Geochemistry Geophysics Geosystems G3
AN ELECTRONIC JOURNAL OF THE EARTH SCIENCES
Published by AGU and the Geochemical Society

Article
Volume 7, Number 11
1 November 2006
Q11001, doi:10.1029/2006GC001246
ISSN: 1525-2027

Efficient parallel inversion using the Neighbourhood Algorithm

P. Rickwood and M. Sambridge
Centre for Advanced Data Inference, Research School of Earth Sciences, Australian National University, Canberra, ACT 0200, Australia (peter.rickwood@gmail.com; malcolm.sambridge@anu.edu.au)

[1] Issues controlling efficient parallel implementations of a popular direct search inversion algorithm are analyzed and discussed. A naive parallelization of a particular method, the Neighbourhood parameter search algorithm, leads to inefficient use of parallel architecture through lack of scalability and intolerance to hardware faults. These factors are quantified, and their origins are explained. A reformulation of the algorithm leads to dramatically improved performance when the cost of the forward problem is low and the number of unknowns is high. Numerical examples are used to illustrate the main results. Factors in the original Neighbourhood Algorithm which lead to poor parallel performance are likely to be present in other ensemble-based inversion or global optimization algorithms. Hence the algorithmic solutions proposed may have widespread application.

Components: 7572 words, 11 figures.

Keywords: inversion; parallel; Neighbourhood Algorithm; distributed.

Index Terms: 0500 Computational Geophysics (3200, 3252, 7833); 3260 Mathematical Geophysics: Inverse theory.

Received 16 January 2006; **Revised** 9 June 2006; **Accepted** 10 July 2006; **Published** 1 November 2006.

Rickwood, P., and M. Sambridge (2006), Efficient parallel inversion using the Neighbourhood Algorithm, *Geochem. Geophys. Geosyst.*, *7*, Q11001, doi:10.1029/2006GC001246.

1. Introduction

[2] Inverse problems arise when some quantity of interest (in geophysics, usually a property of the Earth at depth), is only indirectly constrained by observables. In geophysics, inverse problems have been studied for many years (for a review, see *Sambridge and Mosegaard* [2002]). For every inverse problem there is a corresponding forward problem, where predictions are made using a particular set of values for the unknowns and these are compared to the data. If the mathematical relationship between observables and unknowns is nonlinear, then the inverse problem can become difficult to solve. In the 1970s and 1980s nonlinear inverse problems (like seismic waveform inversion) often required use of restrictive linearizing approximations to make them tractable. More recently, with increased availability of high performance computing, fully nonlinear treatments have become popular. In particular, global optimization and related direct search Monte Carlo techniques have found many applications [*Sambridge and Mosegaard*, 2002; *Mosegaard and Sambridge*, 2002]. Most of these methods are "ensemble"-based, which means that they involve the solution of the forward problem for many candidate solutions simultaneously. Such approaches naturally lend themselves to parallel computation. However, robust and efficient use of parallel computation is a far from trivial task.

[3] There are two trends in computer technology that bring us inevitably to the conclusion that the future of scientific computation is in parallel computing. These are the availability of low-cost, high-performance commodity processors and the increasing availability of high-bandwidth, low-latency networking components. While there will always be a need for traditional supercomputers (such as multi-vector-processor machines), such computers now constitute a small (and falling) percentage of the world's most powerful computers (see, for example, http://www.top500.org). The scope for exponential single processor performance increases is limited, as trends in computer architecture such as limited memory bandwidth [*McKee*, 1995] and heat dissipation are already seeing an end of computer engineers' ability to translate *Moore*'s [1965] prediction of exponentially increasing transistor density into single-chip processing speed. On the other hand, for tasks capable of being distributed over a cluster computer consisting of networked commodity processors, the computing resources available to carry out such tasks will continue to grow exponentially for some time. On current trends, the world will have a supercomputer capable of a petaFLOP within the next few years. Already, the top entry (IBM's Blue Gene) is at 1/4 of a petaFLOP, about 8 times more processing throughput than NEC's Earth Simulator.

[4] The mere availability of computational resources is, however, not enough. In order to harness these resources, care needs to be directed to devising efficient and scalable parallel implementations of the algorithms we wish to execute on these parallel computers. We believe that, generally speaking, the software produced and used by the scientific community still lags in its ability take advantage of such processing power. We consider a program scalable only if it satisfies three conditions: it can be efficiently subdivided to run on a cluster; it can survive hardware failure; it performs efficiently despite processor performance variation. Implementations of the MPI standard [*Message Passing Interface Forum*, 1994; *Snir et al.*, 1998], upon which much distributed scientific software is based, are fault-intolerant (the failure of a single process or node in a distributed program relying on MPI results in the failure of the entire program). This is an indication of how little attention has been devoted to scalable scientific software.

[5] Let us consider the two aspects of scalability separately:

[6] 1. Fault-tolerance: Consider that IBM's Blue Gene, the top entry on the current top500 list (see http://www.top500.org), has over 100,000 processors, and, assuming a mean time to failure of approximately 3 years per processor, the expected time to failure of the complete system is measured in minutes. One way of dealing with such failure rates is to design software that explicitly deals with hardware and software failure. Relatively little effort has been devoted to developing scientific software that does this, and indeed, as already mentioned, the underlying message passing libraries used by many scientists (such as MPI) are themselves fault-intolerant.

[7] 2. Coping with heterogeneity: In a heterogeneous computing environment, software needs to accommodate variation. On computational grids, for example, the machines taking part in the computation will typically have varying latency, bandwidth, load, and processor speeds. Even on a cluster of identical commodity processors, multilevel caches, address translation hardware, and network topology all introduce natural variation in performance. In either case, algorithms that assume homogeneity and do not attempt to accommodate variation will perform poorly on at least some classes of problems.

[8] In this paper we focus on a particular ensemble-based nonlinear (inversion) search algorithm in use in the geoscience community: the Neighbourhood Algorithm [*Sambridge*, 1999a]. We show that its current parallel implementation scales poorly and is fault-intolerant. We then go on to show that a reformulation allows for a fault-tolerant implementation that scales so much better that, even on a moderately sized cluster of 120 nodes, speedups of more than an order of magnitude can be achieved on published computations.

