MULTI-DIRECTIONAL SEARCH METHOD WITH QUADRATIC MODEL

by

Haibo Ma

B.Sc., National University of Defense Technology, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

[redacted] in the Department of Computer Science

ACCEPTED [redacted] STUDIES [redacted] DEAN

We accept this thesis as conforming
to the required standard

[redacted]

Dr. A. G. Buckley, Supervisor (Department of Computer Science)

[redacted]

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

[redacted]

Dr. I. Sharf, Outside Member (Department of Mechanical Engineering)

[redacted]

Dr. D. J. Shpak, External Examiner (Department of Electrical and Computer Engineering)

Supervisor: Dr. A. G. Buckley

Abstract

In recent years, there has been a great deal of interest in the development of optimization algorithms which exploit the computational power of parallel computer architectures. V. Torczon has developed a new algorithm, called the Multi-Directional Search method, which is ideally suited for parallel computation.

The multi-directional search algorithm belongs to the class of direct search methods, a class of optimization algorithms which neither compute nor approximate any derivatives of the objective function. This method is inherently parallel because its basic idea is to perform concurrent searches in multiple directions which are free of any interdependencies. Another significant feature of Torczon's algorithm is that a convergence theory was developed when the algorithm was proposed.

In this thesis, we incorporated the idea of approximating a general function by a quadratic function into Torczon's multi-directional search algorithm, proposing an algorithm called the Multi-Directional Search algorithm with Restart. Using our algorithm, when the objective function is a quadratic, the algorithm locates the minimizer in only two iterations. Even for a general objective function, by using the information obtained from the quadratic approximations, the overall searching process can be accelerated. In this sense our algorithm exhibits better performance than Torczon's original method.

Another issue this thesis addressed is the implementation of the algorithm. We use a tree structure to replace the matrix Torczon used in her implementation. With the new

data structure, we can considerably reduce the amount of storage used and the number of comparisons performed in the whole process.

Also presented here are the numerical results showing the comparisons of the performance of our algorithm with that of Torczon's original method, as well as the comparison of the performance of using a tree structure against Torczon's approach of using a matrix.

Examiners:

[redacted]

Dr. A. G. Buckley, Supervisor (Department of Computer Science)

[redacted]

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

[redacted]

Dr. I. Sharf, Outside Member (Department of Mechanical Engineering)

[redacted]

Dr. D. J. Shpak, External Examiner (Department of Electrical and Computer Engineering)

Acknowledgements

I would like to thank my supervisor, Professor A. G. Buckley, for his valuable guidance of my research work and for providing many detailed and perceptive comments on my thesis.

My thanks also go to Professor C. Paige from McGill University for his idea of dealing with the singularity problem of the coefficient matrix determined by the chosen points.

Also I would like to extend my thanks to the Center for Research on Parallel Computation, Rice University, for providing the equipment support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Almost all numerical methods for determining the optimum of a given non-linear objective function $f(x)$ of n variables[1] $x = (x_1, x_2, \ldots, x_n)^T$ are iterative and start from a given initial estimate for the solution. They proceed by generating a sequence of estimates, each of which represents an improvement over the previous ones. The different procedures are characterized by their strategies used to produce a sequence of improving approximations. This thesis deals with the class of direct search methods, whose strategy is based on the comparison of values of the objective function only. That is to say, these methods make no use of any of the derivatives of the function.

[1] Here, x_i denotes the ith entry of x.

1.1 Background

Simply by looking at the name "Direct Search Methods", it is not hard to understand that these algorithms share the common feature that they are all derivative-free. Several of the direct search methods—in particular the Nelder-Mead simplex algorithm [NM64] and the pattern search algorithm of Hooke and Jeeves [HJ61]—have long enjoyed popularity in the community of computational scientists. These algorithms have remained popular for practical reasons. Often, particularly in experimental settings (for example, if we deal with problems where the function is the result of a computer simulation), the derivatives are not readily available. In addition, function values based on experimental data are often "noisy" (i.e., they can only be trusted to a few digits of accuracy) so that finite-difference derivative approximations may prove unreliable. Finally, most of the direct search methods are easy to understand, and relatively easy to program.

On the other hand, direct search methods have fallen out of favor with the numerical optimization community because, for the most part, they lack any convergence theory; they are exceedingly slow to converge, even in the neighbourhood of a solution; and they either do not converge to a true solution, or else they converge very slowly, when the problem is "not small"—where by not small, we mean there are ten or more variables. Despite these reservations, the active literature in such fields as analytical chemistry suggests that practical considerations often win out over perceived disadvantages.

Recently, Torczon developed a new direct search method [DT91, Tor89], which is called the multi-directional search (MDS) algorithm, with the following features:

- It conducts a multi-directional search that can be executed in parallel;
- It is robust for problems of moderate size;
- It works well with noisy function values;
- It is easy to understand and easy to program.

The key feature of this parallel algorithm is that it introduces a multi-directional search into the decision-making framework of the sequential Nelder–Mead simplex algorithm. This multi-directional search algorithm is a new direct search method that can be easily implemented on certain types of parallel machines, but it is not just a parallel implementation of an existing sequential direct search algorithm. In addition, a convergence theory for this algorithm has been developed [Tor90, Tor89].

Parallel processing has been a popular approach for improving system performance through several generations of computer systems design. It is an important theme of computer system development work at both the "system organization" and "system software" levels. It seeks to achieve each of the following goals, as has each new major systems concept or development (such as multiprogramming, multiprocessing and networking):

- increased system productivity, greater capacity, shorter response time and increased throughput;
- improved reliability and availability;
- ease of expansion;
- improved ability to share system resources.

Torczon's MDS method proposed a way of combining classic optimization methods, in particular the direct search methods, and some of the most recent development in computer architecture. Her work aroused reconsideration of direct search methods, which had been falling out of favor in the past two decades.

1.2 Our work

A general quadratic function has the form:

$$q(x) = \frac{1}{2}x^T G x + b^T x + c \tag{1.1}$$

where G, b, and c are constant and G is symmetric. By the rule for differentiating a product, it can be verified that

$$\nabla(u^T v) = (\nabla u^T)v + (\nabla v^T)u. \tag{1.2}$$

Therefore, it follows from Eq. 1.1 that

$$\nabla q(x) = \frac{1}{2}(G + G^T)x + b = Gx + b. \tag{1.3}$$

Here, the symmetry of G is used. Similarly, $\nabla^2 q = G$ can be derived.

Many methods for unconstrained minimization have the feature that they will work well, or even exactly, if applied to a quadratic function [Fle90]. However, it is more meaningful that these methods can be applied to minimize general functions iteratively. Such methods are said to be derived from a *quadratic model.* (As a matter of fact, in cases when the function can be analytically expressed, the minimization of a quadratic function can be achieved simply by solving a linear system of equations, which is not the situation

here.) The quadratic model approach has proven to be very fruitful in practice and it is relevant to the efficiency and rapid local convergence of many such methods. Some of the probable reasons are the following:

- A quadratic function is the simplest smooth function with a well determined minimum (if G is positive definite), and it is easy to manipulate;
- A general smooth function expanded about a local minimizer x^* is approximated well by a quadratic function. Thus methods based on quadratic models should have a rapid ultimate rate of convergence;
- Even remote from the minimum, it seems preferable to use information based on a quadratic approximation rather than to reject it. This quadratic information is more effective than linear information when used to predict directions along which substantial progress can be made, presumably because a Taylor series of $f(x)$ about an arbitrary point $x^{(k)}$, if taken to quadratic terms, will agree with $f(x)$ to a given accuracy over a much greater neighbourhood of $x^{(k)}$ than will the series taken to linear terms;
- Methods based on quadratic models can often be made invariant under a linear transformation of variables.

However, in Torczon's MDS method, the quadratic model is not utilized at all. Our work tries to improve the MDS algorithm by taking advantage of the idea of the quadratic model and making the MDS method converge more rapidly. In particular, when the objective function is itself a quadratic one, the algorithm will converge after only one or

two iterations. This is quite significant as it represents a substantial reduction in the number of function evaluations needed when compared with Torczon's original method.

In addition to consideration of the quadratic model, studying the implementation details of Torczon's work led us to consider the data structures she used. In her work, a large matrix is used for storing information about the points considered, which leads to using a large number of comparisons in order to decide if a newly generated point is a duplicate of previous ones. In our work, a tree structure is used to replace the matrix. Hence, not only the amount of storage consumed is reduced, especially for large problems with dimension more than ten[2], but also the number of comparisons used is reduced dramatically. In Section 5.2, we will further explain why storage is a particular concern in the algorithm we will propose.

[2]For non-derivative methods, a problem with dimension more than ten is deemed to be a large problem.

Chapter 2

Direct Search Methods

As stated in the previous chapter, the most noticeable characteristic of direct search methods is that they do not use derivatives. There is certainly little doubt that derivatives (if they are available) can be used to speed up the average performance of nonlinear optimization algorithms, but in practice, it is often only derivative approximations that are available instead of derivatives. And sometimes it turns out that even derivative approximations are impractical. In this sense, derivative-free schemes are more widely applicable than gradient methods, including quasi-Newton methods. Besides, derivative-free methods can also be quite robust in dealing with objective functions that can be evaluated to only a few significant digits, in which case finite-difference approximations are too inaccurate to be acceptable.

The problem to be considered by direct search methods is that of minimizing a function $f(x)$, where $x = (x_1, x_2, \ldots, x_n)^T$. The general form of direct search is as follows. Suppose a point x is the current point. A second point x_+ is selected by some method or strategy

and is compared with x. If x_+ is *better* than x, it becomes the current point. For minimization, *better* means that $f(x_+) < f(x)$. If x_+ is not better than x, x is retained as the current point and the process takes a different strategy to go on generating a new point. In this way, a monotonic sequence of points is generated. It is clear that the current point is simply the "best" point found at the current stage of the search process. This process continues until some predetermined stopping criterion is satisfied.

From the above general form, we can see that the strategy for selecting new candidate points is of paramount importance. Different strategies lead to different methods, such as the Pattern Search method [NM64], the Razor Search method [Swa72] and the Simplex method[1] [Swa72]. As the method used in the thesis is primarily based on the *Nelder–Mead simplex method*, which is a modification of the original simplex method, it is necessary to introduce both methods in a bit more detail before going further.

2.1 The Simplex Method

One of the earliest methods which merely compare function values is the one known as the *simplex* method proposed by Spendley, Hext and Himsworth in 1962 [SHH62].

A (regular) simplex is defined to be a set of $n+1$ (equidistant) points in $\mathcal{R}^n$, such as the equilateral triangle for $n = 2$ or tetrahedron for $n = 3$. The information used in the method includes the coordinates of the $n+1$ points and their corresponding function values.

[1]This should not be confused with the simplex method of linear programming, although the name is derived from the same geometrical concept.

The method begins by setting up a regular simplex in $\mathcal{R}^n$ and evaluating the function at each vertex of the simplex. If x_0 is a given initial estimate for the minimum, then a simplex of unit edge can be constructed by choosing the vertices[2] $x_0, x_1, \ldots, x_n$, where

$$\begin{aligned} x_0 &= (x_{0_1}, x_{0_2}, \ldots, x_{0_n})^T, \\ x_i &= (x_{0_1} + \delta_1, x_{0_2} + \delta_1, \ldots, x_{0_{i-1}} + \delta_1, x_{0_i} + \delta_2, \\ &\qquad x_{0_{i+1}} + \delta_1, \ldots, x_{0_n} + \delta_1)^T, \qquad i = 1, \ldots, n, \end{aligned}$$

with

$$\begin{aligned} \delta_1 &= \frac{(n+1)^{\frac{1}{2}} - 1}{n\sqrt{2}}, \\ \delta_2 &= \frac{(n+1)^{\frac{1}{2}} + n - 1}{n\sqrt{2}}. \end{aligned}$$

Equivalently,

$$x_i = x_0 + \delta_1(1, 1, \ldots, 1)^T + \frac{1}{\sqrt{2}} e_i, \qquad i = 1, \ldots, n.$$

Each iteration of the search consists of replacing one of the vertices, x_j say, by its mirror image $\bar{x}_j$ in the centroid of the remaining n vertices, i.e.,

$$\bar{x}_j = \frac{2}{n}\Big(\sum_{\substack{i=1 \\ i \neq j}}^{n} x_i\Big) - x_j.$$

It then computes $f(\bar{x}_j)$. Note that each step needs only one function evaluation. The point to be replaced, x_j, is normally chosen to be the vertex with the highest function value (which is considered to be the *worst vertex* of the current simplex). However, on iterations after the first it might be that the newest vertex is now the worst vertex of the

[2]Here, x_i denotes the ith vertex and x_{i_j} denotes the jth entry of x_i.

new simplex, and that would cause oscillation between the last two simplices. To avoid this, whenever the worst vertex is also the one most recently introduced, the next to worst one is chosen to be reflected instead.

Ultimately, the iteration will fail to make further progress, so an additional rule has to be incorporated. When one vertex x' has been in the current simplex for more than a fixed number M of iterations, the simplex should be contracted. The contraction is done by replacing the other vertices by new ones half way along the edge to the vertex x'. From experimental data, Spendley *et al* suggested that $M = 1.65n + 0.05n^2$ is a reasonable choice.

The *simplex method* provides a useful procedure for minimizing a given function. However, it assumes that the number of simplices generated before the contraction step is taken is known. This makes this method rather rigid for general use, because the method does not take into account the local geometry of the objective function, nor does it consider the possibility that the initial simplex needs to be scaled before reflection steps start.

A modified simplex method is due to Nelder and Mead [NM64], and will be introduced in the next section.

2.2 The Nelder–Mead Simplex Method

In 1965, Nelder and Mead revised the simplex method proposed by Spendley *et al*. In this version, the regularity of the design is abandoned and the simplex automatically rescales itself according to the local geometry of the objective function.

Suppose that the vertices of the simplex are $x_0, x_1, \ldots, x_n$ with corresponding function

values $f_0, f_1, \ldots, f_n$ ordered such that

$$f_n \geq f_{n-1} \geq \cdots \geq f_1 \geq f_0.$$

Hence x_0 is the best vertex of the simplex and x_n is the worst. Then the worst vertex x_n is to be replaced.

Let c denote the centroid of the vertices $x_0, x_1, \ldots, x_{n-1}$, so

$$c = \frac{1}{n} \sum_{j=0}^{n-1} x_j. \tag{2.1}$$

A simple reflection move is tried first, generating a new point x_r, where

$$x_r = c + \alpha(c - x_n).$$

Here $\alpha > 0$ is called the *reflection coefficient* (to be discussed later along with the expansion coefficient and contraction coefficient), and c can be computed using Eq. 2.1.

