Network Modelling of Fault Patterns on Parallel Systems

Steven Webster

EPCC–SS94–10

Network Modelling of Fault Patterns on Parallel Systems

Steven Webster

EPCC–SS94–10

September 1994

Abstract

Deformation in the top 10-15 km of the earth's crust is reflected by populations of faults and fractures on all scales. These structures play a major role in determining the permeability of the crust, affecting the storage and migration of hydrocarbons, groundwater and pollutants.

A network modelling approach allows the analysis of appearance and evolution of these faults and fractures to be reduced to a linear system solver. Code previously existed to perform this network modelling, however the run time of this code was in the order of days.

This document describes the attempts to significantly reduce this run-time by parallelising the existing code. The document describes profiling of the existing code, performance analysis of the proposed parallel implementation and a description of the subsequent parallel port.

1 Introduction

Deformation in the top 10-15 km of the earth's crust leads to the appearance of faults and fractures. The formation and evolution of these fault populations is modelled in a novel way by the Department of Geology and Geophysics at the University of Edinburgh. A 2-Dimensional numerical simulation of crustal deformation is performed using a network model.

The network modelling approach allows modelling of large scale deformation of a material which:

- behaves elastically at low levels of stress,
- ruptures in a pattern which depends on heterogeneous material properties

A unique aspect of the model is that ruptured elements can instantaneously heal themselves. These elements can then continue to support stress; this allows us to specifically model faults. This phenomenon is consistent with frictional sliding on shear fractures, and allows us to model the long time scale characteristics of the deformation.

Since run-times for moderately sized problems are currently measured in days on single-user workstations, this project endeavours to utilise high performance computing in order to reduce the simulation time.

Section 2 will provide a background to some of the geology involved in the project, a description of a useful electrical analogue to the network model and a discussion of how these models are described mathematically. Section 3 aims to discuss the existing serial code supplied to the author, while Section 4 gives the results of profiling this code. Section 5 describes the methods for parallelising the code. A performance model of the ideas in Section 5 is given in Section 6. Details of implementing the parallel port are given in Section 7 with a discussion of the performance gains in Section 8. Section 9 highlights further development that could be carried out on this project, while Section 10 discusses the conclusions made by this project.

2 Background

2.1 Geological Problem

The deformations which occur in the top 10-15 km of the earth's crust lead to populations of faults and fractures. Build-up of stress on the crust encourages the plates to move about these faults, causing the plates to "slip": a phenomena known to us as earthquakes.

Figure 1 shows this situation, with 2 plates moving in opposite directions around a fracture.

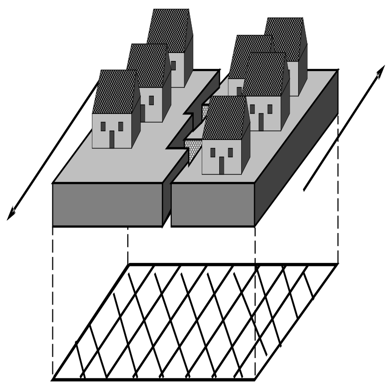
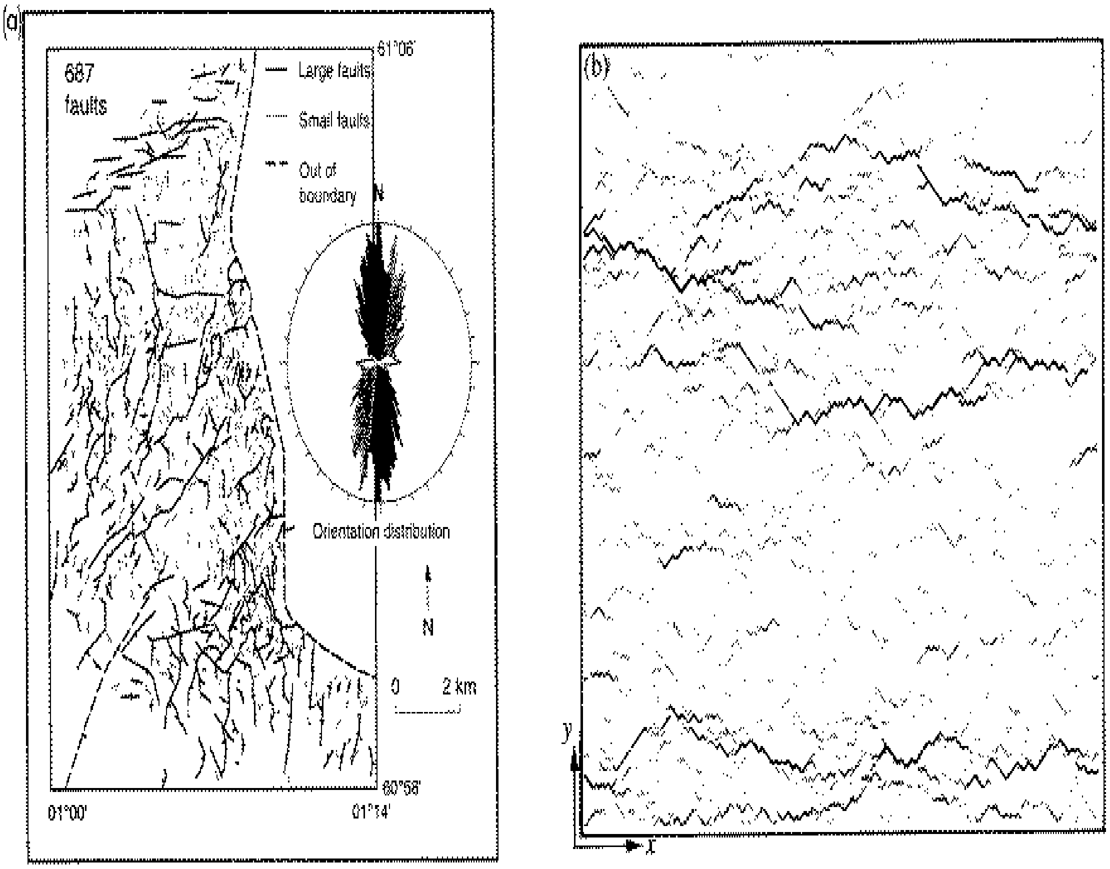


Figure 1: Frictional Sliding

The nature of shear fractures means that once plates start moving about a fracture they do not necessarily continue to do so. Eventually, friction between the 2 plates will cause them to stop moving. At this point, the plates will then support stress once again. This phenomena is known as instantaneous healing – the lattice is said to have annealed. Having annealed, frictional sliding may occur once more, should sufficient stress be placed on the crust.