[9] With our reformulation of the Neighbourhood Algorithm, we are not concerned with marginal improvements in the speed of the algorithm. Our focus has been on creating an implementation that not only is more efficient today, but is also more scalable, and hence will become increasingly more efficient (relative to the "old" implementation of the Neighbourhood Algorithm) as current computing trends continue. This, combined with the fact that our new implementation can already result in an order of magnitude reduction in running time on a moderately sized cluster today means that the range of problems that will be computationally feasible in the future will be significantly expanded.

[10] Although we focus here specifically on the Neighbourhood Algorithm, it is worthwhile noting that our criticisms of its present parallel implementation (detailed below) are likely to be valid for other ensemble-based inversion algorithms. Genetic Algorithms and ensemble-based Simulated Annealing, for example, are commonly used in the geoscience community [e.g., *Sen and Stoffa*, 1991, 1992, 1995] and naive parallel implementations of these algorithms will suffer from many of the same deficiencies as the Neighbourhood Algorithm. Hence the mechanisms described here for efficient parallelization and fault tolerance may also be applicable to implementations of other parameter search algorithms.

2. Neighbourhood Algorithm

[11] The Neighbourhood Algorithm [*Sambridge*, 1999a, 1999b] is a derivate-free, ensemble-based search algorithm, typically used for nonlinear geophysical inverse problems and global optimization [see, e.g., *Sambridge*, 1998; *Kennett et al.*, 2000; *Mosegaard and Sambridge*, 2002; *Pritchard and Simons*, 2002; *Lohman et al.*, 2002; *Sherrington et al.*, 2004; *Agostinetti et al.*, 2004; *Subbey et al.*, 2004a]. A parallel version of the Neighbourhood Algorithm has been available since 2003, but, as we shall explain, can perform inefficiently on certain classes of problems. Here, we describe a reformulation of the Neighbourhood Algorithm that is equivalent (in a probabilistic sense) to the existing formulation, but is fault-tolerant, and lends itself to an implementation that is markedly more efficient.

[12] The Neighbourhood Algorithm consists of two parts: a search stage [*Sambridge*, 1999a] and an appraisal stage [*Sambridge*, 1999b]. These parts can be used independently, and we deal here only with the search stage of the algorithm. This stage consists of a direct search method in a multidimensional parameter space. The objective is to find points (models) with acceptable (high or low) values of a user supplied objective function. In this respect it is the same as other global optimization/inversion methods such as genetic algorithms [*Holland*, 1992a, 1992b; *Goldberg*, 1989] and simulated annealing [*Kirkpatrick et al.*, 1983]. The Neighbourhood Algorithm, however, differs from these methods in that it makes use of geometrical constructs known as Voronoi cells to guide its search (for details on computational geometry and Voronoi cells, see *Okabe et al.* [1992]). Starting with an initial set *S* of points, the Neighbourhood Algorithm samples the neighborhoods of the best of the points in *S*, where the neighborhood of a point is defined as the Voronoi cell that contains it. The objective data misfit function is then evaluated for each of these points, and each point becomes a member of *S*, forming the center of a new Voronoi cell. Figure 1 illustrates the progression of the Algorithm in a 2-D search space.

[13] The current formulation of the Neighbourhood Algorithm [*Sambridge*, 1999a] is iteration based and requires two user-defined parameters, n_r and n_s, to guide the search. In each iteration, n_s samples are randomly generated from within the best n_r Voronoi cells by performing a uniform random walk which relaxes to a spatially uniform distribution within each cell. For example, if $n_s = 80$ and $n_r = 8$, then in each iteration, 10 models (n_s/n_r) are randomly generated within each of the best 8 (n_r) cells by performing a random walk within that cell.

[14] As it is an ensemble inference method (like Genetic Algorithms), the Neighbourhood Algorithm is straightforward to parallelize (as we show in section 4). A parallel implementation has been available and in use for some time [*Subbey et al.*, 2004a, 2004b].

[15] The current parallel implementation of the Neighbourhood Algorithm is exactly equivalent to running the serial version on a faster computer. That is, the exact semantics of the computation are preserved. By relaxing this requirement slightly and requiring only that the essential semantics are preserved, we will eliminate two sources of inefficiency: limited parallelism and barrier synchronization overhead. In this paper, we first explain some key concepts of parallel computation, and define the metrics we will use to compare different parallel implementations. We then describe the essential characteristics of the “standard” parallel implementation of the Neighbourhood Algorithm (what we will call the canonical version), and then proceed to describe the problems with such a straightforward implementation and the steps needed to circumvent these problems. We then present timing results for both our new version and the canonical version on real (i.e., nonsynthetic) published problems [*Braun and Robert*, 2005; *Braun and van der Beek*, 2004; *Yoshizawa and Kennett*, 2002], showing decreases in execution times of between 10% and 96% on a moderately sized cluster of 120 nodes. We also show that the size of the speedup increases with the size of the cluster; in other words, the newer version, as well as being more efficient than the canonical version,