There are then three possible cases to be considered: x_r is a point such that

$$f_0 \leq f_r \leq f_{n-1};$$

$$f_r < f_0;$$

or $$f_r > f_{n-1}.$$

In the case where $f_0 \leq f_r \leq f_{n-1}$, the best point x_0 would not be affected and x_r is surely better than x_n. So x_r simply replaces x_n and then the iteration is complete.

In the case where $f_r < f_0$, x_r would be a new best point. Furthermore, the direction of the reflection may be a useful one. So an attempt is made to expand the simplex along that direction by defining the point

$$x_e = c + \beta(x_r - c),$$

where $\beta > 1$ is called the *expansion coefficient*. Now, if $f_e < f_0$, the expansion is considered to be successful and x_n is replaced by x_e; otherwise the expansion is deemed to have failed and x_n is replaced by x_r instead. In either event, the iteration is then complete.

Finally, if it is the case that $f_r > f_{n-1}$, x_r would be a new worst point. Then it is assumed that the size of the simplex is too large to allow any progress to be made, so a contraction step is performed, giving the point x_c, where

$$x_c = c + \gamma(x_n - c), \qquad \text{if } f_n \leq f_r,$$

or

$$x_c = c + \gamma(x_r - c), \qquad \text{if } f_n > f_r.$$

Now γ is called the *contraction coefficient*, with $0 < \gamma < 1$. If $f_c < \min(f_n, f_r)$, the contraction has succeeded and x_c replaces x_n; otherwise a more comprehensive contraction is to be carried out, which is conducted by halving the distances from the best point x_0 to all the other vertices of the simplex. In either case, the iteration is then complete.

The reflection, expansion and contraction steps for a function of two variables are illustrated in Fig. 2.1.

In fact, different values of α, β and γ provide variations of this simplex method. Experimental results by Nelder and Mead show that $\alpha = 1$, $\beta = 2$, and $\gamma = \frac{1}{2}$ is a suitable choice, at least for the test functions they selected.

Before concluding this section, the convergence criterion of this method needs to be mentioned. Here, the convergence criterion is based on the variation in the function values over the simplex, which means that the search stops when the standard deviation σ of

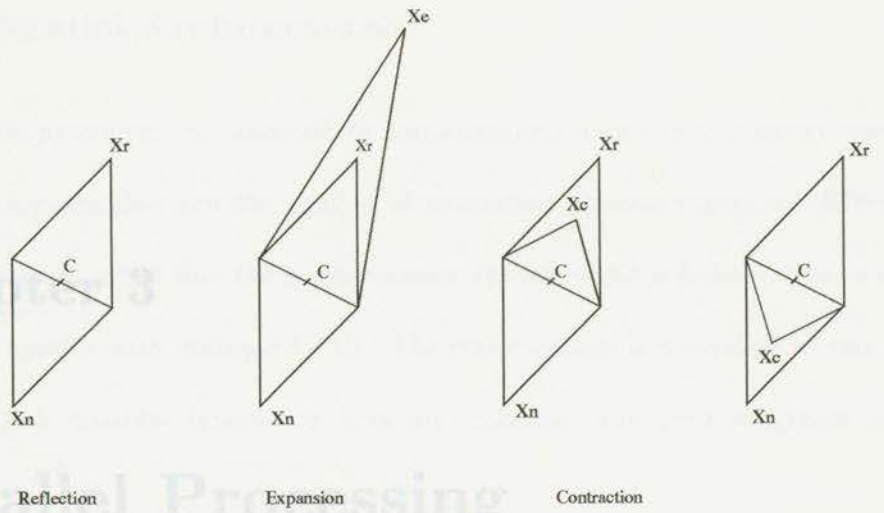

Figure 2.1: Nelder–Mead reflection, expansion and contraction steps

the function values at the vertices falls below some pre-assigned limit, where

$$\sigma = \sqrt{\sum_{i=0}^{n} \frac{(f_i - \bar{f})^2}{n}} \tag{2.2}$$

and $\bar{f}$ is the mean of the function values.

2.3 Summary

So far, the fundamental ideas of the two methods upon which the thesis work is based have been introduced. It is clear that they are both sequential algorithms which cannot be easily adapted to parallel machines. Before presenting a parallel method, which was proposed by Torczon and based on the Nelder–Mead simplex method, we include a discussion of parallel processing.

Chapter 3

Parallel Processing

It is well known that there are many non-linear optimization algorithms which have been proven to be successful on sequential machines. However, these algorithms are not always satisfactory when they are used to solve "real world" problems, since these problems may take an unacceptably long time on a sequential machine before they are solved. For example, Dennis and Schnabel quoted a 50-variable problem in petroleum engineering where each function evaluation costs 100 hours of IBM 3033 time [DS83]. In a real time environment, the time requirement may be especially critical. Therefore the problems may only be soluble in principle but not in fact. By introducing the parallel processing concept into numerical optimization, it is hoped that the range of practically soluble problems can be extended. Our work was done in such a direction. Therefore, we first include an introduction to parallel processing.

3.1 Parallel Architectures

Parallel computers can be classified by considerations about their memory organization, processor organization and the number of instruction streams supported [HB84, Hay78]. Here, we are concerned with the multiprocessor system, which is defined to be an integrated computer system with multiple CPUs. The entire system is controlled by one operating system which provides interaction between processors and their programs at different levels.

3.1.1 Memory Organization

In regard to memory organization, there are two different sets of architectural models. One is called a *tightly-coupled* multiprocessor system and the other a *loosely-coupled* multiprocessor system.

Tightly-coupled multiprocessors communicate through a shared main memory. Therefore the rate at which data can be communicated from one processor to another depends largely on the memory access time. Each processor may have a small local memory or high-speed buffer (cache). A complete connectivity between the processors and the memory can be accomplished either by inserting an interconnection network between the processors and the memory or by a multi-ported memory. However, when more than one processor attempts to access the same memory unit at the same time, memory conflicts occur, which may result in the performance degradation of a tightly coupled multiprocessor system.

In loosely-coupled multiprocessor systems, each processor has a large local memory where it accesses its instructions and most of its data. Though there is no direct sharing

of memory, the local memories together form the shared *address space* of the computer. Processes which execute on different processors communicate by exchanging messages through a *message-transfer system.* Typically a memory reference to the local memory of a processor is orders of magnitude faster than a memory reference to a remote memory. This is so because the remote memory reference has to be routed through the interconnection network. In this sense, loosely-coupled systems are usually efficient only when the interactions between processes are minimal.

Since the system we have used belongs to the category of *multicomputer system*, we are more concerned with the architecture of this group, which is somewhat similar to that of a loosely-coupled multiprocessor system. The significant difference is that a multicomputer system has neither a shared memory nor a shared memory address space. Consequently, if a processor needs to use data in a remote memory, it has to explicitly move that data into the local memory. This and all other kinds of interprocessor communications are done by passing messages (via an interconnection network) among the processors. Hence, we have to make sure that messages are properly assembled before being sent out so that they contain all the necessary data. Also, each relevant processor must know how to extract the useful data from the messages it receives.

3.1.2 Processor Organization

Processor organization is defined by the interconnection network used to connect the processors of a multicomputer system. Some of the more common interconnection networks are the two dimensional mesh, ring, tree and hypercube, as shown in Fig. 3.1.

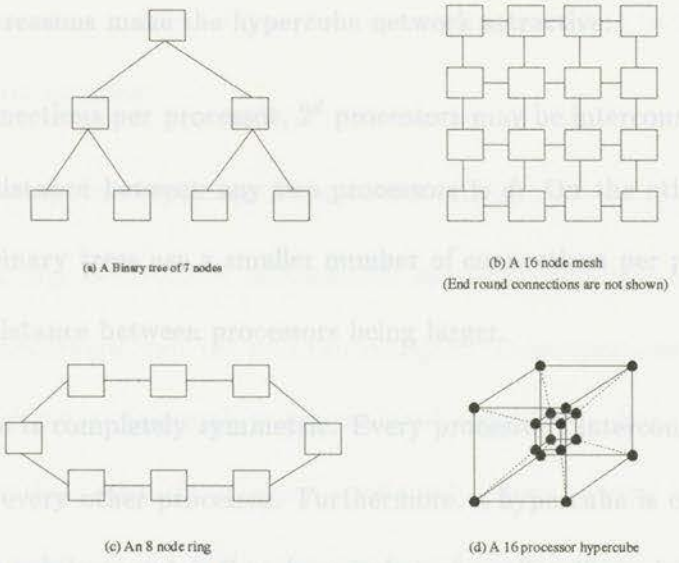

Figure 3.1: Types of interconnection networks

Here, only the hypercube [RS90] is considered, as the processors in the system we used are interconnected this way. But in fact, the basic multi-directional search method can be implemented not only on a hypercube system, but also on other systems, since the interconnection network does not affect the nature of the multi-directional search method. In a hypercube of dimension d, there are 2^d processors, which can be labeled $0, 1, \ldots, 2^d - 1$. Two processors, say i and j, are directly connected if and only if the binary representations of i and j differ in only one bit. Each edge of Fig. 3.1(d) represents a direct connection. Thus in a hypercube of dimension d, each processor is directly connected to d others. If the direct connection between a pair of processors i and j is unidirectional, then at any given time, messages can flow either from i to j or from j to i. In the case of bidirectional connections, it is possible for i to send a message to j and for j to send a message to i simultaneously.

The following reasons make the hypercube network attractive:

- Using d connections per processor, 2^d processors may be interconnected so that the maximum distance between any two processors is d. On the other hand, meshes, rings, and binary trees use a smaller number of connections per processor with the maximum distance between processors being larger.

- A hypercube is completely symmetric. Every processor's interconnection pattern is like that of every other processor. Furthermore, a hypercube is completely decomposable into sub-hypercubes (i.e., hypercubes of smaller dimension).

3.1.3 Instruction Streams

As is well known, a typical processor (CPU) operates by fetching instructions and operands from main memory, executing the instructions, and placing the results in memory. So, the instructions can be viewed as forming an *instruction stream* flowing from main memory to the processor, while the operands form another stream, the *data stream*, flowing bidirectionally between the processor and the memory. According to M. J. Flynn [Fly66], parallel processors can be classified based on the number of simultaneous instruction and data streams the processor sees during program execution. Suppose that a processor P is operating at its maximum capacity (i.e., its parallelism is fully exhibited). Let $m_I(m_D)$ denote the number of distinct instruction (data) streams. Then m_I and m_D can be termed as the instruction and data stream *multiplicities* of P and can be used to define a measure of P's degree of parallelism, which is intended to indicate the processor's ability to perform tasks in parallel.

In general, computers mainly fall into four categories, according to the multiplicity of instruction and data streams:

- Single Instruction Stream Single Data Stream (**SISD**):

  Here, $m_I = m_D = 1$. Most conventional machines with one CPU containing a single arithmetic-logic unit fall into this category. These machines are only capable of scalar arithmetic. Instructions are executed sequentially, but they can be made to overlap in their execution stages (pipelining). Many current SISD uniprocessor systems are pipelined. An SISD computer may have more than one functional unit, with all the functional units under the supervision of one control unit.

- Single Instruction Stream Multiple Data Stream (**SIMD**):

  Here, $m_I = 1$, $m_D > 1$. This category includes machines with a single control unit and multiple processing elements (PE). It also includes associative or content-addressable memory processors (in which any stored item can be accessed directly by using the contents of the item in question, as an address). All PEs receive the same instruction broadcast from the control unit but operate on different data sets from distinct data streams.

- Multiple Instruction Stream Single Data Stream (**MISD**):

  Here, $m_I > 1$, $m_D = 1$. The machines in this category have multiple processor units. Each processor unit receives distinct instructions operating over the same data stream. Less attention has been put on this structure in that it has been deemed as impractical by some computer architects. So far, no actual system with

this structure really exists.

- Multiple Instruction Stream Multiple Data Stream (**MIMD**):

  Here, $m_I > 1$, $m_D > 1$. This covers machines capable of executing several independent programs simultaneously. It turns out that most multiprocessor systems and multiple computer systems can be classified into this category. In MIMD multicomputer systems, there is no separate control unit and the local memory of each PE holds both the data and the program that the PE is to execute.

Now the question is, which types of machines are suitable for the MDS algorithm? It is obvious that SISD is not our concern here, although the multi-directional search method can be implemented on a sequential machine.

We focus on the MIMD machines so that we can separate the function evaluation from the control procedure of the MDS method, making up two instruction sets—one is the control program which is the implementation of the algorithm we proposed, the other is the function evaluation program with several duplicates running on several processors at the same time. We can better understand this after the next section.

For SIMD machines, with the appropriate restrictions on the function evaluation routine, the MDS method can also be customized, taking advantage of the fact that all the processors can equally communicate with each other through message passing mechanism. In this case, each processor is treated equally, running the same program *or rather same set of instructions* but dealing with different sets of data.

Unlike in a SIMD parallel computer where all processors execute in a synchronous

manner, MIMD parallel computers are generally asynchronous and different processors may execute different instructions at any given time.

A block diagram of an MIMD machine is shown in Fig. 3.2.

3.2 Master–Slave Processing

The *master-slave configuration* is one of the basic organizations that have been utilized in the design of algorithms for multiprocessors. In this mode, one processor, called the *master*, maintains the status of all processors in the system and assigns the work to all the *slave* processors. In a multiprocessor system, processes can execute concurrently until they need to interact. Messages are used to define the interface between processes. Planned and controlled interaction is referred to as process communication or process synchronization.

The idea of *master-slave* processing can be adopted when programming in a multiprocessor/multicomputer system. The program running on the master processor can be called the *master* program, and the program running on the slave processor can be called the *slave* program. It is generally the case that the slave programs only communicate with the master program, though they can also communicate with each other without the interference of the master program (although in this case, a more complicated synchronization control is required).