Modelling the formation and evolution of these faults and fractures is an active area of research. The results play a major role in determining the fracture permeability of the crust, which affects the storage and migration of hydrocarbons, groundwater and pollutants. Similarly, this research is being used to develop more realistic oil reservoir models.



—Interpreted fault map at top reservoir (top Brent) in Pelican field. Faults are classified as a function of their throw ("large" > 15 m, and "small" < 15 m) and of their orientation (defined in the rose diagram).

Fault pattern produced by the rupture model. Shade of grey indicates the amount of accumulated displacement on each ruptured element (roughly equivalent to the number of times it has ruptured). A constant velocity boundary condition is applied along the top edge of the lattice and drives the formation and growth of faults orientated approximately parrallel to the x-direction.

Figure 2: Example Results from Model

The model used at Edinburgh University is a probabilistic model — it does not attempt to predict appearance of faults. Instead, it attempts to more accurately model appearance and evolution of faults based on previously collated data. Figure 2 shows example output of the model against actual real data.

The novelty of the model used is based on the conclusion that long-range elastic effects play an important role in the development of faulting on geological timescales. This is in contrast to other models which analyse crustal deformation using nearest-neighbour interaction models over seismic timescales.

For further discussion of the details of the model, the reader is pointed towards [Patience Cowie and

Sornette, 1993]

2.2 Electrical Analogue

The mathematical equations that are used to describe the problem in geological terms have useful electrical analogues. The governing equations in the geological model are:

- $\sigma = \epsilon$ E, where σ is strain, ϵ is stress and E is Young's Modulus (Hookes Law)
- $\Sigma\sigma = 0$ (Condition of Static Equilibrium)

Replacing occurences of strain with voltage, stress with current and replacing the constant for Young's Modulus with a constant conductance (reciprocal of resistance), we arrive at the following mathematical equations:

- V = IR (Ohm's Law)
- Σ I = 0 (Condition of Charge Conservation)

Thus the network model of the earth's crust can be simulated by an electrical network of resistors, with an external voltage source simulating external pressure. The lattice in figure 1 that the earth's crust was mapped onto can be replaced by a regular lattice arrangement of electrical resistors. This can be seen in figure 3.

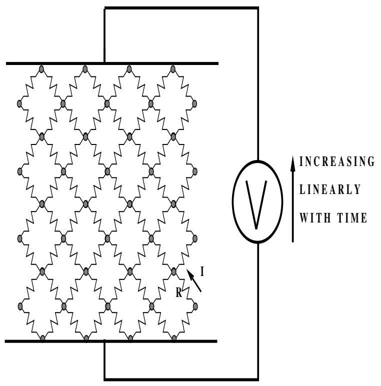


Figure 3: Electrical Resistance Network

In the geological model, when the instantaneous stress on a node exceeds the critical stress threshold, a rupture of the lattice element occurs. To simulate this in the electrical model, each resistor is assigned a "critical" current at random. As the external voltage is increased, current moves through the lattice. When the current in a resistor exceeds it critical value, the resistor "ruptures". A ruptured resistor cannot pass any further current, just as a fault in the earth's crust cannot continue to support stress.

However, a novelty was highlighted previously in the geological model: that of instantaneous healing. Ruptured nodes may heal themselves, thus continuing to support stress. In the electrical model, the ruptured resistor cannot pass any current — it is effectively open circuit. Rather than simply removing the resistor, placing a voltage source in series with the ruptured resistor effectively resets its current to zero, provided the magnitude of the voltage source is opposite to that of the voltage at the time of the rupture. Since the total voltage across the resistor is now zero, no current flows.

The model of a ruptured node can be seen in figure 4.

Using this model, as the external voltage continues to increase, a voltage may develop across the resistor once again; this allows the resistor to continue to pass current. This current can then increase again up to its critical level, where it ruptures again, ad infinitum.

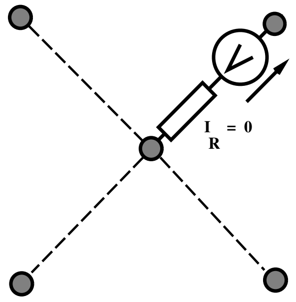


Figure 4: Model of Ruptured Node

Kirchoff's current law states that the sum of currents into a node is always zero (i.e. charge is conserved). This implies that a ruptured node will affect the distribution of current throughout the lattice structure. This is analogous to the modelling of long-range elastic interactions in the geological model, as opposed to a ruptured node only affecting neighbouring nodes.

Thus the electrical model works as follows:

- Initially, each resistor is assigned a critical current.
- An external voltage applied to the lattice is increased linearly with time.
- When the current in a resistor exceeds its critical current, the node is "ruptured" by inserting a voltage source.

After each rupture, the current distribution in the lattice is immediately re-calculated.

2.3 Mathematical Description of Model

Each node in the lattice has a voltage associated with it, therefore the resistors between the lattice nodes have a potential difference across them. Using Ohm's Law, the current through the resistor can be calculated.

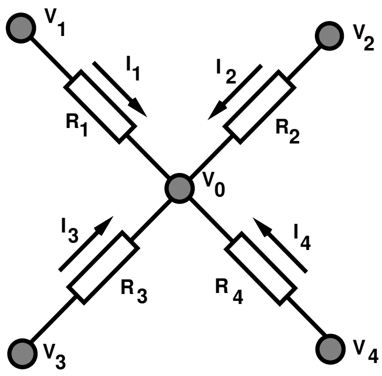


Figure 5: Example Node

Referring to figure 5, the current at any node, n, can be described by $i_n = \dfrac{v_n - v_0}{R_n}$

Kirchoff's Current Law may then be applied to each node in turn, summing the current into each node to zero.