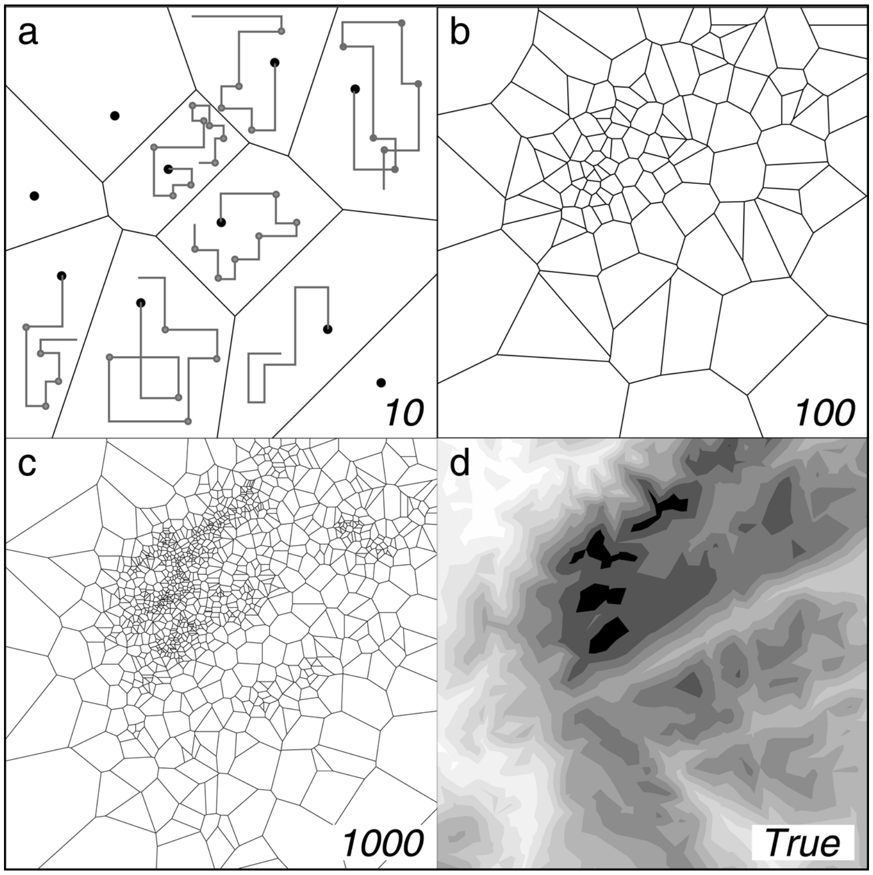


Figure 1. The Neighbourhood Algorithm in 2-D. Starting from some initial set of uniformly distributed models (Figure 1a), the Neighbourhood Algorithm selects models within the better Voronoi cells, and so refines the search space of Voronoi cells (Figures 1b and 1c). Figure 1d shows the contours of the objective function for this 2-D problem.

becomes relatively more efficient as the size of the cluster increases. Given current trends, this final point is critical.

3. Quantifying the Efficiency of Parallel Algorithms

[16] Before commencing our analysis of the existing Neighbourhood Algorithm implementation and our subsequent development of an efficient alternate implementation, we need to be clear on how we measure efficiency.

3.1. Runtime Function ρ

[17] Suppose we have an algorithm A with implementations A_i. We define the runtime function $\rho(A_q, D)_N$ as the time required on a cluster of N computers to execute a particular implementation A_q with input data D. Thus a serial implementation A_s executed on a single machine requires runtime $\rho(A_s, D)_1$. In simple terms, the runtime is just how long a program (called here an implementation) needs to run before finishing.

3.2. Efficiency Measure ϵ

[18] We define the efficiency $\epsilon(A_p, D)_N$ of a parallel implementation A_p on an N-node cluster with input data D as

$$\epsilon\left(A_p, D\right)_N = \frac{\rho\left(A_s, D\right)_1}{\rho\left(A_p, D\right)_N \times N}. \tag{1}$$

[19] We thus measure the efficiency of a parallel implementation in terms of a serial implementation of the same algorithm. In practice, a program that runs in time t in serial can rarely be made to run in less than time t/N on an N-node cluster. Expressed in terms of the runtime function ρ, we expect the following to be true:

$$\rho\left(A_p, D\right)_N \geq \frac{\rho\left(A_s, D\right)_1}{N}. \tag{2}$$

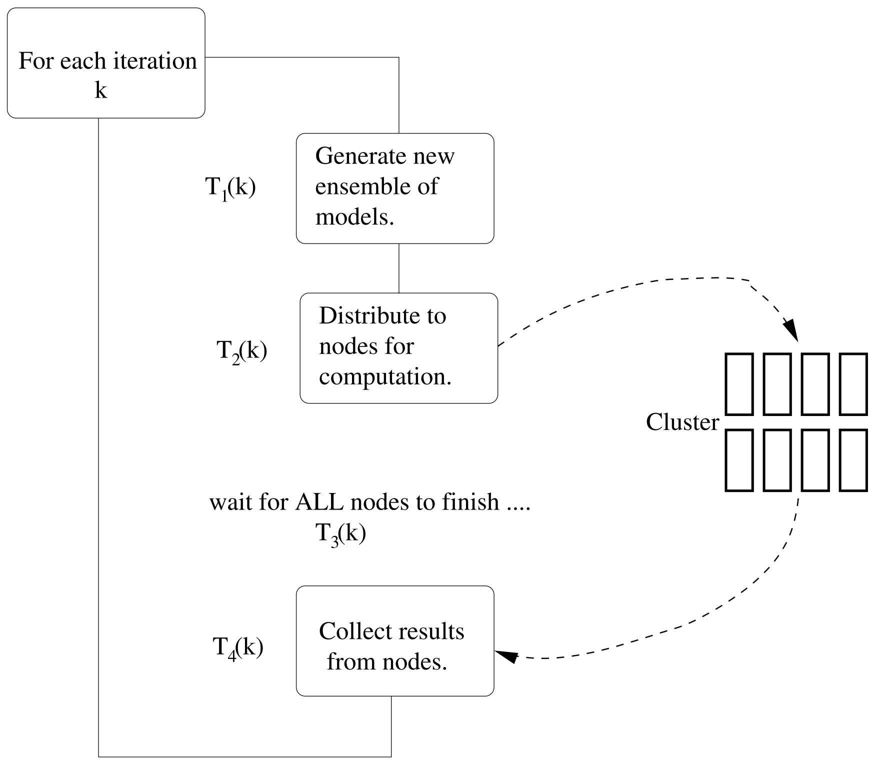

For each iteration k
Generate new ensemble of models.
$T_1(k)$
Distribute to nodes for computation.
$T_2(k)$
Cluster
wait for ALL nodes to finish ....
$T_3(k)$
Collect results from nodes.
$T_4(k)$

Figure 2. A simple method for parallelizing iteration-based ensemble search/optimization algorithms. The forward problem (objective function) is solved independently for each model in the ensemble on a separate node in the cluster.

[20] It is well established that inequality (2) is only false in rare cases, so with little loss of generality, we only consider the common case.