In our work, we regard the program containing the optimization search strategy to be the master program and the function evaluation program to be the slave program. The synchronization is solely handled by the master program through message passing. With this configuration, when the objective function evaluation is quite time consuming, the

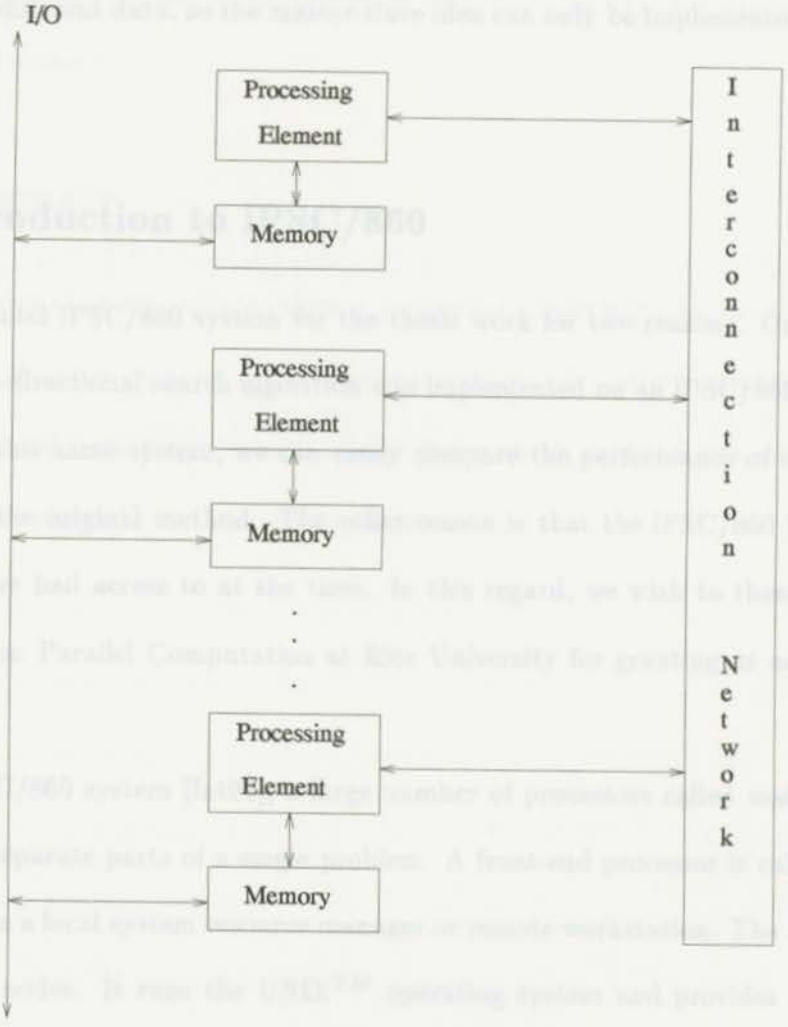

Figure 3.2: Block diagram of an MIMD multicomputer

master program will have to waste a lot of time being idle while waiting for the messages from the slave programs. This observation is the major motivation of our thesis work.

As we can see, the master program and slave program form different instruction sets dealing with different data, so the master-slave idea can only be implemented on a MIMD machine.

3.3 Introduction to iPSC/860

We used an Intel iPSC/860 system for the thesis work for two reasons. One is that the original multi-directional search algorithm was implemented on an iPSC/860, so by doing our work on this same system, we can easily compare the performance of our algorithm with that of the original method. The other reason is that the iPSC/860 is the MIMD system that we had access to at the time. In this regard, we wish to thank the Center for Research on Parallel Computation at Rice University for granting us access to their hypercube.

In an iPSC/860 system [Int91], a large number of processors called *nodes* work concurrently on separate parts of a single problem. A front-end processor is called the *host*, which could be a local system resource manager or remote workstation. The *host* provides access to the nodes. It runs the UNIX^{TM} operating system and provides the interface between the programmer and the nodes. The compilation and linkage of the host and node programs are both done on the host, and then the executable codes of the host program are loaded and run on the host while the executable codes of the node programs are loaded and run on the nodes.

A *node* is in fact a processor/memory pair. Each node's memory is distinct from that

of the host and other nodes. All the nodes can communicate with the others (by passing messages), and can access the host file system.

A collection of nodes belonging to an iPSC/860 system is called a *cube*. The number of nodes in a cube is defined by its dimension (expressed as dn). For example, a $d7$ cube has 2^7 or 128 nodes.

3.4 Summary

In the first three chapters, all the relevant preliminary knowledge for the thesis has been presented. In the next chapter, the multi-directional search method proposed by Torczon will be introduced, including the basic idea of the method, as well as the strategy used to adapt the basic algorithm to a parallel system.

Chapter 4

The Multi–Directional Search Method on Parallel Machines

As stated in the previous chapter, the parallelization of nonlinear optimization algorithms has been widely researched and discussed. Among the work that has been done or is being done, the multi-directional search algorithm developed by Torczon turns out to be more adaptable to parallel systems than those based on the inherently sequential algorithms, like the Newton or quasi-Newton methods. This chapter is organized with the first section discussing the basic multi-directional search algorithm itself and the second section showing how to adapt the basic algorithm to an MIMD parallel system. Note that in order to be consistent with Torczon's notation, in this chapter the points are denoted by v instead of x.

4.1 The Basic Steps of the MDS Algorithm

As will be shown later, the multi-directional search (MDS) algorithm is based on the idea of the Nelder–Mead simplex method. Any iteration of the basic algorithm begins with a simplex S in $\mathcal{R}^n$, with vertices $v_0, v_1, \ldots, v_n$. The best vertex v_0 is designated to be a vertex for which $f(v_0) \leq f(v_j)$ for $j = 1, \ldots, n$. The n edges of the simplex connecting the best vertex to the remaining n vertices determine a set of n linearly independent directions. When an iteration is completed, a new simplex S_+ is created, which might lead to the next iteration.

4.1.1 The reflection step

The first move of the iteration is to reflect $v_1, \ldots, v_n$ through the best vertex v_0. The length of each reflection from v_0 to r_i is equal to the length of the edge from v_0 to v_i. Geometrically, this could be viewed as *reflecting* the original simplex through the best vertex to give a new simplex. In Fig 4.1, the reflected vertices are labeled r_1 and r_2. On the other hand, the reflection step in the Nelder–Mead simplex method is done only for the worst vertex through the centroid of the other vertices of the original simplex (Fig. 2.1).

The step is deemed successful if at least one of the new vertices has a function value that is better than the function value at the best vertex in the original simplex. Specifically, for a successful reflection step, the following condition should be satisfied:

$$\min\{f(r_i), \quad i = 1, \ldots, n\} < f(v_0). \tag{4.1}$$

This is because we would like the reflected simplex to produce a new best vertex which is

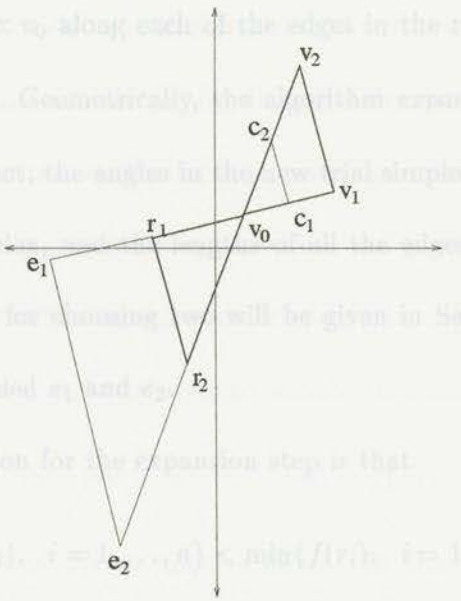

Figure 4.1: The three possible steps given the simplex S with vertices (v_0, v_1, v_2)

better than the current best vertex, v_0. If the best vertex is not replaced, then one more iteration would reflect the simplex back to the original one.

We need to evaluate the function at all the new vertices, $r_1, \ldots, r_n$ in order that the function values can be compared with that at the best vertex. Note that all the function evaluations are independent of each other, which means that they can be easily computed in parallel.

Now two cases may arise: Eq. 4.1 may either be satisfied or not. The two cases will be described more in the following.

4.1.2 The expansion step

If the new vertices do satisfy Eq. 4.1, then it can easily be thought that by considering a longer step size, further improvements may likely be found. Hence, the algorithm takes

a step from the best vertex v_0 along each of the edges in the reflected simplex, but now each step is twice as long. Geometrically, the algorithm *expands* the trial simplex, i.e., the reflected simplex. In fact, the angles in the new trial simplex are still the same as the angles in the original simplex, and the lengths of all the edges have been rescaled by a factor of two. The reason for choosing two will be given in Section 4.2. In Fig 4.1, the expansion vertices are labeled e_1 and e_2.

The acceptance condition for the expansion step is that

$$\min\{f(e_i),\ \ i = 1, \ldots, n\} < \min\{f(r_i),\ \ i = 1, \ldots, n\}. \tag{4.2}$$

If the acceptance condition is satisfied, the next iteration begins with the expanded simplex. There are reasons why the expansion step is not repeated. First, the MDS method only does one expansion step each time. Second, the points obtained by following expansion steps will in fact be considered when the MDS method is customized to a parallel system, which will be explained later in this chapter. Thirdly, if the expansion step continues, the size of the simplex will become increasingly large, which is not a good behavior from the convergence point of view.

On the other hand, if the acceptance condition is not satisfied, since the expansion step is only considered after the reflected simplex has been accepted, the next iteration will start with the reflected simplex.

4.1.3 The contraction step

If the vertices in the reflected simplex do not satisfy the acceptance criterion Eq. 4.1, the question to be considered is whether the steps being taken are too long. Therefore,

the algorithm returns to the original simplex (*not the reflected simplex*) and restarts the search with shorter step sizes. Geometrically, the algorithm simply *contracts* the original simplex towards the best vertex, v_0, by halving every edge in the simplex. Then the next iteration will start with the contracted simplex, which in Fig 4.1 is the simplex (v_0, c_1, c_2).

But before proceeding to the next iteration, it should be confirmed whether or not one of the n new vertices in the contracted simplex satisfies the simple decrease condition:

$$\min\{f(c_i), \quad i = 1, \ldots, n\} < f(v_0). \tag{4.3}$$

By doing this, a new best vertex may be found. In this case, after evaluating the function at the vertices in the contracted simplex and checking Eq. 4.3, the algorithm starts the next iteration with a new best vertex. If, however the contraction step can not produce a new best vertex, at the next iteration the algorithm will again search from the same best vertex v_0 in each of the same n directions considered in the previous iteration, but now with half of the step length. No matter how many times the contracted steps have been taken, if the contracted simplex gives a new best vertex, it is the simplex used to start the next iteration. Also, any time reflection and expansion fail to give a new best vertex, a consecutive sequence of contraction steps would make the process converge to the possible solution.

In summary, the goal of the multi-directional search algorithm is to construct a sequence of best vertices that converges to a minimizer of the function. So it is required that the sequence of the function values at the best vertex be monotonically decreasing. For this reason, a new best vertex is accepted only if it satisfies the simple decrease conditions, as specified in Eq. 4.1, Eq. 4.2, and Eq. 4.3.

Before concluding this section, the stopping criterion for the MDS algorithm needs to be mentioned. In Section 2.2, the stopping criterion suggested by Nelder and Mead was presented. However, later research has shown that (2.2) can lead to premature termination. So for the MDS algorithm, Torczon chose to test if:

$$\frac{1}{\Delta} \max_{1 \leq i \leq n} \| v_i^k - v_0^k \| \leq \epsilon, \tag{4.4}$$

where v_0^k is the best vertex at the kth iteration and $\Delta = \max(1, \| v_0^k \|)$. This stopping criterion measures the relative size of the simplex by considering the length of the longest edge adjacent to the best vertex of this iteration v_0^k and the algorithm stops when the measure falls below the given tolerance ϵ.

4.2 The MDS algorithm

Before presenting the MDS algorithm, it is worth mentioning that there are two arguments, μ and θ, which can affect the performance of the algorithm, where μ is the expansion scale factor and θ is the contraction scale factor. The two factors function similarly to those in the Nelder–Mead simplex method.

In the following description of the algorithm, the stopping criterion refers to (4.4) and the superscripts are used to illustrate the iteration number. It is assumed that $\theta = \frac{1}{2}$, while $\mu = 2$, which will be justified later. It also needs to be noted that in the algorithm the reflection step only changes the directions, which means each step length in the reflection step remains the same as the length of the corresponding edge in the original simplex. Therefore the reflection step is not affected by the choice of μ and θ.

The following is the MDS algorithm.[1]

Given an initial simplex, S_0, *with vertices* $(v_0^0, v_1^0, \ldots, v_n^0)$, $\mu \in (1, +\infty)$, *and* $\theta \in (0, 1)$.

$\min \leftarrow \arg_i \min\{f(v_i^0), \quad i = 0, \ldots, n\}$;

Swap $v_{\min}^0$ and v_0^0;

For $k = 0, 1, \ldots$

{if (the stopping criterion is met) quit;

for $i = 1, \ldots, n$ /*reflection step

{ $v_i^{k+1} \leftarrow 2v_0^k - v_i^k$;

calculate $f(v_i^{k+1})$;}

if $(\min\{f(v_i^{k+1}), \quad i = 1, \ldots, n\} < f(v_0^k))$ then

for $i = 1, \ldots, n$ /* expansion step

{ $v_{e_i}^k \leftarrow v_0^k + \mu(v_0^k - v_i^k)$;

calculate $f(v_{e_i}^k)$; }

if $(\min\{f(v_{e_i}^k), \quad i = 1, \ldots, n\} < \min\{f(v_i^{k+1}), \quad i = 1, \ldots, n\})$

for $i = 1, \ldots, n$

$v_i^{k+1} \leftarrow v_{e_i}^k$,

else

for $i = 1, \ldots, n$ /* contraction step

{ $v_i^{k+1} \leftarrow (1 - \theta)v_0^k + \theta v_i^k$;

calculate $f(v_i^{k+1})$;}

endif

$\min \leftarrow \arg_i \min\{f(v_i^{k+1}), \quad i = 1, \ldots, n\}$;

if $(f(v_{\min}^{k+1}) < f(v_0^k))$, swap $v_{\min}^{k+1}$ and v_0^k

}

The choice of θ and μ in the MDS algorithm was based on the suggestion of Nelder and

[1] Here, the reflection vertices r_i and the contraction vertices c_i at the kth iteration are denoted by v_i^{k+1}.

Mead. Later, more tests have been conducted by other researchers, which indicate that the optimal choice of scale factors is dependent on the function to be minimized, as is the best choice of the initial simplex. If experience with a certain class of problems indicates that a different choice of scaling factors gives better results, it is certainly reasonable to change the scaling factors to reflect this information. However, there is another reason why Torczon chose $\theta = \frac{1}{2}$ and $\mu = 2$, which is that this choice makes it possible to use only integer coefficients in storing the coordinates of the vertices, as will be explained in the next section.

