calculated and broadcast along with α costing $4T_f + T_{sq} + (T_s + 2T_c)\log_5 p$. This allows the calculations of u and v to be performed at cost $(3n/p)T_f$.

Steps 6 and 8 both use the algorithm in step 1 of Method I. An additional cost of $(n/p)T_f$ is incurred in negating t at the end of step 8. The rank one update of step 7 is similar to the rank two update in step 6 of Method I and costs

$$\max\left(T_s + \frac{n}{p}T_c, \frac{2n^2}{p^2}T_f\right)(p-1) + \frac{2n^2}{p^2}T_f.$$

Step 9 is similar to step 2 of Method II except that in this case all elements of the full matrix are non-zero. This increases the cost of the initial partial summation on each process to $(2n^2/p)T_f$. Finally, p^* is gathered to the master process at cost $(p-1)(T_s + (n/p)T_c)/4$.

Figure 3 shows the predicted performance of this method using the same values for problem parameters as for Methods I and II.

5 Comparison of methods

The graphs in Figures 1 to 3 show the MFlop rate achieved by the algorithms for varying numbers of processors and problem sizes. This allows us to see how efficient each parallel algorithm is and to see how well the algorithms scale with the number of processors and problem size. For large problem sizes ($n = 1000$) Methods I and III have very good efficiencies around 60–70% for the numbers of processors shown. For smaller networks Method III has a slightly greater efficiency than Method I, whilst Method I performs better for the larger networks. This indicates that both of these algorithms would scale well for big problems on big machines with Method I being preferred on larger networks. The graph for Method II reveals a very poor performance. Even for large problems the algorithm achieves much lower efficiencies than the other two algorithms. Particularly important is the fact that the algorithm does not scale well as the number of processors is increased. The MFlop rate peaks at only 100 processors for the biggest problem size shown. Also for small networks the performance is not as good as for the other algorithms. Hence Methods I or III are to be preferred for all problem and network sizes.

There seem be two reasons for this contrast in performance between the algorithms. Firstly, the greater proportion of the sequential costs in Methods I and III is in level 2 BLAS operations. The computations required by these BLAS parallelize very well and lead to efficient algorithms with few synchronizations for communication. Method II, however, contains costly Jacobi rotations and backward solves which are difficult to parallelize and require frequent communications between processes. The second factor giving Methods I and III much better performances than Method II is the way in which most of the more expensive communications in Methods I and III have been hidden behind arithmetic operations and therefore do not contribute much to the total cost of the algorithms. This is not achieved by Method II since it requires frequent small communications with only few arithmetic operations being performed in between each communication.

Figure 4 shows a comparison of the methods for two problem sizes. The algorithms are compared by plotting the speedup of each parallel algorithm when compared with the best sequential method, which is Method I. This gives a comparison of the 'elapsed times' of the different algorithms. For large problems the dominant factor in determining the best parallel method is the cost of the method's sequential algorithm. This factor has more effect than the suitability of the sequential algorithm for parallelization, or the details of the particular implementation of the parallel algorithm, such as whether full or triangular storage is used. Hence the top graph shows a clear distinction between the three methods with performance increasing from Method II (sequential cost $\approx 10n^2$) to Method III ($8n^2$) and then Method I ($4n^2$). There is relatively little difference in the performance of the two implementations of Method I compared with the other methods emphasising the importance of basing the parallel algorithm on the best sequential algorithm unless this is clearly unsuitable for parallelization. The rapid fall in speedup when $n = 200$ and more than 50 processors are used indicates that the granularity of all these algorithms is quite coarse and each process should hold several (perhaps at least 4) rows of the matrix for good efficiency.

Another important factor to consider when comparing these algorithms is the amount of information which each algorithm provides. All of the algorithms give an updated Hessian or inverse Hessian matrix and the next search direction. In addition Method II, since it stores the factors LL^T of the Hessian, can easily provide information about the positive-definiteness of the matrix. This is essential for practical problems to ensure that the algorithm is robust. A further factor in favour of Method II is its lower memory requirement; Methods I and III require distributed storage for about n^2 words whilst Method II only needs about $n^2/2$. The main disadvantage of Method II is its large sequential cost. A more efficient sequential algorithm which stores the Choleski factors is given in [5]. This algorithm still has a higher cost than Method I and consists of many level 1 operations involving a lot of synchronisations but may well be worth future investigation due to the advantages of the method outlined above.

The results in this paper suggest that Method I is the best parallel algorithm to use for the linear algebra sections when solving non-linear unconstrained optimization problems using a quasi-Newton method. The algorithm gives good performance and scales very well as the number of processors increases especially for larger problems.