[21] We can see that if (2) is true, then there is an upper bound on ϵ for any parallel implementation A_p:

$$\begin{aligned}\epsilon(A_p, D)_N &\le \frac{\rho(A_s, D)_1}{\rho(A_p, D)_N \times N} \\ &\le \frac{N \times \rho(A_s, D)_1}{\rho(A_s, D)_1 \times N} \\ &\le 1. \end{aligned} \tag{3}$$

[22] The lower bound on ϵ is clearly zero as $\rho(A_p, D)_N \to \infty$, so we have

$$0 < \epsilon(A_p, D)_N \le 1. \tag{4}$$

[23] Thus, in all practical settings, we have lower and upper bounds on ϵ for the Neighbourhood Algorithm.

[24] To summarize in plain English, we assume that a program can, at best, be made to run N times faster on a cluster of N computers. The efficiency function ϵ is a measure of how close we get to this "optimal" increase in speed, with our upper bound ($\epsilon = 1$) indicating we have achieved this optimal increase in speed. Now that we have our defined efficiency metric, we proceed to analyze the existing implementation of the Neighbourhood Algorithm.

4. Naive Approach

[25] A commonly used approach to parallelizing any iteration-based ensemble search algorithm is by the method shown in the flowchart in Figure 2. The currently available implementation of the Neighbourhood Algorithm follows just such an approach, as do many other direct search inversion techniques. The analysis that follows applies to all such algorithms.

[26] From Figure 2 we see that the task of generating and evaluating models has been divided into four subtasks:

[27] 1. Generate on a "master" node a new ensemble of models. We denote the time taken by this subtask in iteration k as $T_1(k)$.

[28] 2. Distribute to cluster nodes for evaluation (taking time $T_2(k)$ in iteration k).

[29] 3. Cluster nodes evaluate received model(s) (taking time $T_3(k)$ in iteration k). Note that it is in

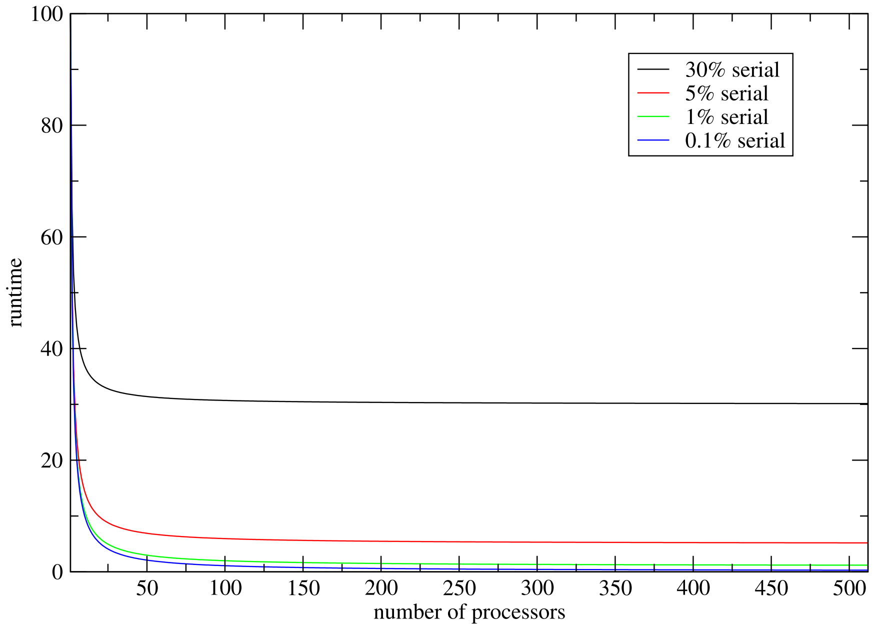


Figure 3. The limiting effect of Amdahl's law. As the size of the cluster computer increases, the speedup of a program is limited by the proportion of code that is not parallelized. The lines are plots of $\rho = 100s + 100(1 - s)/N$ for differing values of s, the percentage of the program that must be performed in serial (i.e., cannot be parallelized).

stage that the forward problem is solved (i.e., the objective function is evaluated).

[30] 4. Collate results from cluster nodes on "master" node (taking time $T_4(k)$ in iteration k).

[31] So $T(k)$ is the total time taken to execute iteration k, and can be expressed as the simple sum of these components:

$$T(k) = T_1(k) + T_2(k) + T_3(k) + T_4(k). \tag{5}$$

[32] Many ensemble-based inversion and optimization techniques can be parallelized through the technique illustrated in Figure 2. However, we contend that such an approach results in parallel algorithms that suffer from two important sources of inefficiency: limited parallelism and barrier synchronization overhead. We address each of these separately.

4.1. Limited Parallelism

[33] In the canonical parallel implementation of the Neighbourhood Algorithm, which follows the design illustrated in Figure 2, it is only the evaluation of the user-supplied objective function (performed in stage 3) that is performed in parallel. The generation of new models, the dissemination of those models, and the collation of results, are all carried out in serial, by the main (master) node.

[34] Amdahl's law [*Amdahl*, 1967] states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. Recast in terms of the speedup achievable through parallelization, the implication of Amdahl's law is that, for an implementation A_i with serial component s and parallel component p, the absolute lower bound (i.e., given an infinite number of processors) on the runtime of A_i is s. In other words, Amdahl's law states that the serial component of a program is the limiting factor in determining how much faster it can be made to run on a cluster. While this may seem somewhat obvious, it is important to remember that when one considers the rapidly increasing size of cluster computers, this lower bound can be the dominant limiting factor in a program's scalability. Figure 3 shows the diminishing returns dictated by Amdahl's law for programs that require varying proportions of nonparallel execution.