Along with the MDS algorithm, Torczon developed a convergence theory for the method [Tor90, Tor93]. According to the convergence analysis, for the MDS algorithm, the shape of the simplices and the values of θ and μ which determine the step sizes remain fixed across all iterations. This means that by looking at each vertex in each of the trial simplices generated at the previous iteration (starting from the original simplex) as potentially the "best", all the points to be possibly visited[2] can be determined *a priori* and all possible simplices to be considered can be predicted. Eventually, all the points that can possibly be visited generate a grid which can be computed—without any knowledge of function information—from any given initial simplex. Once the grid has been set up, only those points on this grid will be considered. With the assumption that the function be continuously differentiable (even though the algorithm does not explicitly compute derivatives), Torczon proved the existence of an upper bound and a lower bound on the lengths of the edges of the simplices. Furthermore, by enforcing the simplices to be contained in

[2]By *visiting* a point, we mean evaluating the function value at this point.

a compact set, it has been proven that there are always a finite number of points in the grid. Therefore the multi-directional search algorithm can visit only a finite number of points; hence it converges to a solution after a finite number of steps.

4.3 The Parallel Version of the Algorithm

In the previous section, the multi-directional search algorithm was presented, but so far nothing has been said about how this algorithm can be customized to accommodate more than one processor or even a large number of processors. As well, can it be adapted to very powerful processors? In other words, how to make this algorithm work on parallel machines is our concern at the moment.

The strategy employed is quite simple—look ahead to subsequent possible iterations of the algorithm until a sufficient number of vertices has been generated so that all available processors can be kept busy, as we now explain.

First, by combining the three fundamental steps in the basic multi-directional search algorithm, a *core* step can be obtained. The core step is in fact completed by simultaneously doing the reflection, expansion, and contraction steps from the basic algorithm. It is not hard to see that the core step requires $3n$ independent function values at each iteration and each point belongs to one constructed simplex. Suppose we have $3n$ processors available, all the $3n$ function values can be computed simultaneously. In the two-dimensional example given in Fig. 4.1, at the core step, the algorithm computes the function values at six new vertices r_1, r_2, e_1, e_2, c_1 and c_2 simultaneously. Then choosing from those six, the vertex, denoted by v_0^+, is obtained which produces the best function value. S_+ is used to

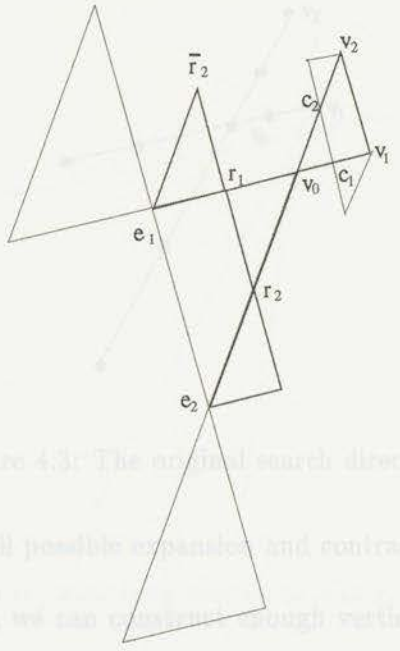

Figure 4.2: The core step with all the reflection steps for the next iteration

denote the simplex that v_0^+ belongs to. If $f(v_0)$ is still the least function value, then S_+ must be the contraction simplex. Note that the idea of core step may actually produce a sequence of iterates different from that produced by the basic method in the last section. For example, the choice of S_+ might now be the expansion simplex even if it would never have been constructed in the basic algorithm in the cases when the reflection step fails to produce any better points.

To take advantage of more processors, we simply continue the look-ahead to subsequent iterations of the algorithm. For instance, still using the example of Fig. 4.1, if each of r_1, r_2, e_1, e_2, c_1, c_2 or v_0 were to become v_0^+, we could continue to construct all the reflection simplices that *could* be considered at the *next* iteration, as shown in Fig. 4.2. Consider the case when r_1 becomes v_0^+ and S_+ is (r_1, v_0, r_2). The reflection simplex from S_+ is

Figure 4.3: The original search directions

now $(r_1, e_1, \bar{r}_2)$. Of course, all possible expansion and contraction vertices from v_0^+ can be included as well. Therefore, we can construct enough vertices to keep all the processors busy.

If the processors we use are computationally powerful processors, it is possible that the cost of a single function evaluation on a single processor could be swamped by the cost of the communication between processors. In this case, we can assign several function evaluations in one message-passing to a single processor to balance the communication cost.

It can be easily observed that all the vertices constructed in one core step lie along n directions determined by the n edges adjacent to v_0, as seen in Fig. 4.3, where we suppress the simplices to show only the points. Thus, for $n = 2$, the core step can be viewed as a rudimentary line search consisting of three steps along each of two directions. Proceeding to the next iteration and considering all possible core steps, new line searches are introduced, besides refining the search along the original n directions, as can be seen in Fig. 4.4.

Figure 4.4: The search directions with new steps and additional search directions

Thinking in another way, we imagine the situation that, given an initial simplex, no information about the function values at any of the vertices in the simplex is available. This means that there is no way of identifying the "best" vertex in the simplex. However, even without identifying the best vertex, all the simplices that *might* be generated in the first iteration can still be determined. This means, all the points that can *possibly* be visited can be determined, *a priori*. To do this, simply allow each of the vertices in the original simplex to be "best" and consider all the possibilities, as shown in Fig. 4.5.

Either of the above ways illustrate that it is possible to systematically enumerate all the vertices that can possibly be visited in the next several core steps. Since in each iteration the same look-ahead strategy is adopted and every vertex can be represented as a *fixed* linear combination of v_0 and the edges adjacent to v_0, it is unnecessary to regenerate these coefficients at every iteration. Let the coefficients for constructing the vertex v_i be $a_{i_1}, \ldots, a_{i_n}$. A fixed template of these coefficients can be defined before the

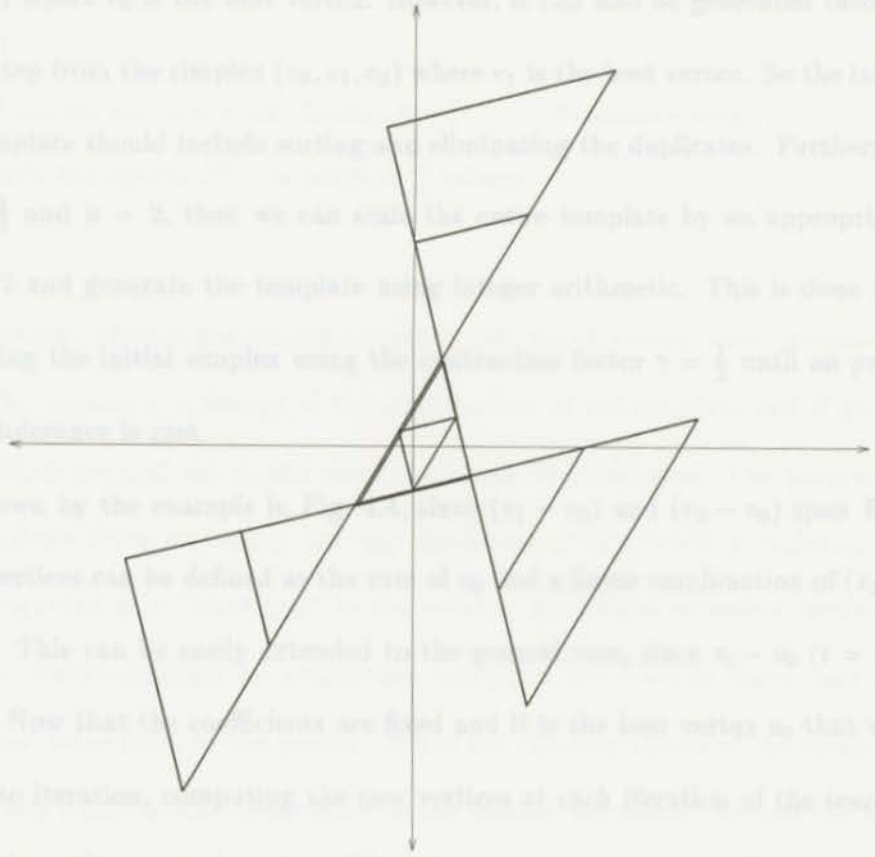

Figure 4.5: Enumerating the vertices when the best vertex is not known

search actually begins. The size s of the template actually indicates how many points the look-ahead generates.

It is not hard to see that some points may be generated by different steps from different "best" points. As in Fig. 4.2, r_1 is first generated by a reflection step from the simplex (v_0, v_1, v_2) where v_0 is the best vertex. However, it can also be generated through a contraction step from the simplex (v_0, e_1, e_2) where e_1 is the best vertex. So the initialization of the template should include sorting and eliminating the duplicates. Furthermore, if we use $\theta = \frac{1}{2}$ and $\mu = 2$, then we can scale the entire template by an appropriately large power of 2 and generate the template using integer arithmetic. This is done by keeping on shrinking the initial simplex using the contraction factor $\gamma = \frac{1}{2}$ until an pre-specified step size tolerance is met.

As shown by the example in Fig. 4.4, since $(v_1 - v_0)$ and $(v_2 - v_0)$ span $\mathcal{R}^2$, each of the new vertices can be defined as the sum of v_0 and a linear combination of $(v_1 - v_0)$ and $(v_2 - v_0)$. This can be easily extended to the general case, since $v_i - v_0$ $(i = 1, 2, \ldots, n)$ span $\mathcal{R}^n$. Now that the coefficients are fixed and it is the best vertex v_0 that varies from iteration to iteration, computing the new vertices at each iteration of the search reduces to computing a linear combination of the edges adjacent to the new best vertex.

The following is a statement of the algorithm as implemented on a master–slave parallel system, which is different from the original MDS algorithm.

Given an initial point v_0^0, and an initial simplex S^0 with vertices $(v_0^0, v_1^0, \ldots, v_n^0)$.

Initialize template with the size s preset

$k \leftarrow -1$ /* the iteration number */

while (stopping criterion is not satisfied) **do**

$k \leftarrow k+1$

for $i = 1, \ldots, s$ **do**

$v_i^{k+1} \leftarrow v_0^k + a_{i_1}(v_1^k - v_0^k) + \ldots + a_{i_n}(v_n^k - v_0^k)$

send out v_i^{k+1} to a slave processor

end

find out the new best point, denoted by v_0^{k+1} /*communication */

identify the simplex S^{k+1} to which v_0^{k+1} belongs

end

Assume that there is one *master* processor with which several *slave* processors communicate. The master is in charge of the initialization of the template and it generates the vertices which are sent out to the slave processors for evaluation. The programs running on the multiple slave processors are only duplicates of the function evaluation program, which is regarded as a "black box" by the master. Each slave processor receives from the master the vertex assigned for evaluation, computes the function value, and then sends the function value back to the master. When the function values of the vertices sent out all come back from the slave processors to the master, the master chooses the best vertex, from where a new iteration starts.

Now that we have the parallel version of the MDS algorithm, we will introduce our thesis work based on its master-slave version. Specifically, our concerns lie in two aspects:

- The data structure used in the implementation of the algorithm;
- The strategy to use quadratic approximation models to accelerate the convergence process.

Before concluding this chapter, the practical use of the algorithm should be mentioned. In fact, Torczon's experience [DT91] of experimenting with a data set provided by researchers at the University of Texas at Houston, M.D. Anderson Cancer Center, and the University of Texas Medical Branch at Galveston, shows that the MDS algorithm is very useful as an experimental tool.

Chapter 5

The MDS Algorithm with Restart

This chapter describes the work done for the thesis, which consists of two parts. First, a strategy has been developed to incorporate the idea of using a quadratic model to improve the convergence of the MDS method by restarting from quadratic approximation minimizers. We call our new algorithm *the MDS algorithm with Restart* (MDSR). Second, by using a tree structure to replace the large matrix of visited points, we successfully reduced the space/time cost.

5.1 The MDS Algorithm with Restart

The implementation of the MDS algorithm on a master–slave multiprocessor system makes the master processor act as a coordinator of the system. With the assumption that the function evaluation process is very complicated or time-consuming, the master processor has to wait a considerable amount of time for all the points sent out to return with their function values. It is the primary motivation of our algorithm to make the master

processor do some useful work instead of being idle waiting. As we will show later, by incorporating the idea of using a quadratic model in the MDS algorithm, the convergence can be improved a lot in some cases, even when the function evaluation is quite simple.

What we do is let the master processor develop a quadratic approximation for the objective function each time a sufficient number of points have returned. Then the minimizer of the quadratic approximation (if it is a good one) is used to guide further searches.

To set up a quadratic approximation for the objective function, it can be seen from Eq. 1.1 that it is G, b and c that need to be determined. Since MDS falls into the direct search category, the only information available is the points and their function values. Therefore, how to obtain G, b and c using only the coordinates of the points and the returned function values is our primary concern.

Take $n = 2$ for example. Let $x = (x_1, x_2)$, where again x_i is the ith element of the variable x. Then Eq. 1.1 can be written as:

$$q(x) = \frac{1}{2}(x_1 \quad x_2) \begin{pmatrix} g_{11} & g_{12} \\ g_{12} & g_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + (b_1 \quad b_2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + c. \tag{5.1}$$

Eq. 5.1 tells us that for $n = 2$, there are altogether six unknowns. (Here, the symmetry of the Hessian matrix has been taken into account to reduce the number of unknowns, since the lower triangle of the Hessian equals the upper triangle.) Therefore six equations are needed to obtain the unknowns for the approximation, which means that six points, together with their function values, have to be available before each approximation can be done. Furthermore, for an arbitrary n, totally $\frac{(n+1)(n+2)}{2}$ points together with their function values are required for each approximation.

In the next part of this section, we'll first show how to obtain G, b and c for the case of $n = 2$ and then generalize this to the arbitrary n–dimensional case. Finally, a statement of our algorithm will be given.