Therefore, using $\sum_{n=1}^{4} i_n$ and $\sigma_n = \dfrac{1}{R_n}$ where σ is the conductance between 2 nodes, we find that for node 0 in figure 5:

$$\sigma_1(v_1 - v_0) + \sigma_2(v_2 - v_0) + \sigma_3(v_3 - v_0) + \sigma_4(v_4 - v_0) = 0$$

For this node, the right hand side of the equation is zero; this is not necessarily the case however. Nodes at the boundary conditions of the matrix are tied to particular voltages, leading to constants appearing in the right hand side of the equation. Similarly, the source voltages which are added between nodes at rupture can also cause voltage terms to appear on the right hand side of the equation.

By extending the previous calculation to each of the n nodes, a system of n linearly independent equations with n unknowns is formed. Consequently, the voltages and currents in the system can be expressed as a series of linear simultaneous equations of the form $Ax = B$, so that it is possible to find the current in the lattice at each node, given the node voltages and the resistances/conductances between the nodes.

A system of linear equations of the form $Ax = B$ looks like:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots &= b_2 \quad \text{,etc.} \end{aligned}$$

A useful means of representing these systems of simultaneous equations is using matrix algebra. The equations above can be given in matrix form as shown below:

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & \ddots & & \\ \vdots & & \ddots & \\ a_{n1} & & & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Computationally, the problem essentially involves solving the series of equations above in terms of x. Thus the computational core of the problem will be some sort of linear system solver. This will allow the current distribution to be re-calculated instantaneously at each rupture. This in turn models the long-range elastic interactions in the geological model.

3 Discussion of Serial Code

The department of Geology and Geophysics at Edinburgh University already utilises some code to perform this network modelling. The code is a large piece of serial Fortran 77.

The function of the code is to perform the rupture analysis described above, by using the electrical model.

The main portion of the code performs the following analysis:

- Increment supply voltage
- Calculate Current Distribution of Matrix
- Check for Ruptured Resistors
  - Calculate new currents with Conjugate Gradient
- Increase time-step
- Repeat as required

The code dumps all information out to data files, which can be visualised at a later date to give output as in figure 2 (b).

4 Profile of Serial Code

The existing serial code supplied by the Department of Geology and Geophysics at Edinburgh University was recompiled with profiling flags, to allow run-time profiling of the code.

The motivation for this profiling was to identify the sections of code which were executed most often. This would highlight which areas of code should then be considered for parallelisation.

Profiling was carried out on a selection of problem sizes. The size of the lattice was varied as was the convergence tolerance and the number of ruptures desired.

The Gnu Profiler, `gprof` was used to perform code profiling — example output can be seen in table 1. The general trend exhibited by code profiling was that approximately 98% of execution time of the rupture analysis code was spent in the subroutine `conjgrad`. This is highlighted in figure 6. `conjgrad` is the subroutine responsible for solving the linear system of simultaneous equations: the conjugate gradient algorithm. This algorithm will be discussed further in Section 5.

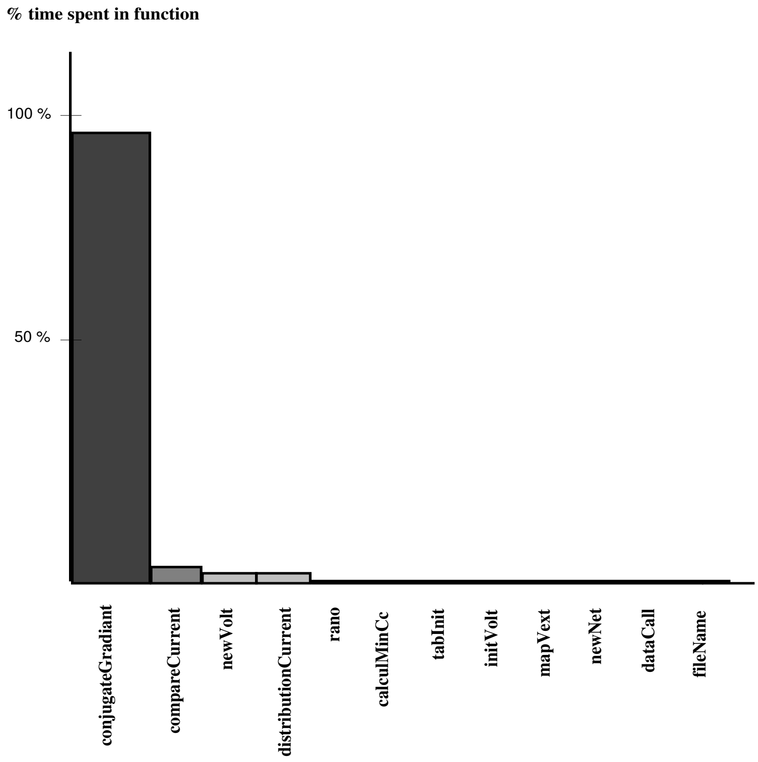


Figure 6: Graph of Code Profile

It is not surprising that so much of the rupture analysis is spent in the conjugate gradient algorithm. At each timestep, whenever a rupture occurs, the distribution of current in the matrix is altered. This re-distribution of current consequently affects the voltage at each node in the matrix. These new voltages are determined by solving a linear system with a call to the conjugate gradient algorithm.

5 Scope for Parallelisation

5.1 Discussion of Conjugate Gradient Algorithm

The method of conjugate gradients may be used for solving a system of simultaneous linear algebraic equations,

$$Ax = B$$

where A is an $N \times N$ matrix of coefficients, x is an $N \times l$ vector of unknowns and B is an $N \times l$ vector of constants. The method will provide a solution where it exists. The method is an N-step iterative one; i.e. an algorithm is applied to give succesive approximations to the solution of the given linear system and, if computations are done with complete accuracy, a solution is obtained (where it exists) after M iterations, where $M \leq N$ (the order of the system).