[35] Since the lower bound on the execution time of a program, given perfect parallelism, is determined by that part of the program that is performed in serial, we can see that the strict lower bound on the runtime ρ of a program implemented in the manner depicted in Figure 2 is

$$\rho > \sum_{k=1}^{K} T_1(k) + T_2(k) + T_4(k), \tag{6}$$

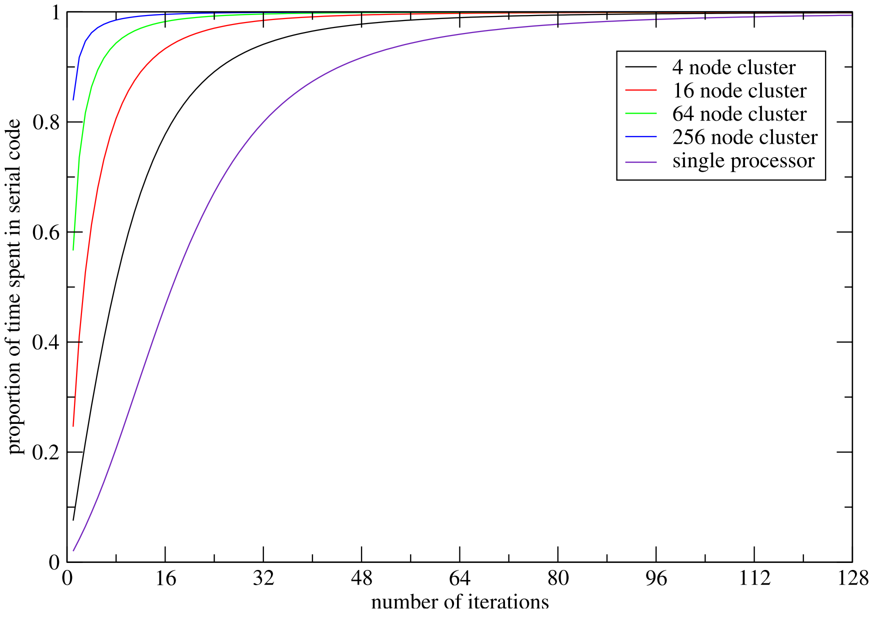


Figure 4. Shows proportion of overall runtime spent executing serial code for a program with a serial component that grows quadratically in the number of iterations. The program is assumed to have a serial component that grows quadratically in the number of iterations ($s = i^2 + 10i + 89$) and a fixed (constant) nonserial component of 4900. At i = 1, 2% of time is spent in serial code, but we see from the figure that such algorithms can quickly spend a great deal of their time executing in serial mode, thus failing to take advantage of the available parallel computing resources. The choices of constants defining the quadratic function (10,89,4900) are relatively unimportant here. The key point is that due to the quadratic dependence of runtime on i, the serial component eventually dominates over the parallel component.

where K is the number of iterations. This is just equation (5) with the T_3 term dropped, since step 3 is the only step performed in parallel.

[36] This bound also holds for many ensemble-based search algorithms, (such as the Genetic Algorithm, Simulated Annealing, and uniform sampling), if that algorithm is implemented according to Figure 2. With these other algorithms, however, $T_1(k)$ typically scales linearly with k, whereas in the Neighbourhood Algorithm, T_1 scales quadratically with k (see *Sambridge* [1999a] for the exact analysis), so the limiting behavior of Amdahl's law can come into play even on small clusters for the Neighbourhood Algorithm. Figure 4 illustrates the effect of quadratic growth in the serial component of a program.

4.2. Barrier Synchronization Overhead

[37] We refer to the second type of inefficiency as barrier synchronization overhead. It is a less obvious form of inefficiency than the limited parallelism discussed in section 4.1, and, on small clusters at least, is only noticeable for certain classes of problem, but its effect can be substantial, as we now demonstrate. Barrier synchronization overhead is a result of the need to collate all results on the "master" node in step 4. This collation results in wasted compute time on all nodes except the last to finish, since the main node (and all the other nodes) sit idle until results are collected from all nodes. This time-wasting synchronization occurs in every iteration of the canonical implementation. There are three situations, especially, where we expect a lot of time will be wasted in this manner:

[38] 1. In heterogeneous cluster environments, where processing power varies from node to node. The slowest node in this environment would likely be significantly slower than the average node.

[39] 2. In problems where the computational cost of the forward problem is parameter dependent. That is, the computational cost of evaluating the objective function $f(\mathbf{x})$ at point $\mathbf{x}$ depends on the value of $\mathbf{x}$.

[40] 3. In problems where external sources of variability, such as operating system and networking overhead, are significant compared with the cost of evaluating the objective function. Since any source of variability will result is wasted computation time, external sources of variability are just

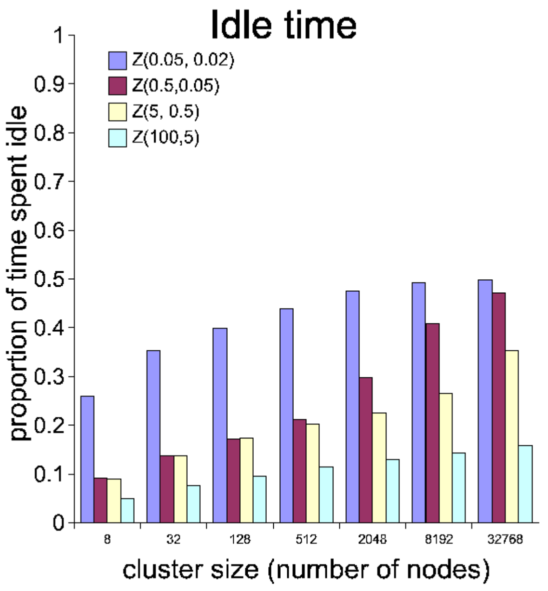


Figure 5. Proportion of time spent idle as a function of cluster size and size of the forward problem. We assume that the time taken to evaluate the objective function is described by a Gaussian distribution. $Z(\bar{x}, \theta)$ indicates a Gaussian with mean $\bar{x}$ and standard deviation θ. We also include a communication overhead of 0.2 milliseconds per node per iteration. The important point is that a significant proportion of time can be wasted idling in naively implemented parallel computations.

as harmful as intrinsic ones (as in case 2, above). External sources of variability typically become significant where the objective function is easy (i.e., quick) to evaluate.

[41] Now let us engage in a more thorough analysis. Remembering from equation (5) that $T(k)$ denotes the time taken to perform iteration k, let us denote the amount of actual computation (i.e., excluding idle time) performed on node n in that iteration k as $t_k(n)$.