5.1.1 The Quadratic Approximation Approach

Suppose $n = 2$ and we have six points $x_1, x_2, \ldots, x_6$ whose function values are $f_1, f_2, \ldots, f_6$. Let x_{i_j} denote the jth entry of x_i. From Eq. 5.1 we have

$$
\begin{aligned}
q(x_1) &= \frac{1}{2}(x_{1_1} \quad x_{1_2}) \begin{pmatrix} g_{11} & g_{12} \\ g_{12} & g_{22} \end{pmatrix} \begin{pmatrix} x_{1_1} \\ x_{1_2} \end{pmatrix} + (b_1 \quad b_2) \begin{pmatrix} x_{1_1} \\ x_{1_2} \end{pmatrix} + c = f_1 \\
q(x_2) &= \frac{1}{2}(x_{2_1} \quad x_{2_2}) \begin{pmatrix} g_{11} & g_{12} \\ g_{12} & g_{22} \end{pmatrix} \begin{pmatrix} x_{2_1} \\ x_{2_2} \end{pmatrix} + (b_1 \quad b_2) \begin{pmatrix} x_{2_1} \\ x_{2_2} \end{pmatrix} + c = f_2 \\
&\vdots \\
q(x_6) &= \frac{1}{2}(x_{6_1} \quad x_{6_2}) \begin{pmatrix} g_{11} & g_{12} \\ g_{12} & g_{22} \end{pmatrix} \begin{pmatrix} x_{6_1} \\ x_{6_2} \end{pmatrix} + (b_1 \quad b_2) \begin{pmatrix} x_{6_1} \\ x_{6_2} \end{pmatrix} + c = f_6 \ .
\end{aligned}
$$

By rewriting the above equations, we get

$$
\begin{aligned}
\frac{1}{2}x_{1_1}^2 g_{11} + x_{1_1}x_{1_2}g_{12} + \frac{1}{2}x_{1_2}^2 g_{22} + x_{1_1}b_1 + x_{1_2}b_2 + c &= f_1 \\
\frac{1}{2}x_{2_1}^2 g_{11} + x_{2_1}x_{2_2}g_{12} + \frac{1}{2}x_{2_2}^2 g_{22} + x_{2_1}b_1 + x_{2_2}b_2 + c &= f_2 \\
&\vdots \\
\frac{1}{2}x_{6_1}^2 g_{11} + x_{6_1}x_{6_2}g_{12} + \frac{1}{2}x_{6_2}^2 g_{22} + x_{6_1}b_1 + x_{6_2}b_2 + c &= f_6 \ .
\end{aligned}
$$

Writing the above equations in matrix form, we then obtain

$$\begin{pmatrix} \frac{1}{2}x_{1_1}^2 & x_{1_1}x_{1_2} & \frac{1}{2}x_{1_2}^2 & x_{1_1} & x_{1_2} & 1 \\ \frac{1}{2}x_{2_1}^2 & x_{2_1}x_{2_2} & \frac{1}{2}x_{2_2}^2 & x_{2_1} & x_{2_2} & 1 \\ \frac{1}{2}x_{3_1}^2 & x_{3_1}x_{3_2} & \frac{1}{2}x_{3_2}^2 & x_{3_1} & x_{3_2} & 1 \\ \frac{1}{2}x_{4_1}^2 & x_{4_1}x_{4_2} & \frac{1}{2}x_{4_2}^2 & x_{4_1} & x_{4_2} & 1 \\ \frac{1}{2}x_{5_1}^2 & x_{5_1}x_{5_2} & \frac{1}{2}x_{5_2}^2 & x_{5_1} & x_{5_2} & 1 \\ \frac{1}{2}x_{6_1}^2 & x_{6_1}x_{6_2} & \frac{1}{2}x_{6_2}^2 & x_{6_1} & x_{6_2} & 1 \end{pmatrix} \begin{pmatrix} g_{11} \\ g_{12} \\ g_{22} \\ b_1 \\ b_2 \\ c \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{pmatrix}. \tag{5.2}$$

It can be easily seen that solving the linear system (5.2) will determine all the required coefficients G, b and c of the quadratic model [Wat91], and thus will set up the quadratic approximation. The solvability of the linear system (5.2) will be discussed later.

Then the next step is straightforward, to compute the minimizer x^* of the quadratic model simply by solving another linear system

$$Gx^* = -b.$$

For the case of an n-dimensional objective function, let $l = \frac{(n+1)(n+2)}{2}$. Suppose we have l points with their function values. By analogy to the two dimensional case above, we can obtain

$$Ag = f \tag{5.3}$$

where

$g = (g_{11}, \ldots, g_{1n}, g_{22}, \ldots, g_{2n}, \ldots, g_{nn}, b_1, \ldots, b_n, c)^T$,

$f = (f_1, f_2, \ldots, f_k, \ldots, f_l)^T$, and

$$A = \begin{pmatrix} \frac{1}{2}x_{1_1}^2 & x_{1_1}x_{1_2} & \cdots & x_{1_1}x_{1_n} & \frac{1}{2}x_{1_2}^2 & x_{1_2}x_{1_3} & \cdots & \frac{1}{2}x_{1_n}^2 & x_{1_1} & \cdots & x_{1_n} & 1 \\ \frac{1}{2}x_{2_1}^2 & x_{2_1}x_{2_2} & \cdots & x_{2_1}x_{2_n} & \frac{1}{2}x_{2_2}^2 & x_{2_2}x_{2_3} & \cdots & \frac{1}{2}x_{2_n}^2 & x_{2_1} & \cdots & x_{2_n} & 1 \\ \vdots & \vdots & \ddots & & \vdots & \ddots & & \vdots & & \ddots & \vdots & \\ \frac{1}{2}x_{k_1}^2 & x_{k_1}x_{k_2} & \cdots & x_{k_1}x_{k_n} & \frac{1}{2}x_{k_2}^2 & x_{k_2}x_{k_3} & \cdots & \frac{1}{2}x_{k_n}^2 & x_{k_1} & \cdots & x_{k_n} & 1 \\ \vdots & \vdots & \ddots & & \vdots & \ddots & & \vdots & & \ddots & \vdots & \\ \frac{1}{2}x_{l_1}^2 & x_{l_1}x_{l_2} & \cdots & x_{l_1}x_{l_n} & \frac{1}{2}x_{l_2}^2 & x_{l_2}x_{l_3} & \cdots & \frac{1}{2}x_{l_n}^2 & x_{l_1} & \cdots & x_{l_n} & 1 \end{pmatrix}.$$

Again, we will discuss the solvability of the above linear system later. It is not hard to see that solving the above linear system leads to obtaining the matrix G, as well as b and c. Using Gaussian Elimination, it costs $\frac{1}{3}\left(\frac{(n+1)(n+2)}{2}\right)^3 = O(n^6)$ flops (floating point operations per second) for the above system to be solved. And then, simply by solving the linear system $Gx^* = -b$, which costs another $O(n^3)$ flops, we obtain the minimizer x^* of the quadratic approximation. So the total cost of the quadratic approach is $O(n^6)$ flops. Since the MDS method is mostly used to solve problems of moderate size, i.e., problems with no more than 30 or 40 variables, the above quadratic approach which costs $O(n^6)$ flops is practical. Furthermore, since this linear-system-solving process is essentially performed by the master processor during its "idle" time, the computational cost can also be deemed acceptable, though we have to admit that it does limit the usefulness of this approach.

5.1.2 The MDSR Algorithm

Based on the above ideas, we proposed a modification of the MDS algorithm, namely the MDS algorithm with Restart. Here again, $a_{i_1}, \ldots, a_{i_n}$ are the scalars used to construct the vertex v_i. The superscripts denote the iteration number.

The MDSR Algorithm

Given an initial guess v_0

Set up a simplex S_0^0 *with vertices* $(v_0^0, v_1^0, \ldots, v_n^0)$

Initialize the template with its size s preset

$l \leftarrow \frac{(n+1)(n+2)}{2}$

$f_{\min} \leftarrow \infty$

$k \leftarrow -1$, /* the iteration number */

While (stopping criterion is not satisfied) **do**

$k \leftarrow k+1$

For $i = 1, \ldots, s$ **do** /*construct the vertices */

$v_i^{k+1} \leftarrow v_0^k + a_{i_1}(v_1^k - v_0^k) + \cdots + a_{i_n}(v_n^k - v_0^k)$

send out v_i^{k+1} to a slave processor

end

$i \leftarrow 0$, $j \leftarrow 0$

$m \leftarrow 0$, /* number the quadratic approximation in this iteration */

/* all the vertices sent out return with their function values */

While $(i < s)$ **do**

one vertex v_t^{k+1} returns, $1 \le t \le s$, t may or may not be equal to i

$i \leftarrow i+1$, $j \leftarrow j+1$

/* sufficient number of points for quadratic approximation returned */

If ($j = l$) **then**

choose the l points that produce the l smallest function values

find the quadratic approximation minimizer q_m

if (this is not the first quadratic approximation) and

(this approx. is close enough to the previous quadratic approx.) then

return the quadratic minimizer.

endif

send out q_m to a slave processor

set $j \leftarrow 0$, $m \leftarrow m+1$

endif

/* record the point that produced the least function value in the current iteration */

If $(f(v_t^{k+1}) < f_{\min})$ **then**

$f_{\min} \leftarrow f(v_t^{k+1})$

$v_{\min} \leftarrow v_t^{k+1}$

endif

end/*While $i < s$ */

/* the quadratic minimizer produces better function value */

If $\min(f(q_m), \quad m = 1, \ldots) < f_{\min})$ **then**

set up a new initial simplex S_0^{k+1} at the vertex v_0^{k+1},

where $f(v_0^{k+1}) = \min(f(q_m), \quad m = 1, \ldots)$

else

$v_0^{k+1} \leftarrow v_{\min}$

update the simplex at $v_{\min}$

endif

end/*While*/

return

In our algorithm, at any iteration, the first time a sufficient number, say l, of vertices have returned with their function values, one quadratic approximation is performed and the quadratic minimizer obtained is sent to a slave processor to compute the function value. When another l vertices have returned, we choose from all the vertices already returned those l vertices which have the smallest function values and then perform another approximation, obtaining a new quadratic minimizer. This continues until all the vertices

sent out in this iteration have returned. Next, we choose the vertex that produces the smallest function value in this iteration. However, this vertex can be either a vertex on the grid or a quadratic minimizer. If it is a vertex on the grid, the algorithm goes on as the MDS method does. On the other hand, if the best vertex happens to be one of the quadratic minimizers, which probably does not reside on the grid, we start the next iteration from this off-grid point, with a newly created initial simplex for the next iteration. Its size is taken to be the same as that of the simplex which contains the best on-grid point of this iteration as a vertex. We call this a *Restart*.

Consider the convergence of the MDSR algorithm. If no restart ever occurs, the MDSR algorithm is the same as the MDS algorithm. Then from the original initial simplex, the convergence theory remains valid. However, when a restart point is chosen, this point is likely to be off the original grid. Therefore if a restart ever occurs, the grid is no longer fixed across all iterations. Though there remains some theoretical work to be done on the convergence analysis for the MDSR method, at this stage, in cases when restarting does not happen very frequently, we can regard the searches completed prior to the last restart as a *pre-process* to locate the initial point and the initial simplex for the MDS algorithm. Then from this initial simplex, the convergence of the MDS method—it is the actual method in use now—still applies.

The numerical results for the MDSR algorithm turn out to be better than those of the original MDS algorithm in many cases. The analysis will be presented in the next chapter.

5.1.3 The Singularity of the Coefficient Matrix

In the previous section, the MDSR algorithm was presented. Now we come to the discussion of the singularity of the coefficient matrix, which solely depends on the coordinates of the given points. We wish to thank Professor C. Paige [Pai] of McGill University for suggesting the following approach for identifying singularity of the matrix A in (5.3).

Consider the linear system (5.3). The singularity of A determines whether or not there exist solutions for $Ag = f$. Since A is determined only by the coordinates of the points we use, we expect that if the points are chosen carefully, A can be made to be nonsingular. So now, the discussion has turned to how to choose the points properly. A lemma is presented first.

Lemma 5.1 *In a matrix, adding a multiple of one column to another does not alter the rank of the matrix.*

The lemma is a basic one that can be easily found in any elementary linear algebra text. In fact, adding a multiple γ of column i to column j $(i \neq j)$ is equivalent to multiplying the matrix by an elementary matrix $(I_n + \gamma e_i e_j^T)$, whose rank is n. Here, I_n is the n-dimensional identity matrix.

Now we start with the $n = 2$ case, i.e., setting up a quadratic model if given six points[1] (x_i, y_i), $i = 1, \ldots, 6$. Consider the ith row of the matrix A (where, without loss of generality, the constant coefficients are omitted in the following discussions),

$$x_i^2 \;\; x_i y_i \;\; y_i^2 \;\; x_i \;\; y_i \;\; 1, \quad i = 1, \ldots, 6. \tag{5.4}$$

[1] Here, for $n = 2$, we use (x, y) to denote one point.

Now, suppose there are $k \leq 6$ points lying on the line

$$y = \alpha x + \gamma. \tag{5.5}$$

Without loss of generality, we can always arrange the first k rows of the matrix to correspond to these k points, simply by multiplying the matrix by a permutation matrix if necessary. Using (5.5) to substitute for y in (5.4), the ith row becomes

$$x_i^2 \;\; \alpha x_i^2 + \gamma x_i \;\; \alpha^2 x_i^2 + 2\alpha\gamma x_i + \gamma^2 \;\; x_i \;\; \alpha x_i + \gamma \;\; 1, \quad i = 1, \ldots, k \leq 6.$$

Next, if we first subtract γ times the last column from the fifth column, then subtract γ^2 times the last column from the 3rd column, we obtain

$$x_i^2 \;\; \alpha x_i^2 + \gamma x_i \;\; \alpha^2 x_i^2 + 2\alpha\gamma x_i \;\; x_i \;\; \alpha x_i \;\; 1 \quad i = 1, \ldots, k \leq 6. \tag{5.6}$$

In (5.6), subtract 2γ times the fifth column from the third, and then subtract γ times the fourth column from the second. Now we have a block of k rows of A which is

$$\begin{matrix} x_1^2 & \alpha x_1^2 & \alpha^2 x_1^2 & x_1 & \alpha x_1 & 1 \\ x_2^2 & \alpha x_2^2 & \alpha^2 x_2^2 & x_2 & \alpha x_2 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_k^2 & \alpha x_k^2 & \alpha^2 x_k^2 & x_k & \alpha x_k & 1 \end{matrix}$$

We see the second and third columns are multiples of the first, while the fifth column is a multiple of the fourth. So, this block can have rank of at most 3.

From Lemma 5.1, all the operations done above do not alter the rank of the matrix A. Therefore,

- If $k = 6$, then A has rank of at most 3, so A is singular;

- If $k = 5$, then the block of these five rows has rank of at most 3. The other row,

$$x_6^2 \quad \alpha x_6^2 \quad \alpha^2 x_6^2 \quad x_6 \quad \alpha x_6 \quad 1,$$

if chosen properly, can be independent of the previous five rows. Therefore, A can have rank of at most 4, which also determines that A is singular;

- If $k = 4$, then the block of these four rows has rank of at most 3. Again, even if the other points are chosen to be independent of these four, A can have rank of at most 5, which still determines that A is singular;

- If $k = 3$, then the block of these three rows have rank of at most 3. Now if the other three points are independent of these three, A could have a rank of 6, i.e., A could be *nonsingular*.