The principal advantages of using the conjugate gradient algorithm are:

- The simplicity of the computational procedure
- The preservation of the original matrix of coefficients during the computation and the consequent ability to exploit the advantage of having a sparse matrix
- The need for a small amount of storage space to hold intermediate results
- The superiority of each approximation to the preceding ones in the sense of being closer to the true solution

A discussion of the derivation of the conjugate gradient algorithm, along with proofs of its correctness is omitted from this report. Interested readers are referred to [Ralston and Wilf, 1960].

Figure 7 shows the flowchart for the algorithm; the eventual Fortran implementation of the conjugate gradient algorithm closely follows this flowchart in implementation.

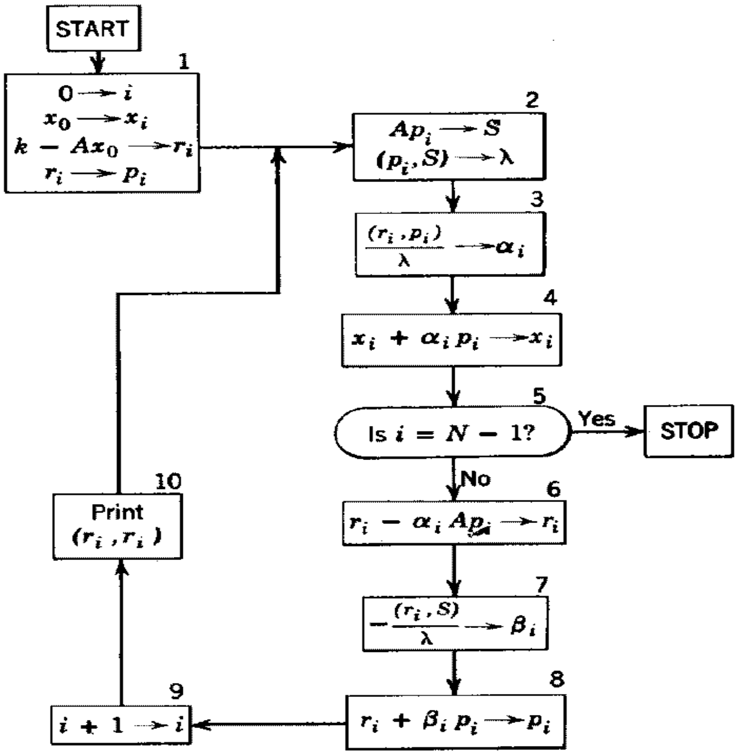


Figure 7: Flowchart of Conjugate Gradient Algorithm

5.2 Approach to Parallelism

Observation of the algorithm flowchart in figure 7 shows that there are 3 "computationally intensive" subroutines which occur:

- Matrix by Vector Multiplication
- Inner Product of 2 vectors
- Sum and Difference of 2 vectors

Parallelisation of the conjugate gradient algorithm will be achieved by parallel implementation of the 3 subroutines above. These will be discussed in turn:

5.2.1 Matrix by Vector Multiplication

The lattice structure obtained from both the geological and electrical model is given in node form in figure 8:

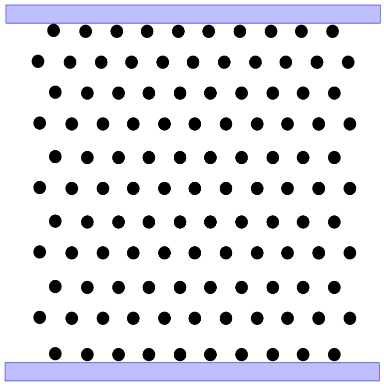


Figure 8: Node representation of Matrix

In performing a matrix by vector multiplication, this matrix is shared amongst the available processors. Each processor then performs calculation on its share of the matrix, thus diminishing the size of the task. Due to the nature of matrix multiplication, the subdividing of the matrix over several processors is realised as in figure 9. Each processor is allocated consecutive rows of the matrix to operate on.

The calculation to be performed at each nodes was given in Section 2. This calculation is a consequence of Ohm's Law and Kirchoff's Current Law. In order to sum the currents into a particular node, the currents in the 4 surrounding resistors are required. These currents are calculated from the voltage across the resistor. This voltage is determined by the voltage at the nodes either end of the resistor.

As a result, the calculation performed at any particular node is a 5-point calculation — it requires the voltages at the node being calculated *and* the surrounding 4 nodes.

Consider a node which lies on the first or last row of a processor's share of the matrix. Since the calculation involves surrounding nodes and these surrounding nodes are resident on *another* processor, the surrounding nodes will require to be communicated between processors. These nodes which require ancillary nodes resident on neighbouring processors are said to fall in the processor's halo region. The halo regions for a matrix divided over 3 processors are shown in figure 10.

Consequently, each processor requires the voltages at all nodes in its 2 neighbours halo regions. This requires that each processor:

- Sends its 2 halo regions to its 2 neighbouring processors
- Receives 2 halo regions from its 2 neighbouring processors

Communication of these halo regions requires a finite time. It would be useful to mask this communication time by performing useful computation. The approach taken is that each processor starts calculating for nodes where *all* the surrounding nodes are local (on the same processor). This computation is termed

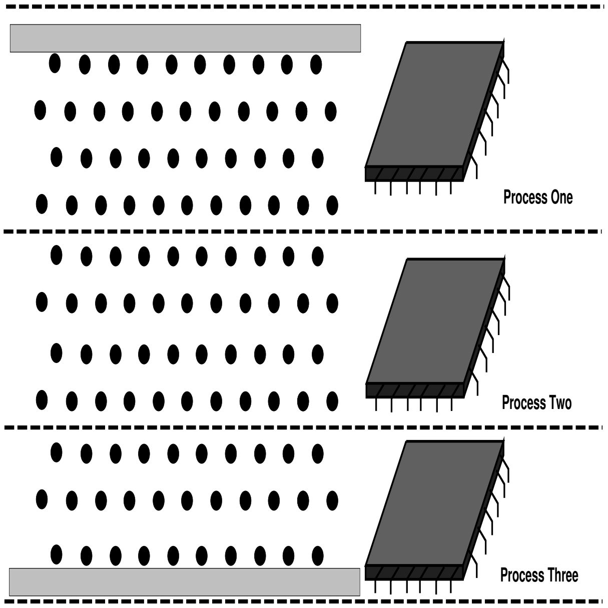


Figure 9: Subdivision of Matrix over Processors

"core computation" (where the core is the portion of the processors data which does not fall into the halo region). The sequence of events is thus:

- Processor sends its halo regions to neighbours
- Processor requests neighbour's halo regions
- Processor proceeds to perform core computation
- Processor receives neighbour's halo regions
- Processor proceeds to perform halo computation

Ideally, during the time the processor is performing core computation, the neighbouring processors have successfully sent the halo regions. The processor will then have the halo data available immediately it has performed the core computation and may continue to perform the halo computation. This *latency hiding* hides as much of the communication overhead as possible behind useful computation.

5.2.2 Inner Product of 2 Vectors

The parallel implementation of inner product is similar to the matrix by vector multiplication, in that the large data set is subdivided over several processors. Each processor then performs smaller calculation over the local data set.

Inner product is an element-wise algorithm; corresponding elements are multiplied together and all the products are then added together. By dividing the data set over several processors, each processor may perform a sum of products on its local data. Performing a global sum of all processors' local calculations then yields the inner product of the original vectors.

This is demonstrated in figure 11 for a data set which is spread over 2 processors.

Unlike the matrix multiplication, there is no useful computation can be performed to hide the communication latency, hence the time to perform the global sum operation will be a rate-determining step.

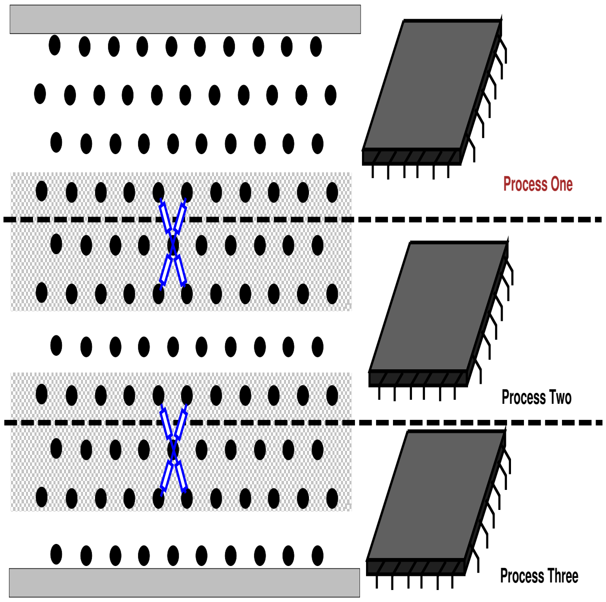


Figure 10: Example of Processor Halo Regions

5.2.3 Vector Addition/subtraction

Vector addition and vector subtraction are very similar to the inner product calculation above. Data can be spread across processors in a similar manner, and each processor may then perform addition or subtraction in an element-wise fashion on its local data.

Unlike the inner product calculation however, there is no requirement for processors to communicate their results to other processors. Consequently, vector addition and subtraction are highly amenable to parallel implementation.

6 Performance Modelling of Conjugate Gradient Algorithm

Before commencing coding of a parallel implementation of the Conjugate Gradient Algorithm, a decision was made to design a performance model of the proposed implementation. The primary reason for this performance model was to investigate if parallelisation of the algorithm would increase performance significantly enough to justify a parallel implementation, as opposed to looking for a new solution technique.

A simulation of the rupture analysis code was run using the following parameters:

- Problem size was a 90 x 90 Grid
- 100% Variation in critical currents was associated with each resistor element
- Random Number Generator was (arbitrarily) seeded at 666
- Boundary Condition Increment was set to 0.01
- Error Criterion was set to 1×10^{-20}
- Number of ruptures set to 20

The analysis code was compiled with certain compiler options, to allow profiling code to be used. The code was then run and profiled using the software profiling package gprof.

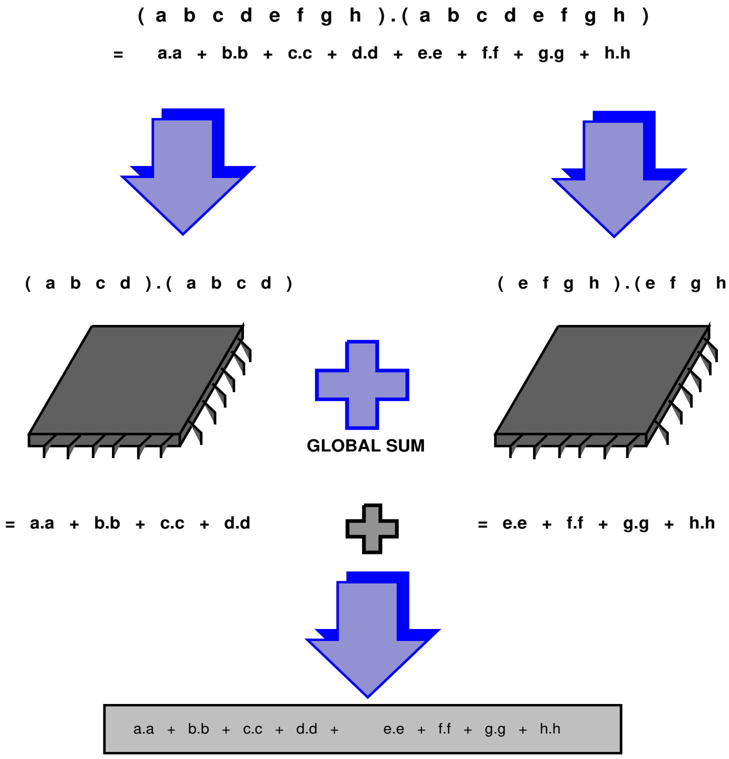


Figure 11: Example Inner Product Calculation

% time	seconds	cumsecs	calls	function
98.51	4629.99	4629.99	80	conjgrad_
:	:	:	:	:
:	:	:	:	:

Table 1: Results of Code Profile

6.1 Profile Results using GPROF

The following results were obtained from the `gprof` profiler. The results indicated the amount of time that was spent in each of the functions in the code. These results are given in table 1 and correspond to the profiling graph shown in figure 6.