[42] Referring back to equation (5), and noting that in practice, T_2 and T_4 are typically small compared to T_1 and/or T_3, we also make the following simplifying assumptions:

[43] $T_2(k) = C \ln N$. Network topology and message passing software limitations usually mean that distribution time is logarithmic or linear in the size of the cluster (N). We assume logarithmic with some multiplicative constant C.

[44] $T_4(k) = 0$. We assume that it takes no time to collate results back on the "master" node. Since T_4 is typically small, and since transmission delay can be modelled as part of stage 3 (T_3) anyway, we find it convenient to ignore this term.

[45] With these assumptions, for an N-node cluster equation (5) becomes

$$\begin{aligned} T(k) &= T_1(k) + T_2(k) + T_3(k) + T_4(k), \\ &= T_1(k) + C \ln N + t_k(\hat{n}), \end{aligned} \tag{7}$$

where $\hat{n} = \underset{n}{\operatorname{argmax}}\, t_k(n)$.

[46] As mentioned in section 4.1, $T_1(k)$ grows quadratically in the number of iterations (k). We now see that in each iteration, it is the slowest node that determines $T(k)$. Even an evenly divided computation on a homogenous cluster will, in practice, experience some variability. If it is always the slowest computer that determines $T(k)$, then compute time is wasted on all other nodes, while they wait for the slowest node to complete. Figure 5 shows that the proportion of time wasted during stage 3 (evaluation of the objective function) can be substantial, even when the variation in objective function evaluation time is small. The data for the figure came from a synthetic numerical simulation; in each synthetic iteration we select N numbers $n_1 \ldots n_N$ (one for each node in the synthetic cluster) from a Normal distribution. This gives the required execution time for each node. Calculating $\sum_{i=1}^{N} \max(n_1 \ldots n_N) - n_i$ gives the total time "wasted" by nodes waiting for the slowest one to finish.

4.3. The Two Effects in Combination

[47] When we consider limited parallelism and barrier synchronization overhead in combination, we see that many factors limit scalability in the current implementation of the Neighbourhood Algorithm. First, the quadratic growth (in k) of T_1, together with the fact that T_1 is entirely serial, means that Amdahl's law quickly comes into effect as the number of iterations increases, as Figure 3 shows. Second, the distribution, computation, and synchronized collation of results means that the amount of time spent idling by each node increases as the size of the cluster increases (see Figure 5). So, in concrete terms, we can expect that the following types of tasks will perform poorly: (1) tasks with a large number of iterations, (2) tasks performed on large clusters, and (3) tasks where there is a significant degree of variation in the time each node takes to perform its part of the computation, whether this be due to factors related to the computation, or external factors related to the computing environment. The first and second of these points are

worrying, given the trend toward larger clusters and correspondingly more intense computations. The third point is relevant in a large range of tasks. Flipping things around, we can say that it is only tasks performed on small homogenous clusters, requiring a small number of iterations, and exhibiting little variability in objective function evaluation time that will scale efficiently.

[48] In Figure 6, we plot the performance of the current implementation of the Neighbourhood Algorithm against the assumed maximum ($\epsilon = 1$). As we see, there is a stark difference between the current implementation of the Neighbourhood Algorithm and a maximally efficient parallel implementation.

5. New Approach

[49] We will show that, with a reformulation of the algorithm, it is possible to dramatically reduce the amount of wasted computation time, and to create a more efficient, scalable implementation. We have seen that the principal sources of inefficiency in the canonical implementation are as follows:

[50] 1. Generation of new models for evaluation is performed in serial, not parallel.

[51] 2. Collection of results involves a barrier synchronization.

[52] In order to remove these inefficiencies, we will do away with the concept of a "master" node that generates models, distributes them to nodes, and collects results. We will also do away with the whole concept of an "iteration" in order to avoid the need to synchronize at the end of each such iteration.

5.1. Removing the Need for a Master Node

[53] The concept of a "master" node is the cause of inefficiency in the current parallel implementation of the Neighbourhood Algorithm. In order to create an efficient and fault-tolerant implementation, we wish to make each node "independent," in some sense, of other nodes. This will ensure that the computation can continue if any other node(s) fail, and will also ensure that fast nodes are not "held up" by slower nodes. In order to make each node independent, we need each node to generate, for itself, the models whose objective function it will evaluate. We also need to change the way that nodes communicate results so that synchronization is not required. This approach, though, brings with it several difficulties. The first of these is that the concept of a Voronoi-cell composition of the parameter space that defines the Neighbourhood Algorithm requires knowledge of all previously evaluated models in order to generate new models for evaluation. (This is also the source of the quadratic growth in computational cost as the number of models increases.) Thus nodes cannot generate new models completely independently; they must continually communicate with other nodes to obtain an up-to-date list of models that have had their objective function evaluated. In addition, we need to make sure that there is no duplication, that two or more nodes do not evaluate the objective function of the same point. Our approach to overcome these problems is depicted in Figure 7. The important points about the new approach are as follows:

[54] 1. In step 1, each node independently generates/chooses the next model by performing a random walk within a Voronoi cell randomly selected from the best n_r Voronoi cells. (That is, the n_r Voronoi cells that contain the models with the objective function value.) This random selection of Voronoi cell means that while the expected number of samples drawn from within each of the best n_r Voronoi cells is the same, the actual number will be subject to variation. This is not the case in the canonical version, where the number of samples within each cell is fixed. One benefit of this approach is that nodes no longer need any arbitration or communication to determine in which Voronoi cell to perform a random walk.

[55] 2. In step 2, each node evaluates the objective function for the model generated in step 1.

[56] 3. In step 3, each node "posts" a message to all other nodes informing them of the newly generated model, and the value of the objective function for that model.