However, suppose the other three points happen to be on another single line and suppose the two lines are non-parallel, so the lines can be written as

$$y = \alpha x + \gamma, \quad y = \beta x + \theta. \tag{5.7}$$

As we know, performing a transformation (or rather a *translation*) on y by

$$\bar{y} = y - \theta,$$

the ith row of the matrix A becomes

$$x_i^2 \quad x_i(\bar{y}_i + \theta) \quad (\bar{y}_i^2 + 2\theta\bar{y}_i + \theta^2) \quad x_i \quad \bar{y}_i + \theta \quad 1.$$

Now if we subtract multiples of the last column from the second last column and the third column, then subtract a multiple of the fifth column from the third and a

multiple of the fourth from the second, we can get

$$x_i^2 \quad x_i\bar{y}_i \quad \bar{y}_i^2 \quad x_i \quad \bar{y}_i \quad 1.$$

From Lemma 5.1, we know that the above operations do not affect the rank of the matrix A, that is, translation of the origin does not change the rank of A.

Furthermore, with the form of A unchanged, returning to the non-parallel lines (5.7) and translating the origin to the intersection of the two lines, the two lines are now

$$y' = \alpha x, \quad y'' = \beta x, \tag{5.8}$$

and the matrix A becomes

$$\begin{pmatrix} x_1^2 & x_1y_1' & y_1'^2 & x_1 & y_1' & 1 \\ x_2^2 & x_2y_2' & y_2'^2 & x_2 & y_2' & 1 \\ x_3^2 & x_3y_3' & y_3'^2 & x_3 & y_3' & 1 \\ x_4^2 & x_4y_4'' & y_4''^2 & x_4 & y_4'' & 1 \\ x_5^2 & x_5y_5'' & y_5''^2 & x_5 & y_5'' & 1 \\ x_6^2 & x_6y_6'' & y_6''^2 & x_6 & y_6'' & 1 \end{pmatrix}.$$

Now using (5.8) to substitute for y' and y'', A becomes

$$\begin{pmatrix} x_1^2 & \alpha x_1^2 & \alpha^2 x_1^2 & x_1 & \alpha x_1 & 1 \\ x_2^2 & \alpha x_2^2 & \alpha^2 x_2^2 & x_2 & \alpha x_2 & 1 \\ x_3^2 & \alpha x_3^2 & \alpha^2 x_3^2 & x_3 & \alpha x_3 & 1 \\ x_4^2 & \beta x_4^2 & \beta^2 x_4^2 & x_4 & \beta x_4 & 1 \\ x_5^2 & \beta x_5^2 & \beta^2 x_5^2 & x_5 & \beta x_5 & 1 \\ x_6^2 & \beta x_6^2 & \beta^2 x_6^2 & x_6 & \beta x_6 & 1 \end{pmatrix}.$$

According to Lemma 5.1, without affecting the rank of the matrix, we can subtract α times the first column from the second column and α^2 times the first column from

the third, and then subtract α times the fourth column from the fifth. Then A becomes

$$\begin{pmatrix} x_1^2 & 0 & 0 & x_1 & 0 & 1 \\ x_2^2 & 0 & 0 & x_2 & 0 & 1 \\ x_3^2 & 0 & 0 & x_3 & 0 & 1 \\ x_4^2 & (\beta-\alpha)x_4^2 & (\beta^2-\alpha^2)x_4^2 & x_4 & (\beta-\alpha)x_4 & 1 \\ x_5^2 & (\beta-\alpha)x_5^2 & (\beta^2-\alpha^2)x_5^2 & x_5 & (\beta-\alpha)x_5 & 1 \\ x_6^2 & (\beta-\alpha)x_6^2 & (\beta^2-\alpha^2)x_6^2 & x_6 & (\beta-\alpha)x_6 & 1 \end{pmatrix}.$$

Finally, subtract $(\beta+\alpha)$ times the second column from the third. Then we get

$$\begin{pmatrix} x_1^2 & 0 & 0 & x_1 & 0 & 1 \\ x_2^2 & 0 & 0 & x_2 & 0 & 1 \\ x_3^2 & 0 & 0 & x_3 & 0 & 1 \\ x_4^2 & (\beta-\alpha)x_4^2 & 0 & x_4 & (\beta-\alpha)x_4 & 1 \\ x_5^2 & (\beta-\alpha)x_5^2 & 0 & x_5 & (\beta-\alpha)x_5 & 1 \\ x_6^2 & (\beta-\alpha)x_6^2 & 0 & x_6 & (\beta-\alpha)x_6 & 1 \end{pmatrix}.$$

We see that the rank of this matrix is no larger than 5, which means that in this case too, the matrix A is singular.

From the above analysis, we can see that for the $n = 2$ case, only when no more than three points are on one line can A possibly be nonsingular. Ideally, if we could choose the points accordingly, i.e., choose the x_i's ($i = 1, \ldots, 6$) and y_i's randomly so that no more than three points were on one line, we could obtain a matrix A with the rank of 6, thus obtaining a nonsingular matrix A. However, as can be seen from Fig. 4.4, it is very likely in the MDS algorithm that more than three points lie on one line, since we use the points returned from the template. This is because of the nature of how points are

generated in the MDS algorithm. Therefore, from the quadratic approximation point of view, discarding some of the returned points to ensure that no more than three points are on a line will reduce the probability of the coefficient matrix A being singular. This is the subject of future work and will be discussed further in the next chapter.

Now, consider the n–dimensional case [Ken61], where a line is defined by $n-1$ equations[2]

$$\frac{x_1 - p_1}{t_1} = \frac{x_2 - p_2}{t_2} = \cdots = \frac{x_n - p_n}{t_n}. \tag{5.9}$$

We can write (5.9) as

$$\begin{aligned} x_2 &= \alpha_2 x_1 + \gamma_2 \\ x_3 &= \alpha_3 x_1 + \gamma_3 \\ &\vdots \\ x_n &= \alpha_n x_1 + \gamma_n \end{aligned} \tag{5.10}$$

with $\alpha_i = t_i/t_1, \quad i = 2, 3, \ldots, n$.

Analogously to the $n = 2$ case, suppose $(x_{i_1}, \ldots, x_{i_n}), \quad i = 1, \ldots, k \leq l$ lie on one single line (5.10). Consider the ith row of the matrix A

col.#	1	2	⋯	n	n+1	n+2	⋯	⋯	⋯	$\frac{n(n+1)}{2}$	$\frac{n(n+1)}{2}+1$	⋯	$\ell-1$	ℓ
	$x_{i_1}^2$	$x_{i_1}x_{i_2}$	⋯	$x_{i_1}x_{i_n}$	$x_{i_2}^2$	$x_{i_2}x_{i_3}$	⋯	$x_{i_m}x_{i_j}$	⋯	$x_{i_n}^2$	x_{i_1}	⋯	x_{i_n}	1.

[2] In the n–dimensional line equation, $(p_1, p_2, \ldots, p_n)$ is a point on the line, while $(x_1, \ldots, x_n)$ denotes the point being considered at the moment.

We can substitute for the x_{i_j}'s $(j = 2, \ldots, n)$ using (5.10). Thus,

$$\begin{aligned}
x_{i_1}x_{i_j} &= \alpha_j x_{i_1}^2 + \gamma_j x_{i_1}, && j = 2, \ldots, n; \\
x_{i_m}x_{i_j} &= \alpha_m\alpha_j x_{i_1}^2 + (\alpha_m\gamma_j + \alpha_j\gamma_m)x_{i_1} + \gamma_j\gamma_m, && m = 2, \ldots, n, \quad j = m+1, \ldots, n; \\
x_{i_j}^2 &= \alpha_j^2 x_{i_1}^2 + 2\alpha_j\gamma_j x_{i_1} + \gamma_j^2, && j = 2, \ldots, n; \\
x_{i_j} &= \alpha_j x_{i_1} + \gamma_j, && j = 2, \ldots, n.
\end{aligned}$$

Subtract appropriate multiples of the last column (which are all 1's) from column $n+1$ through the second last one, then subtract multiples of column $(\frac{n(n+1)}{2} + 1)$ (which are x_{i_1}) from column 2 through column $\frac{n(n+1)}{2}$, as well as from column $(\frac{n(n+1)}{2} + 2)$ through the second last column. Then we get

$$x_{i_1}^2 \;\; \alpha_2 x_{i_1}^2 \;\; \ldots \;\; \alpha_n x_{i_1}^2 \;\; \alpha_2^2 x_{i_1}^2 \;\; \alpha_2\alpha_3 x_{i_1}^2 \;\; \ldots \;\; \alpha_2\alpha_n x_{i_1}^2 \;\; \ldots \;\; \alpha_n^2 x_{i_1}^2 \;\; x_{i_1} \;\; 0 \;\; \ldots \;\; 0 \;\; 1, \qquad (5.11)$$
$$i = 1, \ldots, k$$

We can see from (5.11) that in this block of k rows, there are at most three columns that are independent of each other—the column of $x_{i_1}^2$ $(i = 1, \ldots, k)$, the column of x_{i_1} $(i = 1, \ldots, k)$ and the last column, which is all 1's. Therefore this block of k rows has a rank of at most 3. With Lemma 5.1, we know that the above operations do not alter the rank of the matrix. In considering the rank of A, it is not hard to see that

- If all l points are on one line, i.e., $k = l$, A can have rank of at most 3. Hence, A is singular;
- Suppose k points are one line, $k \geq 4$. Then the block of these k rows has rank of at most 3. Since a block of four or more rows has rank of at most 3, A is singular;
- If only three points are on one line, $k = 3$, the three rows have rank of at most 3. If the remaining $l - 3$ rows can be chosen to be independent, then A can have rank of l, i.e., A may be nonsingular in this case.

What we have obtained is a general strategy that for an arbitrary n-dimensional problem, if the points are chosen carefully, with no more than three points on one line, we can make the matrix A nonsingular. Before this idea can be incorporated into the MDSR algorithm, more work needs to be done here to obtain a sufficient condition for the matrix A to be nonsingular.

5.2 Using A Tree Structure For The Implementation of The MDS

Another issue of this thesis work came from studying implementation details of Torczon's MDS algorithm. We noticed that a matrix was used in her implementation to store the coefficients of all the visited points, with each column corresponding to the n coordinates of a point. It is quite obvious that there is no reason for a point to be visited more than once, which means the function value at each of the points should only be computed once. Therefore, each time a new point is generated, it must be checked to see whether the new point is a duplicate of a previous one before deciding to visit it or not. Evidently, this check (called *duplicate–elimination*) requires that the coefficients of the new point be compared with those of all the points already visited.

In Torczon's implementation, each duplicate–elimination is a sequential search starting from the very first column (row in Table 5.1) of the matrix (i.e., the first visited point). For each point–point comparison, in the worst case, all the n coordinate coefficients may have to be checked. With the search proceeding, the number of visited points increases,

x_1	x_2	x_3	x_4
0	0	0	2048
0	0	2048	0
0	2048	0	0
2048	0	0	0
0	0	0	0
-2048	0	0	0
0	-2048	0	0
0	0	-2048	0
0	0	0	-2048
1024	0	0	0
⋮	⋮	⋮	⋮
-1536	-1024	1024	-512
-1536	-1024	0	-512
512	-2048	0	-512
0	-2048	0	-512
0	-2048	512	-512
256	-1024	0	-768
0	-768	0	-768
0	-1024	256	-768
⋮	⋮	⋮	⋮

Table 5.1: Coefficient matrix of visited points in Torczon's implementation

which makes the duplicate–elimination become more and more expensive, especially when n is large, say $n > 10$. What's more, it has also been observed that there exists a great deal of redundancy in the matrix, resulting from various points sharing some values of the n coordinate coefficients. Indeed, it is often the case that two points differ in only one coordinate. Table 5.1 shows part of the matrix containing the combination coefficients for some visited points for the function $f(x) = x_1^2 + x_2^2 + x_3^2 + x_4^2$. Also related to the storage concern is that the matrix A in (5.3) will grow very quickly as the dimension of the problem increases.

Motivated from what is stated above, we choose to replace the matrix by a tree structure—*a binary tree*. A binary tree is one in which each node has two subtrees, a left subtree and a right subtree; either or both of the subtrees may be null [BS80, Wyk90]. Using a binary tree, when a point is to be stored, only the coordinates (often only one coordinate) that are different from the parent node need to be recorded. So a reduction in the amount of redundancy in storing the matrix can be achieved.

It is straightforward to represent a binary tree with a linked list using pointers. However, as the most widely used programming language in the numerical field is FORTRAN, which does not have the data type of pointer, we have to turn to the sequential representation of a binary tree instead. In fact, even for the languages like C or Pascal, in which the pointer data type is supported, the sequential representation of a binary tree can be valuable to some extent in that it is more efficient in space.

A binary tree is usually linked with a well-defined ordering, that is, with the following definition of $a < b$, the nodes in the left subtree of a node have smaller values than the

node has, while the nodes in the right subtree of a node have larger values.

Def 5.1 *Given two n-dimensional arrays a and b, then*

$a < b$ *iff for some* k, $1 \leq k \leq n$, $a_k < b_k$ *and* $a_i = b_i$ *for all* $1 \leq i < k$.

In our implementation, the binary tree is represented sequentially in a block of contiguous memory locations. The nodes are stored as follows (see Fig. 5.1): the first n memory locations contain the n coordinate coefficients of the first visited point, which is the root node of the tree. The subsequent two locations contain the indices of the left and right children of the root node, respectively. The following memory locations contain the non-root nodes. Each non-root node records only the coordinate coefficients that are different from those of its parent, i.e., the coordinate index and the corresponding value, with the last coordinate index negative, denoting the end of the index-value pairs list of the node. Then the following two entries respectively contain the indices of the left and the right children of this node. In the case where any of the two children is null, zero is put in the location for this child.

For the data in Table 5.1, the first several points are organized in the binary tree as shown in Fig. 5.2.

Bearing the above structure in mind, we now look at the traversal of the binary tree to determine whether a newly generated point is already in the tree or not, and how to fit it in if the new point is not already present. To illustrate the process, a statement of the algorithm we used is presented.