The code file which implemented the conjugate gradient algorithm, `conjgrad.f`, was amended to produce a dumpfile. This file indicated the number of iterations of the algorithm required until convergence criterion was met. Using the results of the code profile and the results obtained from the dumpfile, the time required to perform 1 iteration of the conjugate gradient algorithm was determined:

Average Number of Iterations of Conjugate Gradient per Call = 426.8

$\Rightarrow$ Number of Iterations of Conjugate Gradient = 426.8 $\times$ 80 = 34144

$\Rightarrow$ Time for 1 iteration of Conjugate Gradient Algorithm = $\frac{4629.99}{34144}$ = 0.14 seconds

6.2 Performance Model of Parallel Operations

Looking at the flowchart for the conjugate gradient algorithm in figure 7, it can be seen that the conjugate gradient algorithm requires 3 calculations of an inner product (in steps 2, 3 and 7) and 1 multiplication of a matrix with a vector (in step 2).

The proposed parallel implementation of inner product will require a global sum operation to be performed, while the proposed parallel implementation of matrix/vector multiplication will require processes exchanging one-dimensional arrays of size p, where p is the number of nodes in the top row of the lattice structure.

A suite of benchmarking programs were designed which accurately measured the time required to perform the array-swapping and the global sum operation. These programs are briefly described in the following sections:

6.2.1 Array-swapping Benchmark

The proposed implementation of a parallel matrix multiplication algorithm is discussed in section 7. It is important to note that each processor in the parallel algorithm will require to perform 2 send operations and 2 receive operations — these are for sending and receiving halo regions from the neighbouring processors. The performance model therefore tries to capture the temporal overhead which will be involved in a processor sending and receiving a fixed length array (the halo region) from both its neighbouring processors. Figure 12 demonstrates the array swapping that is involved:

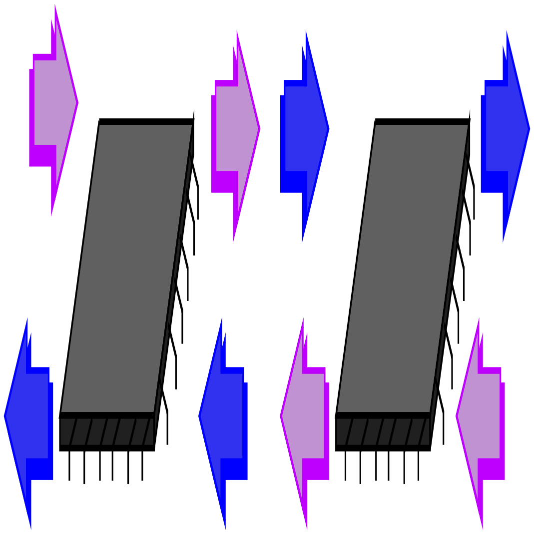


Figure 12: Array Swapping Benchmark

The benchmark code performed 1000 iterations and calculated the time required for a single iteration, each iteration performing 2 sends and 2 receives.

From this code, an estimate could be made for the time critical part of the matrix multiply algorithm — that of the swapping of the halo regions.

6.2.2 Global Sum Benchmark

Calculation of an inner product is a simple enough process — each processor takes a portion of the 2 vectors and performs a smaller, local sum. A global sum of each processor's local value will then give the inner product of the 2 initial vectors. The performance critical code in the inner product algorithm is therefore the time to perform a global sum. The benchmark code therefore performed 1000 global sum operations between a number of processors and then calculated the time to perform a single global sum (as required by the inner product algorithm).

6.2.3 Benchmark Results

Performance modelling of array-swapping and global sum was then performed on a Sun 4 cluster (the proposed hardware platform) and it was found that the times for a typical sized problem (where p ≈100 nodes, with a network cluster of between 4 and 10 machines) were:

- Global Sum Operation: ≈50 ms
- Array Swap Operation: ≈40 ms

This performance analysis was carried out on a "live" Ethernet network, therefore message contention will be included in the results above.

6.3 Analysis of Time for Serial Code to Run in Parallel

6.3.1 Inherently Serial Code

Serial computation performed on N processors speeds up by a factor of N when compared with the time to execute on a single processor. This is an obvious result of "division of labour" as each processor is responsible for performing an equal fraction of the work involved.

6.3.2 Inner Product (Global Sum)

The global sum operation will introduce an extra latency, as each processor is required to send its share of the inner product calculation to all the other processors in order that the final result may be calculated.

Thus an individual global sum introduces an extra time, T_g. T_g will be a function of the number of processors, with the function being dependent on the underlying software and the network topology.

Hence

$$T_p = \frac{T_s}{N} + T_g(N)$$

T_p - Time for parallel execution
T_s - Time for serial execution
N - Number of processors
T_g - Latency introduced by Global Sum

6.3.3 Matrix/Vector Multiply (Array Swap)

Swapping an array between 2 processes requires a time T_{swap}. During the time T_{swap}, useful computation may be performed (e.g. multiplying matrix and vector elements). Thus T_{swap} should be weighted with a function C(N) (where $0 < C(N) < 1$) which will reduce T_{swap} according to the amount of time that can be usefully used to perform some computation. Thus the product $C(N).T_{swap}$ will be a real measure of the overhead involved in array swapping during the matrix multiplication.