[57] 4. In step 4, each node checks its own post-box for postal messages from other nodes. It is important to note that such a "postal" system is asynchronous, meaning that each node does not wait for acknowledgment from other nodes. Moving to an asynchronous communication model brings us two important benefits over the synchronous communication model used in the canonical implementation of the Neighbourhood Algorithm: first, a software or hardware error on a single machine can no longer lock up the entire computation; second, slow nodes no longer hold up fast ones. The disadvantage of such an asynchronous scheme is that there is

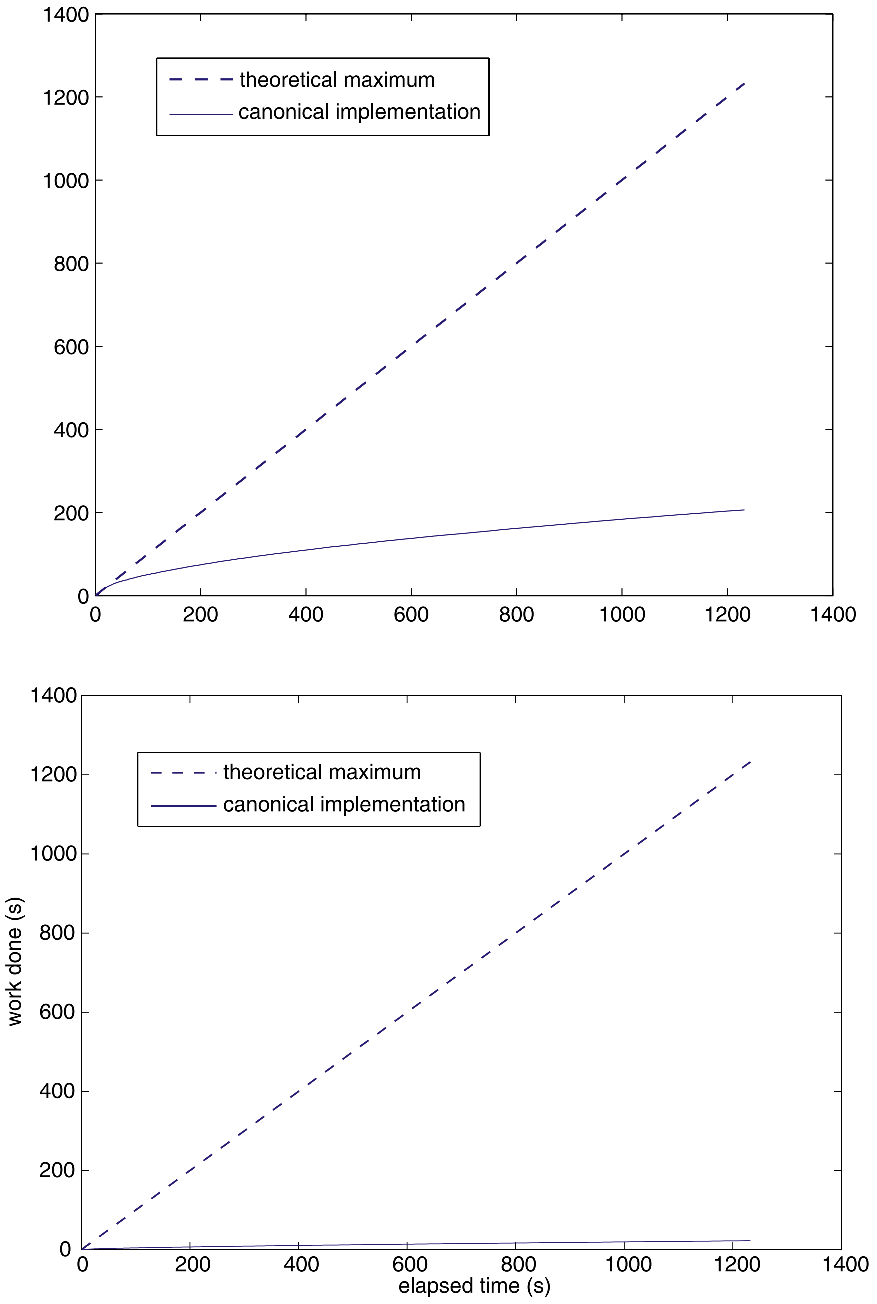


Figure 6. Performance of the canonical implementation on a 120-node cluster. The top (dashed) line shows the assumed upper bound (i.e., ϵ = 1). The colored lines show how close the current implementation of the Neighbourhood Algorithm gets to this maximum for inversions involving different forward problems. The top graph shows results from an inversion with a forward problem that takes ≈0.5 s and in the bottom a forward problem (adapted from *Yoshizawa and Kennett* [2002]) that takes ≈0.08 s.

overhead in performing the necessary message processing and buffering, but our results demonstrate that the benefits outweigh this cost.

[58] Notice that with the new formulation of the Neighbourhood Algorithm, the whole concept of an "iteration" has disappeared. This is a necessary precondition for the Neighbourhood Algorithm to be fault tolerant, because the definition of an iteration implies a barrier synchronization, which is both inefficient and fault-intolerant. In addition, this lack of a barrier synchronization eliminates the overhead we describe in section 4.2. The fact that each node now generates its own models (rather

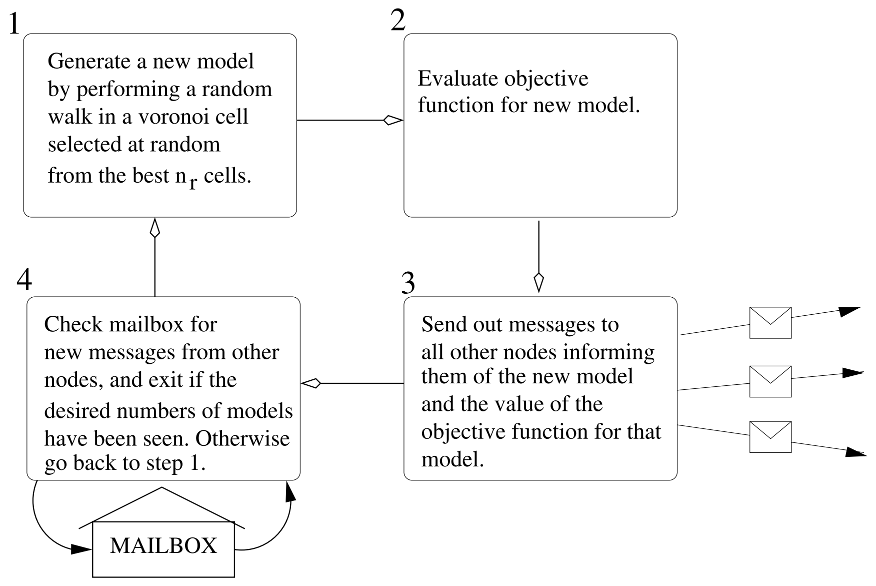

1
Generate a new model by performing a random walk in a voronoi cell selected at random from the best n_r cells.
2
Evaluate objective function for new model.
3
Send out messages to all other nodes informing them of the new model and the value of the objective function for that model.
4
Check mailbox for new messages from other nodes, and exit if the desired numbers of models have been seen. Otherwise go back to step 1.
MAILBOX