Index	Info. contained
1	n coordinates of the point
⋮	
n	
n+1	lc: the left child
n+2	rc: the right child
⋮	⋮
lc	i_1: coord index
lc+1	val_1: coord value
⋮	⋮
lc+2$(k-1)$	$-i_k$: **last coord index of this node**
lc+2$k-1$	val_k: coord value
lc+2k	lc-l: the left child
lc+2$k+1$	**0: null right child**
⋮	⋮
rc	coord index
⋮	⋮
lc-l	coord index
⋮	⋮

Figure 5.1: The sequential representation of the binary tree

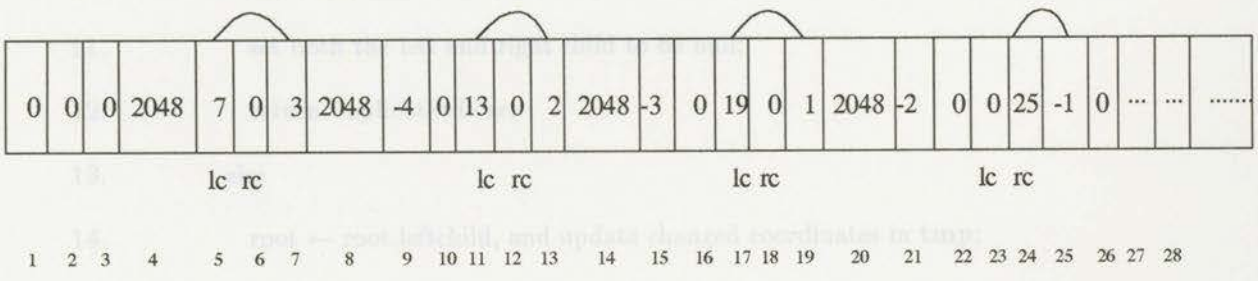

Figure 5.2: The Sequential Representation of a binary tree

The algorithm for maintaining the binary tree

Input:

point(n): *the newly generated point;*

tree(): *the sequential array that stores the binary tree;*

tmp(n): *the temporary array containing the n coordinates of the current root node;*

Output:

duplicate:

true: the point already in the tree;

false: the point was not in the tree and has been inserted.

```
1. if ( tree() is null ) then /** root node **/
2.    simply store all the n coordinates of point in the tree;
3.    set both the left and right child to be null;
4.    return duplicate=false;
5. else
6.    tmp ← the vector at the root of the tree;
7.    while (tmp ≠ point) do
8.          if point < tmp (using Def 5.1) then /** left branch **/
9.             if (root.leftchild = null) then
10.               insert the new point here, only record the index-value pairs;
11.               set both the left and right child to be null;
12.               return duplicate=false;
13.            else
14.               root ← root.leftchild, and update changed coordinates in tmp;
15.            endif
16.         else /* point > tmp */ /** right branch **/
```

```
17.          if (tmp.rightchild = null) then
18.            insert the new point here, only record the index-value pairs;
19.            set both the left and right child to be null;
20.            return duplicate=false;
21.          else
22.            root ← root.rightchild, and update changed coordinates in tmp;
23.          endif
24.        endif
25.  end/*while*/
26.  return duplicate=true;
27.endif
```

The performance of this method of using a tree structure is hard to predict analytically, because different objective functions generate different and unpredictable sets of visited points. Therefore, numerical tests are performed to compare the storage and the cost required using the tree representation against that of using Torczon's matrix representation. The testing results will be presented in the next chapter.

Chapter 6

Performance of our Method and Future Work

In the previous chapter, we discussed the ideas of our work and gave the algorithm details. In this chapter, we present the results of our numerical experiments, comparing our algorithm with Torczon's master-slave version of the MDS algorithm.

6.1 Preliminaries

As we have already noted in the previous chapters, the difference between our work and Torczon's work lies in two areas:

- The MDS algorithm vs. the MDSR algorithm;
- A large matrix vs. a tree structure for storing the coefficients of visited points.

To test our MDSR algorithm, we used standard problems from the optimization literature (J.J. Moré, B.S. Garbow and K.E. Hillstrom [MGH81]), none of which is expensive to compute. Though the problems we chose to test for the MDSR method are of small dimension, as per the discussion in Section 5.1.3, they will give us some indication of the performance of our algorithm. To measure this performance, we adopted the main measure Torczon used in her implementation of the MDS algorithm, i.e., the number of iterations and the number of function evaluations, used to return a solution for a specified tolerance level.

On the other hand, in order to show the performance of our tree structure, we used both the total number of comparisons between each pair of coordinate coefficients and the number of storage entries used (four bytes per entry). Since the larger the dimension of the objective function, the more our method of using a tree structure may gain, we chose to test some problems with variable dimensions.

In summary, our tests were designed to address the following questions:

- Is the performance of MDSR better than that of MDS?
- How much can we gain by using a tree structure instead of a large matrix?

6.1.1 The test problems

According to the above, our numerical experiments fell into two categories. One concerned the performance of our MDSR algorithm, which included tests on such classic problems as the Beale function, the Rosenbrock function and the Sum function, all of which are of dimension two. Another category of tests dealt with the performance of using a tree

structure, which included tests on the extended Rosenbrock function, the extended Powell singular function and the variably dimensioned function.

In order to define the test functions we chose, we use the following general format:

Name of the function [abbreviation]

(a) Dimensions

(b) Function definition

(c) Standard starting point (designated x_0)

(d) Minima

Then the test functions are:

1. Beale function [Beale]

   (a) $n = 2$

   (b) $f(x) = (1.5 - x_1(1 - x_2))^2 + (2.25 - x_1(1 - x_2)^2)^2 + (2.625 - x_1(1 - x_2^3))^2$

   (c) $x_0 = (1, 1)$

   (d) $f = 0$ at $(3, 0.5)$

2. The Sum function [Sum]

   (a) n variable

   (b) $f(x) = \sum_{i=1}^{n} x_i^2$

   (c) $x_0 = (2, 0.2)$

   (d) $f = 0$ at the origin

3. Extended Rosenbrock function [Ext-Rosenbrock]

   (a) n variable but even

   (b) $f_{2i-1}(x) = 10(x_{2i} - x_{2i-1}^2)$

   $f_{2i}(x) = 1 - x_{2i-1}$

$f(x) = \sum_{i=1}^{n/2}(f_{2i-1}(x)^2 + f_{2i}(x)^2)$

(c) $x_0 = (-1.2, 1, -1.2, 1, \ldots)$

(d) $f = 0$ at $(1, \ldots, 1)$

4. Extended Powell singular function [Ext-Powell]

(a) n variable but a multiple of 4

(b) $f_{4i-3}(x) = x_{4i-3} + 10x_{4i-2}$

$f_{4i-2}(x) = 5^{\frac{1}{2}}(x_{4i-1} - x_{4i})$

$f_{4i-1}(x) = (x_{4i-2} - 2x_{4i-1})^2$

$f_{4i}(x) = 10^{\frac{1}{2}}(x_{4i-3} - x_{4i})^2$

$f(x) = \sum_{i=1}^{n/4}(f_{4i-3}(x)^2 + f_{4i-2}(x)^2 + f_{4i-1}(x)^2 + f_{4i}(x)^2)$

(c) $x_0 = (3, -1, 0, 1, 3, -1, 0, 1, \ldots)$

(d) $f = 0$ at the origin

5. Variably dimensioned function [Variable]

(a) n variable.

(b) $f_i(x) = x_i - 1,\ i = 1, \ldots, n$

$f_{n+1}(x) = \sum_{j=1}^{n} j(x_j - 1)$

$f_{n+2}(x) = (\sum_{j=1}^{n} nj(x_j - 1))^2$

$f(x) = \sum_{i=1}^{n+2} f_i^2(x)$

(c) $x_0 = (1 - \frac{1}{n}, 1 - \frac{2}{n}, \ldots, 0)$

(d) $f = 0$ at $(1, \ldots, 1)$

6.1.2 Varying the dimension

Varying the dimension of the test problems was straightforward. Since the extended Rosenbrock function requires that n be even, and the extended Powell singular function

requires that n be a multiplier of four, we limited our tests accordingly.

However, at this stage, as the current strategy in the MDSR algorithm for choosing points for a quadratic approximation is quite simple, we restricted our tests of the MDSR algorithm to two-dimensional problems. In fact, when testing higher dimensional problems, we found that no restart ever occurred. This observation led us to the discussion in Section 5.1.3, from which we determined why the MDSR algorithm does not do better than the MDS algorithm for higher dimensional problems. The reason is that, for higher dimensional problems, the matrices A in (5.3) determined by the coordinates of the chosen points were all nearly singular, causing all the fitted quadratic models based on these points to be rejected by the algorithm. Thus no restarting point has ever been generated. So future work can be done on reducing the probability of A being singular.

Since at this point, for higher dimensional problems, the MDSR method performs the same as the MDS method does, varying the dimension of the problems is mainly done for testing the tree structure.

6.1.3 Decreasing the accuracy tolerance

In Section 4.1 we introduced the stopping criteria Torczon used for the MDS method. As we noted, since the length of each edge adjacent to the best vertex relative to the norm of the best vertex is used to measure the size of the simplex, the test for stopping criterion could then be viewed as a step size tolerance test. What is really being tested is the accuracy tolerance, i.e., what accuracy should the step size fall below before the algorithm is stopped? Thus, we can decrease the value of ϵ, which we used in Section 4.1 to denote

the tolerance, to achieve higher accuracy. In our test, we allowed ϵ to vary from 0.5×10^{-2} to 0.5×10^{-7}.

6.1.4 Varying the size of the template

As explained in the previous chapters, the size s of the template indicates how many points the look-ahead considers. The larger the size of the template, the more points are being considered in each iteration, thus making it possible for more processors to be used or for each processor to be allocated more points for function evaluation. Or we can say, the variability of the size of the template shows how adaptable the algorithm is for a parallel system. In our tests, we took the cases where $s = 8$, $s = 16$, $s = 32$ and $s = 64$.

6.2 Numerical Results for the MDSR algorithm

Now we compare the performance of the MDSR algorithm with that of the MDS algorithm, by testing several functions with both the size of the template varying and the accuracy tolerance varying.

Notice that a simple quadratic function, i.e., the Sum function, is one of the choices to be tested for the MDSR algorithm, because it is the characteristics of quadratic functions that initially motivated our MDSR algorithm. Thus using the MDSR method to minimize a quadratic function should exhibit much better performance than using the MDS method. In fact, the MDSR algorithm exactly locates the analytic minimizer if the objective function is a quadratic one, because the fitted quadratic approximation of a set of points is uniquely determined if the matrix A in the linear system (5.3), which depends

only on the coordinates of the set of points, is nonsingular.

Table 6.1 shows the results obtained when testing the Sum function. Since $n = 2$, six points are needed to accomplish a quadratic approximation. Thus, at each iteration, when eight points are considered (as $s = 8$ indicates), the first six points returned might possibly set up a quadratic model and then after the two remaining points visited in this iteration have returned, another quadratic model might possibly be fitted by choosing the best six points from the eight. But in reality, the eight points in the first iteration can not fit a quadratic approximation because the matrix A in (5.3) turns out to be singular in both of the attempts, for the reason explained in Section 5.1.3. So the algorithm has to wait for the six points visited in the second iteration to return. Then a quadratic approximation can be done. And after the remaining two points in this iteration return, another quadratic approximation is successfully done. At this stage, we have two approximations, which for the case of a quadratic function, are both actually the objective function itself because of the uniqueness discussed in the previous paragraph. These two approximations locate exactly the same minimizer, i.e., the analytic minimizer of the objective function.

Next, the MDSR algorithm checks the two consecutive quadratic approximations to see if the two minimizers are close enough. If they are, which is the case here, we regard the quadratic approximation as good enough that we can use the minimizer of one of the quadratics as the minimizer of the objective function we have been seeking. Therefore, the MDSR algorithm will quit right after two approximations are done. So it takes only *two* iterations for $s = 8$ compared with the MDS algorithm's *sixteen* to *forty-one* iterations (for different step sizes). For quadratic problems, the minimum located by the MDSR

method is exact (within roundoff error) whereas the minimum is approached iteratively by the MDS method.

Table 6.1 shows that the MDSR algorithm does work well for a quadratic function. Now, in Table 6.2, we see the case of $s = 8$ for the Beale function, which is not a quadratic function. Again $n = 2$, and six points are needed to accomplish a quadratic approximation. The results in the table show that when the step size tolerance is 0.5×10^{-2}, the MDSR algorithm does no better than the MDS algorithm, as no six points successfully set up a quadratic approximation before the tolerance is met, resulting in no restart occurring. However, as the tolerance decreases, which means that the accuracy of the minima is increased, the MDSR algorithm exhibits better performance in that it converges to a better point than the MDS algorithm (with f_* again much closer to the function value at the analytical minima), and with fewer iterations. Also the number of function evaluations used is much less than that used in the MDS algorithm. This is because when the tolerance decreases, restart does occur. Further, when the search process proceeds into a neighborhood of the actual minima, the quadratic model approximates the objective function very well, thus giving a good approximation of the minima as the next restart point or as the minima of the objective function that is sought. Tables 6.3, 6.4 and 6.5 show the test results for the Beale function with varying sizes of the template. All these tables illustrate the superior behavior of the MDSR algorithm in comparison with that of the MDS method, except the last row in Table 6.5 where machine overflows bring the seemingly "strange" result.