Hence,

$$T_p = T_p + C(N).T_{swap}$$

6.4 Parallel Implementation of Conjugate Gradient

Using the model parameters discussed above, and noting that conjugate gradient will involve 3 global sum operations and 1 matrix/vector multiply, a model for the parallel implementation may be described by:

$$\boxed{T_p = \frac{T_s(p)}{N} + 3.T_g(N) + C(N).T_{swap}}$$

Where:

T_p	- Parallel Execution Time
$T_s(p)$	- Time to execute in serial
$T_g(N)$	- Time to perform global sum
T_{swap}	- Time to perform array swap
C(N)	- Weighting function for useful computation
N	- Number of Processors
p	- Size of matrix in problem

6.5 Limitations of the Model

It has been noted that for a workstation network, T_g is a function of the number of processors (since more processors introduces a larger communication latency).

C(N).T_{swap} could also be more accurately described as C(N,p).T_{swap}, since the larger p is, the more computation can be performed and the smaller the effect T_{swap} will have on slowing the code.

Note that the functions which describe T_g and T_{swap} were hard to obtain experimentally, as network loading had a significant effect on execution times. This network loading causes many anomalous results which mask trends over a small result set. It is clear from runs of the benchmark programs however that the values are not hugely dependent on N within a small range of processors (4-10).

6.6 Performance Figures

The main conclusion arising from the performance modelling was that a parallel implementation of the conjugate gradient algorithm is limited largely by the execution time of the global sum operation.

For targeted workstation cluster, execution time for global sum is in the order of 45-50 ms. The conjugate gradient algorithm was measured previously to run in about 0.14 s.

Performance will deteriorate when the global sum dominates all computation, hence number of processors that can usefully be used is given by:

$$NumberofProcessors = \frac{Timeforexecutionofconjugategradient}{Timeforglobalsumoperation}$$

Using a Sun4 cluster, the optimum number of processors comes out at around 1 or 2 processors. The problem would therefore appear to be not very scalable to a large number of processors: a parallel implementation would possibly be slower than its serial counterpart, due to the inherent communication involved.

However, the Cray T3D exhibits much tighter coupling between processors, meaning that the global sum operation will not be as expensive to perform. The conjugate gradient algorithm should therefore be suitable as a parallel linear solver, if implemented on a machine with fast intra-processor communication.

7 Implementation of Parallel Rupture Analysis Code

Originally, the target port for the parallel code was to a workstation cluster. Such a port would be DM-MIMD (Distributed Memory: Multiple Instructions, Multiple Data). Each workstation could have its own data set and indeed its own code. Thus the rupture analysis code could run on one machine, with all the other machines executing the linear solver (conjugate gradient) algorithm.

However the decision to port to the Cray T3D meant that the code had to be written as SPMD code (Single program, multiple data). This is a consequence of the Cray T3D being an SPMD machine. Note however that SPMD code may still be run across a workstation cluster.

The SPMD ethos of programming requires that each processor executes the same piece of code. Different processors may operate however on their own unique data sets. Thus the design of the parallel rupture analysis code is around a conceptual master/slave relationship between processors.

A single master processor performs all the book-keeping, initialisation and update of files and the general rupture analysis algorithm (increasing the voltage at each timestep, checking for ruptures, etc). However, when a rupture occurs, the master orders the slave processes (running on all the other processors) to execute the conjugate gradient algorithm.

In order that each slave may perform the conjugate gradient algorithm, the master must send each slave:

- Its share of the global data set
- Information regarding the size of the data set, convergence criterion, etc

The slaves may then independently perform the conjugate gradient algorithm on their own portion of the data set. On completion of the algorithm, each slave returns its solution to the master process, which re-assembles the slave solutions to obtain a complete solution to the linear system.

Thus the only truly parallel code is that in performing the conjugate gradient algorithm. Since around 98% of program execution is spent in this algorithm however, this should constitute significant performance increase.

8 Performance Gains

Since the code had not yet been run on the Cray T3D at the time of this report, it is impossible to quantify what the advantages are, except in terms of the performance model. These advantages are:

- For a machine with fast inter-processor communication, the code will scale well (i.e. speed up almost linearly with the number of processors up to a large number)
- If the calculation is eventually fully distributed (i.e. no gathering and scattering of data between master and slaves) then there will be a memory scalability pay-off.
- The halo swapping can be implemented with Cray shared memory routines. This will only require changes inside the halo-swap routines, and would look like message passing at the higher level (ie the same control structures can be used)
- Apart from a trivial amount in the global sum calculation, no computation has been added - the cost is all communications. This means that the speed up resulting from parallelism all goes into the original calculation, and not into extra overheads (i.e. the algorithm is the same). Thus performance can be measured by comparing the parallel code against the serial code and *not* against the parallel code running on one processor.

9 Future Development

The direction of this project led to a parallel implementation of the conjugate gradient algorithm design for an SPMD system; specifically the massively-parallel Cray T3D.

However a significant conclusion of this project was that this implementation would be largely inefficient over a workstation cluster. Since access to machines such as the Cray T3D is limited, it would be useful to implement a solution technique for linear systems which could be run across a workstation cluster.

During the design and evaluation phase of this project, much interest was directed towards the Jacobian Method for solving linear systems. This method deals well with sparse matrices (such as those obtained in the rupture analysis outlined here) and is particularly amenable to parallelisation. The Jacobian Method should not be difficult to implement within the current code framework. Success of this algorithm is largely dependent however on the method's convergence qualities.

The emphasis in this project was in parallelisation of the *linear solver* subroutine. This is the obvious "first choice" for parallelisation, since 98% of code execution was within this algorithm. There is opportunity however to parallelise more of the rupture analysis code; comparing current distributions, checking for ruptures, etc. Indeed large problems could be sub-divided into smaller analyses carried out over different groups of processors.

By fully distributing the calculation, gathering and scattering of data between the master and the slaves could be completely eliminated. This would have little effect on performance, but would mean that the memory requirements of the master process would be significantly reduced, allowing larger overall problems to be tackled.

A further possibility, and one which should be straightforward to implement, is to utilise the master during the conjugate gradient calculation. The current program has the master process idle while the slaves perform conjugate gradient analysis; by giving the master a share of the problem, the benefits of utilising an extra processor are gained with little or no performance overhead.

10 Conclusions

This project has undertaken to model performance of a particular parallel implementation of the Conjugate Gradient algorithm for solving a linear system and then implement that model.