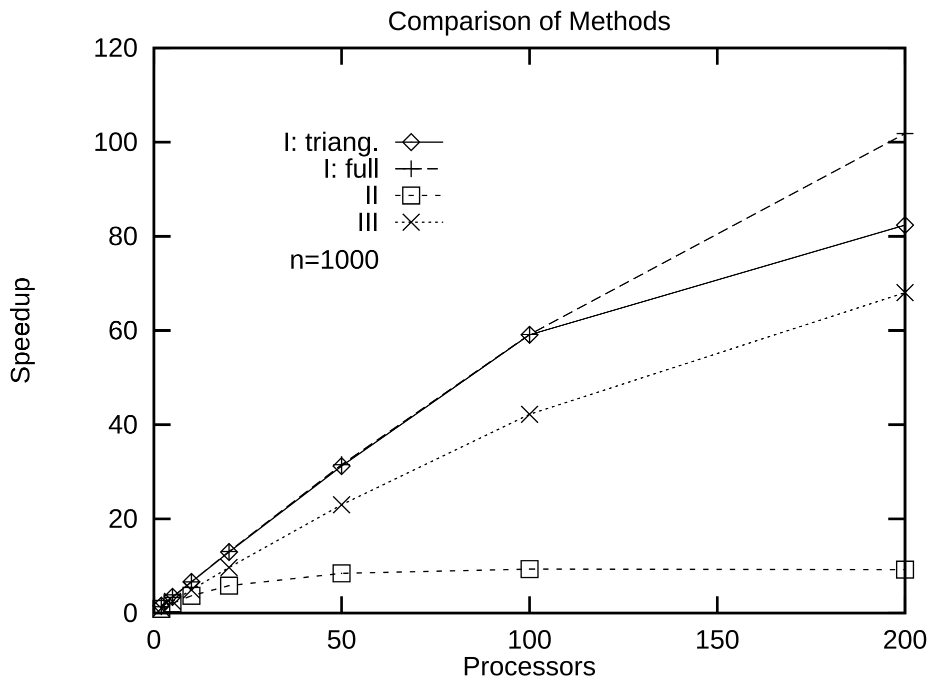
It is important to view these results in the context of a complete optimization algorithm. As mentioned in Section 1, the calculation of the step length along the search direction is also costly. This operation normally involves function and gradient evaluations and when the objective function is complicated these calculations can dominate over the linear algebra costs. Hence, as well as using parallel linear algebra in a quasi-Newton algorithm one must use parallel function evaluations in the line search of Step 2.

References

[1] Richard H. Byrd, Robert B. Schnabel, and Gerald A. Shultz. Parallel quasi-newton methods for unconstrained optimization. Technical report, Department of Computer Science, University of Colorado, April 1988.

[2] W. C. Davidon. Optimally conditioned optimization algorithms without line searches. *Mathematical Programming*, 9:1–30, 1975.

[3] J.E. Dennis, Jr and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall, 1983.



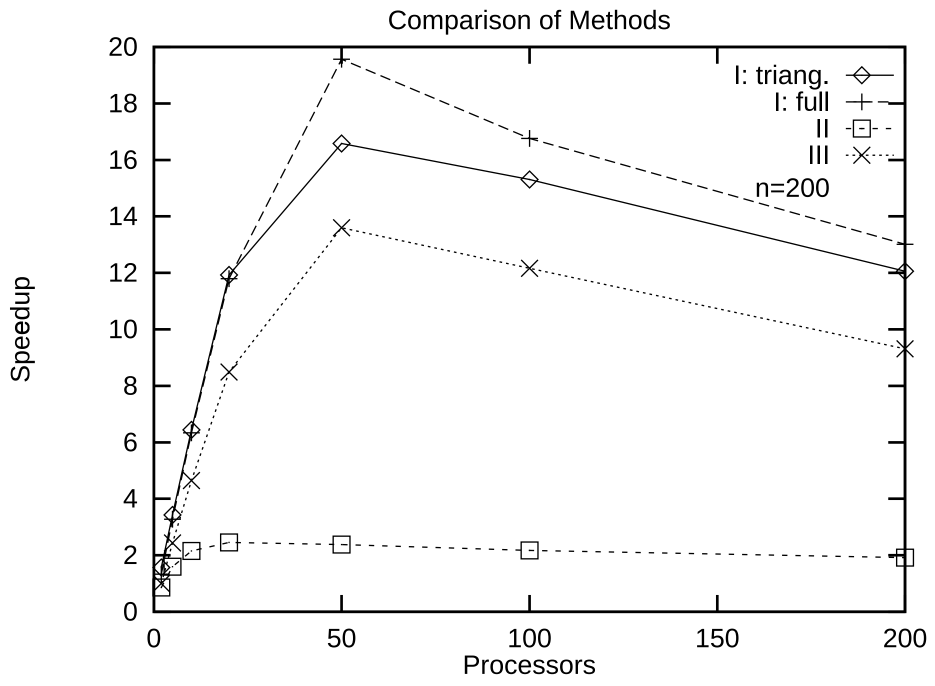


Figure 4: Comparison of methods

[4] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Academic Press, 1981.

[5] Donald Goldfarb. Factorized variable metric methods for unconstrained optimization. *Mathematics of Computation*, 30(136):796–811, October 1976.

[6] Holm Hofestädt, Axel Klein, and Erwin Reyzl. Investigation of dynamically switched, scalable network structures. PUMA Deliverable 2.1.1, Siemens AG, October 1990.

[7] INMOS. H1 transputer: Advance information. Technical report, June 1990.

[8] K.W.Brodlie, A.R.Goulay, and J.Greenstadt. Rank-one and rank-two corrections to positive definite matrices expressed in product form. *Journal of the Institute of Mathematics and its Applications*, 11:73–82, 1973.

[9] Tim Oliver. A communications model for a PUMA machine. PUMA Working paper 17, University of Liverpool, October 1990.

[10] P.E.Gill, G.H.Golub, W.Murray, and M.A.Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, April 1974.

[11] M. J. D. Powell. Updating conjugate directions by the BFGS formula. *Mathematical Programming*, 38:29–46, 1987.

[12] R.Fletcher. *Practical Methods of Optimization*. John Wiley & Sons, 2nd edition, 1987.