Similar tests were done for the Rosenbrock function, which are shown in Tables 6.6, 6.7,

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(2, 0.2)	8	0.5D-02	2	18	0.43195D-32	16	128	0.67090D-06
(2, 0.2)	8	0.5D-03	2	18	0.43195D-32	20	160	0.14110D-06
(2, 0.2)	8	0.5D-04	2	18	0.43195D-32	27	216	0.60044D-09
(2, 0.2)	8	0.5D-05	2	18	0.43195D-32	31	248	0.72893D-11
(2, 0.2)	8	0.5D-06	2	18	0.43195D-32	36	288	0.40802D-12
(2, 0.2)	8	0.5D-07	2	18	0.43195D-32	41	328	0.17829D-14

Table 6.1: Sum, $s = 8$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(2, 0.2)	8	0.5D-02	7	56	0.14535D-01	7	56	0.14535D-01
(2, 0.2)	8	0.5D-03	13	108	0.10246D-06	32	256	0.28200D-05
(2, 0.2)	8	0.5D-04	16	132	0.764690D-07	39	312	0.88381D-08
(2, 0.2)	8	0.5D-05	21	174	0.13013D-14	43	344	0.42519D-08
(2, 0.2)	8	0.5D-06	24	198	0.13013D-14	46	368	0.42441D-08
(2, 0.2)	8	0.5D-07	27	222	0.68881D-15	64	512	0.12376D-12

Table 6.2: Beale, $s = 8$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(2, 0.2)	16	0.5D-02	5	81	0.12530D-01	5	80	0.12530D-01
(2, 0.2)	16	0.5D-03	16	262	0.33055D-08	17	272	0.15471D-03
(2, 0.2)	16	0.5D-04	18	294	0.11014D-09	26	416	0.19296D-06
(2, 0.2)	16	0.5D-05	20	326	0.35047D-10	34	544	0.19230D-09
(2, 0.2)	16	0.5D-06	22	356	0.36704D-11	36	576	0.139080D-10
(2, 0.2)	16	0.5D-07	24	390	0.55946D-13	41	656	0.28652D-12

Table 6.3: Beale, $s = 16$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(2, 0.2)	32	0.5D-02	10	325	0.23359D-06	9	288	0.43222D-03
(2, 0.2)	32	0.5D-03	11	357	0.23359D-06	13	416	0.37793D-05
(2, 0.2)	32	0.5D-04	14	453	0.16222D-07	16	512	0.60643D-08
(2, 0.2)	32	0.5D-05	15	488	0.27760D-15	20	640	0.20172D-09
(2, 0.2)	32	0.5D-06	15	488	0.27760D-15	24	768	0.19088D-12
(2, 0.2)	32	0.5D-07	15	488	0.27760D-15	27	864	0.11732D-13

Table 6.4: Beale, $s = 32$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(2, 0.2)	64	0.5D-02	7	449	0.23082D-03	7	448	0.23082D-03
(2, 0.2)	64	0.5D-03	10	643	0.13382D-07	9	576	0.26021D-05
(2, 0.2)	64	0.5D-04	12	771	0.54227D-08	12	768	0.60643D-08
(2, 0.2)	64	0.5D-05	13	838	0.20635D-17	15	960	0.59758D-11
(2, 0.2)	64	0.5D-06	16	1030	0.42099D-18	16	1024	0.35992D-11
(2, 0.2)	64	0.5D-07	17	1094	0.15404D-16	21	1344	0.35683D-13

Table 6.5: Beale, $s = 64$

			MDSR			MDS		
x_0	s	steptol	# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(1.2, -1)	8	0.5D-03	21	179	0.29960D-04	16	128	0.22153D-01
(1.2, -1)	8	0.5D-04	29	244	0.78292D-05	81	648	0.15196D-01
(1.2, -1)	8	0.5D-05	34	285	0.72948D-10	81	648	0.15196D-01
(1.2, -1)	8	0.5D-06	37	309	0.16616D-11	81	648	0.15196D-01
(1.2, -1)	8	0.5D-07	49	412	0.66493D-10	81	648	0.15196D-01

Table 6.6: Rosenbrock, $s = 8$

6.8 and 6.9. From the tables, we can see that when $s = 8$ and the tolerance is 0.5×10^{-3}, the MDSR algorithm achieves the function value of 10^{-5} compared with 10^{-2} obtained by the MDS method, which is substantially better, though with five more iterations and fifty more function evaluations. Tables 6.7, 6.8 and 6.9 show the results obtained when varying the size of of the template. In each table, we can see that when the step size tolerance decreases, the MDSR algorithm arrives at considerably better points with considerably fewer iterations and considerably fewer functions, compared with the MDS method.

One thing needed to be mentioned here is that we impose a maximum number of iterations in our tests for the Rosenbrock function. It is for the purpose of testing, since we are only concerned about comparing the two methods. With this limit, the algorithm will be stopped after the number of iterations reaches this upper bound even if the stopping criterion has not been met. The limit was set to be 80 in our tests. Tables 6.6, 6.7, 6.8 and 6.9 illustrate where the MDS method arrives after 80 iterations, in comparison with the MDSR method.

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(1.2, -1)	16	0.5D-03	20	337	0.59199D-05	43	688	0.12740D-01
(1.2, -1)	16	0.5D-04	22	368	0.70771D-06	81	1296	0.82267D-02
(1.2, -1)	16	0.5D-05	25	417	0.75824D-09	81	1296	0.82267D-02
(1.2, -1)	16	0.5D-06	28	465	0.64760D-09	81	1296	0.82267D-02
(1.2, -1)	16	0.5D-07	40	683	0.14636D-10	81	1296	0.82267D-02

Table 6.7: Rosenbrock, $s = 16$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(1.2, -1)	32	0.5D-03	15	499	0.41019D-05	57	1312	0.44772D-02
(1.2, -1)	32	0.5D-04	17	563	0.40770D-05	81	2592	0.15364D-02
(1.2, -1)	32	0.5D-05	20	660	0.28851D-10	81	2592	0.15364D-02
(1.2, -1)	32	0.5D-06	22	724	0.11381D-11	81	2592	0.15364D-02
(1.2, -1)	32	0.5D-07	23	758	0.26144D-13	81	2592	0.15364D-02

Table 6.8: Rosenbrock, $s = 32$

x_0	s	steptol	MDSR			MDS		
			# of iter's	# of evals.	f_*	# of iter's	# of evals.	f_*
(1.2, -1)	64	0.5D-03	14	913	0.81887D-04	15	960	0.10303D-03
(1.2, -1)	64	0.5D-04	19	1237	0.18392D-12	31	1984	0.42001D-04
(1.2, -1)	64	0.5D-05	21	1366	0.36346D-16	81	5184	0.28486D-06
(1.2, -1)	64	0.5D-06	23	1494	0.36346D-16	81	5184	0.53742D-06
(1.2, -1)	64	0.5D-07	20	1302	0.35626D-17	42	2688	0.61019D-10

Table 6.9: Rosenbrock, $s = 64$

6.3 Numerical Results for Using a Tree Structure

To compare the performance of using a tree structure with that of Torczon's approach of using a matrix to store the visited points, we tested the functions with varying dimensions. The basis for the comparison is that using either of the data structures, the final result should be the same. Here we are only concerned about the implementation of the MDS method, with no regard to the MDSR algorithm. And the numbers in the storage column are the number of integer entries, each of which needs four bytes in standard machine representation.

Looking at Table 6.10, we can see that when n is small, say two, using the tree structure does not gain anything in the storage, because quite a number of storage allocations are used to store the location of the left/right child in the sequential representation of the tree structure (i.e., used as the pointer). Another reason is that for small problems, there is not as much redundancy in the matrix as for large problems. However, when the dimension of a problem increases, a tree structure turns out to save a significant amount of storage. With $n = 28$ for example, it consumes roughly one quarter of the amount of storage used by a matrix.

On the other hand, consider the number of comparisons done during the process of duplicate elimination. From Table 6.10, no matter what the dimension of a problem is, whether it is large or small, using a tree structure always needs far fewer comparisons than using a matrix. Tables 6.10, 6.11, 6.12 are similar results obtained when testing the extended Rosenbrock function, varying the size of the template. Further tests on the extended Powell singular function and the variably dimensioned function brought out the

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
2	11	384	24511	877	2553
4	27	12544	172089	2511	12205
8	12	1664	31834	1213	3966
12	23	4608	112831	2369	10315
16	12	3328	49444	1303	5764
20	18	6080	105247	1933	11730
24	26	10368	195691	2781	18117
28	24	11200	234275	2509	25581

Table 6.10: Ext-Rosenbrock, s=16

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
2	7	512	43981	1153	4070
4	13	1792	115738	2543	10539
8	22	5888	352041	4259	18781
12	18	7296	261238	3889	14368
16	12	6656	124865	2495	11292
20	12	8320	123735	2657	11694
24	12	9984	130443	2653	11931
28	13	12544	172089	2811	17532

Table 6.11: Ext-Rosenbrock, s=32

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
2	6	896	117405	2005	6401
4	14	3840	508032	5045	21823
8	30	14992	2047557	11523	53061
12	*13	10752	492015	5555	28662
16	15	14976	554791	5661	21524
20	*12	14980	377425	4453	31160
24	*10	14976	252737	3733	21500
28	*9	14952	195527	3177	24371

Table 6.12: Ext-Rosenbrock, s=64

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
4	15	1024	43349	1335	4093
8	31	4096	199761	3071	15214
12	32	6336	250625	3311	16964
16	*40	10496	461768	4169	23140
20	*40	13120	363286	4057	32900
24	*39	14976	432339	3893	38917
28	*34	14952	282126	3339	60734

Table 6.13: Ext-Powell Singular, s=16

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
4	19	2560	245158	3389	11509
8	21	5632	313125	4409	19621
12	36	14208	972761	7767	37996
16	*30	14976	660301	5815	28672
20	*24	14980	462853	5079	32537
24	*20	14976	357052	3937	19750
28	*17	14952	270695	3367	54324

Table 6.14: Ext-Powell Singular, s=32

n	iter's	Matrix		Tree	
		storage	# of comparisons	storage	# of comparisons
4	16	4352	693183	5727	26410
8	17	9216	781414	6943	30680
12	*20	14988	946349	7869	37029
16	*15	14976	554791	5661	21524
20	*12	14980	361392	4693	17660
24	*10	14976	253190	4031	13904
28	*9	14952	198414	3405	16710

Table 6.15: Ext-Powell Singular, s=64

n	iter's	Matrix		Tree	
		storage	comparisons	storage	comparisons
4	13	896	32514	1269	3419
8	17	2304	62027	1849	7994
12	11	2304	38308	981	5182
16	12	3328	47926	1019	5497
20	*40	13120	324300	4205	36684
24	*39	14976	409370	3785	52465
28	*34	14952	298826	3245	37161

Table 6.16: variable, s=16

n	iter's	Matrix		Tree	
		storage	comparisons	storage	comparisons
4	13	1792	117374	2471	7969
8	27	7168	520338	5451	25485
12	7	3072	49448	1333	4874
16	7	4096	45528	1311	4411
20	*24	14980	452303	4943	39652
24	18	14592	325531	3555	23579
28	14	12992	235262	2441	41827

Table 6.17: variable, s=32

n	iter's	Matrix		Tree	
		storage	comparisons	storage	comparisons
4	11	3072	334818	4131	12675
8	24	12800	1471445	9669	40273
12	*20	14988	954517	8307	27774
16	*15	14976	544146	5763	30932
20	*12	14980	358337	5053	16835
24	8	13824	249617	2943	19621
28	*9	14952	203284	2925	20216

Table 6.18: variable, s=64

results shown in Tables 6.13 through 6.18. (Note that in this section, the $*$'s in the tables designate the cases where the algorithm stops after the maximum number of iterations has been reached.) These results further illustrate how much we can gain using a tree structure to replace the matrix structure in the implementation of the multi-directional search method.

6.4 Conclusions and Future Work

Our tests about the MDSR algorithm and the data structure issue have led us to the following conclusions.

First, from the testing results for the MDSR method, we believe we have the start of a promising algorithm. The performance results for small problems give us reason to believe that the MDSR algorithm can do better than the MDS algorithm. For higher dimensional problems, the MDSR algorithm does no worse than the MDS method in that, when no restart occurs, the MDSR algorithm actually works the same as the MDS

algorithm. However, we must admit that the problem of the singularity of the coefficient matrix discussed in Section 5.1.3 is so serious when the dimension of the problem gets larger, that not even one good quadratic approximation may be obtained if the set of points used is chosen without any checking. It is obvious that there is work remaining to be done to deal with this singularity problem, both theoretically and practically. The next step would be to study more thoroughly the n-dimensional case in order to work out a set of appropriate conditions to determine under what circumstances the coefficient matrix A in (5.3) is singular, and then to propose a strategy so that we can choose the proper set of points in such a way that the fitted quadratic model will be well-defined. Thus a definite, dependable minimizer could be obtained, which is to be a prospective restarting point. Starting from the analysis in Section 5.1.3, we feel that it should be possible to determine such a strategy. In fact, some further preliminary work on extending the singularity results of Section 5.1.3 has already been done by Dr. A. Buckley.

Second, from the tests on the tree structure, we can see that using a tree structure to replace a linear matrix has brought a considerable savings both in the storage and in the number of comparisons needed. From the system point of view, the savings in both areas are important in that there still remain situations where limitations of the memory and CPU time exist. However in our tree structure, we didn't pay any attention to balance conditions on the binary trees. The balance conditions can keep binary trees from becoming too scrawny and imply that a search in a tree on n items can be performed in time of $O(\log(n))$, in comparison with $O(n)$, which results in the worst case of a binary tree becoming a linear list.

Bibliography

[BS80] R.J. Baron and L.G. Shapiro. *Data Structures and Their Implementation.* Van Nostrand Reinhold, 1980.

[DS83] J.E. Dennis, Jr. and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice-Hall, 1983.

[DT91] J. E. Dennis and V. Torczon. Direct search methods on parallel machines. *SIAM J. on Optimization*, 4:448–474, 1991.

[Fle90] R. Fletcher. *Practical Methods of Optimization.* John Wiley and Sons, 1990. second edition.

[Fly66] M. J. Flynn. Very high-speed computing systems. *Proc. IEEE*, 54:1901–1909, Dec 1966.

[Hay78] J.P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, 1978.

[HB84] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[HJ61] R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Jounal of the ACM*, 8:212–229, 1961.

[Int91] Intel Corporation. *iPSC^R/2 and iPSC^R/860 User's Guide*, 1991.

[Ken61] M.G. Kendall. *A Course in the Geometry of n Dimensions.* Charles Griffin & Company Limited, London, 1961.

[MGH81] J. J. Moré, B. S. Garbow, and K. E. Hilstrom. Testing unconstrained optimization software. *ACM Transactions on Mathematical Software*, 7(1):17–41, March 1981.

[NM64] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(1):308–313, 1964.

[Pai] C. Paige. Fitting a quadratic to several points. Personal communication.

[RS90] S. Ranka and S. Sahni. *Hypercube Algorithms with Applications to Image Processing and Pattern Recognition.* Springer-Verlag, 1990.

[SHH62] W. Spendley, G. R. Hext, and F. R. Himsworth. Sequential application of simplex designs in optimisation and evolutionary operation. *Technometrics*, 4:441, 1962.

[Swa72] W. H. Swann. Direct search methods. In W. Murray, editor, *Numerical Methods for Unconstrained Optimization*, pages 13–28. Academic Press, 1972.

[Tor89] V. Torczon. *Multi-Directional Search: A Direct Search Algorithm for Parallel*

Machines. PhD thesis, Rice University, P. O. Box 1892, Houston TX 77251-1892, Houston, Texas, 1989.

[Tor90] Virginia Torczon. On the convergence of the multi-Directional search algorithm. Technical Report TR90-8, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston TX 77251-1892, March 1990.

[Tor93] Virginia Torczon. On the convergence of pattern search methods. Technical Report TR93-10, Department of Mathematical Sciences, Rice University, P. O. Box 1892, Houston TX 77251-1892, June 1993.

[Wat91] D. S. Watkins. *Fundamentals of Matrix Computations.* John Wiley and Sons, 1991.

[Wyk90] C. J. Van Wyk. *Data Structures and C Programs.* Addison-Wesley, 1990.

VITA

Surname: Ma Given Names: Haibo

Place of Birth: Luoyang, China Date of Birth: September 18, 1966

Educational Institutions Attended:

University of Victoria	1991 to 1993
National University of Defense Technology	1984 to 1988

Degrees Awarded:

B.Sc.	National University of Defense Technology	1988

PARTIAL COPYRIGHT LICENSE

Title of Thesis:

MULTI-DIRECTIONAL SEARCH METHOD WITH QUADRATIC MODEL

Author: 

Haibo Ma

August 12, 1993