The conclusions reached by the performance model were that the proposed implementation would be limited by slow communications over a workstation cluster. The decision was then made to target a parallel port towards the Cray T3D machine, which exhibits tighter coupling between processors and hence a reduced communications latency.

This project was concerned with implementing the port; at the date of completion of the project a full implementation of the required message-passing library (MPI) had not been ported to the Cray T3D, therefore no performance benefits could be measured by running the parallel code on the Cray T3D.

This project did however demonstrate the efficiency with which the MPI message passing system can be used to realise the movement of data between processes and the management of inter-process communication. This significantly raised the level of abstraction from the machine level.

A further conclusion of this project was that *iterative* methods of solving linear systems are more efficiently implemented on a parallel architecture than non-iterative methods such as Gaussian Elimination and LU-Decomposition methods.

For further reference on the methods of linear solving discussed in this report (conjugate gradient and Jacobian), the reader is pointed towards [Hildebrand, 1974; Isaacson and Keller, 1966; Johnson and Riess, 1982; Laboratory, 1970].

11 Acknowledgements

Thanks to Colin Brough (EPCC) for guidance throughout the project.

Thanks to Patience Cowie for explaining the geology behind hundreds of simultaneous equations.

Special thanks to Mark Sawyer for continuous and insightful assistance throughout the project.

A Appendix A - Supplied Subroutines

This appendix lists a brief description of the code functions supplied by the author during this project.

A.1 conjgrad.F

conjgrad(i_slaveID, i_comm) This is the parallel conjugate gradient function.

multiplyAp(ap, pi, in, i_numNodes) This function performs the multiplication $Ap_i \rightarrow S$ in Step 2 of the conjugate gradient algorithm (see figure 7).

A.2 dataSend.F

slave_reserveBuffers(p) Called by slave process to calculate how much buffer space it will require to store its share of the matrix and also to calculate which part of the matrix it will be storing.

spreadData(v, sv, p) Master process sends the voltages at each node and the source voltages added after rupture events to all the slave processes.

gatherData(v, p) Master process receives each slave's newly computed portion of the matrix.

A.3 innerProduct.F

innerProduct(dp_vector1, dp_vector2, i_vectorLength,dp_result, i_comm) Parallel implementation of the inner product calculation used by the conjugate gradient algorithm.

A.4 masterSlave.F

slaves_start_conjgrad Called by master process to inform slaves when they should call the conjugate gradient algorithm.

inform_slaves_if_finished(finished) Called by master process to inform slave whether the rupture analysis is complete or not.

wait_for_slaves_to_finish Holds up the master process during conjugate gradient analysis until slaves notify that they are finished.

wait_until_told_todo_conjgrad Holds up the slave processes until they are required to perform conjugate gradient algorithm.

inform_master_that_finished Called by slaves when they have completed conjugate gradient algorithm.

finished_rupture(done_rupture_analysis) Called by slave processes to check if they will require to perform further conjugate gradient analysis.

A.5 MPI_init.F

MPI_Start(i_errorCode, i_processRank) Initialises the MPI message passing interface.

MPI_GroupSlaves(i_comm, i_errorCode) Groups all the slave processes within their own MPI communicator. This is required for the inner product calculation (since slaves all need to communicate with each other, but not the master).

MPI_Stop(i_errorCode) Shuts down the MPI message passing system. Must be called by *all* processes.

A.6 parMulAP.F

multiplyAp(ap, pi, in, i_numNodes) Parallel implementation of the multiplyAP routine in `conjgrad.F`. Operates over a portion of the grid rather than the entire grid. Not fully implemented by author.

haloSwap_init Initiates halo swapping. Process sends its halos to neighbours and initiates receipt of halos from neighbours.

haloSwap_completeRx(commHandle) Process busy waits until it receives halos from neighbours.

haloSwap_completeTx(commHandle) Process busy waits until its neighbours have received its halo regions.

References

[Hildebrand, 1974] F B Hildebrand. *Introduction to Numerical Analysis*. McGraw Hill, second edition, 1974.

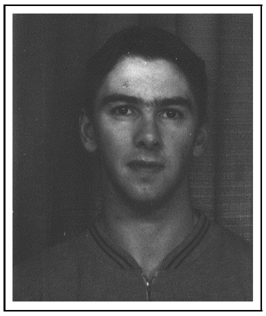
[Isaacson and Keller, 1966] E Isaacson and H B Keller. *Analysis of Numerical Methods*. John Wiley and Sons, 1966.

[Johnson and Riess, 1982] Lee W Johnson and R Dean Riess. *Numerical Analysis*. Addison-Wesley, second edition, 1982.

[Laboratory, 1970] National Physical Laboratory. *Modern Computing Methods*. Notes on Applied Science No. 16. Her Majesty's Stationery Office, 1970.

[Patience Cowie and Sornette, 1993] Christian Vanneste Patience Cowie and Didier Sornette. Statistical physics model for the spatiotemporal evolution of faults. *Journal of Geophysical Research*, 98(B12):21809–21821, December 1993.

[Ralston and Wilf, 1960] A Ralston and H.S Wilf. *Mathematical Methods for Digital Computers*, volume 1. John Wiley and Sons, 1960.



Steven Webster was born in Kirkcaldy, Scotland in 1973. He is currently in his final year at the University of Edinburgh, studying for a joint honours degree in Computer Science and Electronic Engineering.

On graduation, Steven intends to follow a career in Software Engineering, hopefully allowing him to continue in both Computer Science and Electronics.

This Project was supervised by Mark Sawyer (Edinburgh Parallel Computing Centre) and Patience Cowie (Department of Geology and Geophysics, University of Edinburgh).

Contact:

Mark Sawyer
Edinburgh Parallel Computing Centre
James Clerk Maxwell Building
King's Buildings
University of Edinburgh
West Mains Road
Edinburgh

E-Mail: cms@epcc.ed.ac.uk

Patience A Cowie
Department of Geology and Geophysics
King's Buildings
University of Edinburgh
West Mains Road
Edinburgh

E-Mail: cowie@mail.glg.ed.ac.uk