Figure 7. Outline of control logic for each independent node in the new implementation of the Neighbourhood Algorithm.

than receiving them from the master node) means that the limited parallelism described in section 4.1 is also eliminated, because in the canonical implementation, the generation of models was performed in serial on a "master" node, but now model generation is performed in parallel.

[59] The removal of barrier synchronization also means that each node "sees" a slightly different set of Voronoi cells at the time of model generation. Overall, however, the asynchronous postal system of communication ensures that each node eventually receives all models for which the forward problem has been solved, and hence (on average) the same set of Voronoi cells.

[60] Given that there is no concept of an iteration in the new formulation of the Neighbourhood Algorithm, an analytic comparison of the two schemes is difficult. Instead we rely principally on results obtained by timing the old (canonical) implementation and new implementation on published inversion problems. Since the "new" algorithm does not have any concept of an iteration, we run both old and new algorithms for an equal number of objective function evaluations. This allows direct comparison of timing results. We also use a 2-D test function to compare the actual sampling differences.

6. Experimental Results

[61] Our preceding analysis dealt with the sources of inefficiency in the original formulation of the Neighbourhood Algorithm and what effect they have for different types of forward modelling. We then presented a new method that eliminates the principal sources of inefficiency. We now make the discussion more complete, by comparing performance of the new algorithm to inverse problems previously tackled with the canonical form of the Neighbourhood Algorithm.

[62] First, however, we illustrate that the new formulation of the Neighbourhood Algorithm retains the essential characteristics of the original, by performing some numerical experiments on sampling densities. Figure 8 shows samples generated by three separate runs of the Neighbourhood algorithm applied to minimization of the Rosenbrock function (see Figure 9), a standard test function for optimization algorithms [see *Gill et al.*, 1981]. The first two rows of Figure 8 are samples from the original NA formulation using different random seeds, while the third uses the new implementation. We see that the differences in sampling density between the new and the original implementation of the NA are rather slight, and certainly no greater than that produced by different random seeds. Hence, in this case, the sampling densities are comparable.

[63] For a comparison on geophysical inverse problems we have selected two "test" problems, one in the inversion of seismic receiver functions [*Yoshizawa and Kennett*, 2002] and one in inversion of thermochronological data [*Braun and Robert*, 2005; *Braun and van der Beek*, 2004].

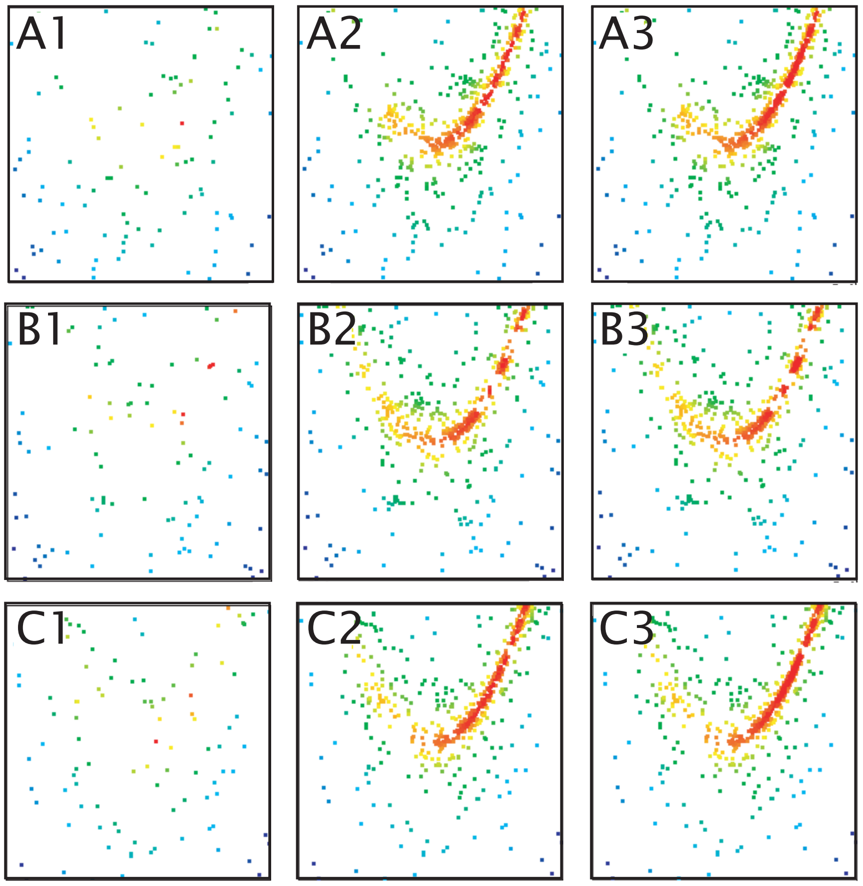


Figure 8. A comparison of the canonical and new implementations applied to the well-known 2-D Rosenbrock function (see Figure 9). Rows A and B were produced from two separate runs of the canonical NA using different random seeds. Row C shows a run of the new algorithm. In all cases, samples are generated with $n_r = 50$, $n_s = 100$. Samples are colored by the values of the Rosenbrock function, which has a global minimum at (1,1). Here all axes have range (−2,2). Column 1 shows the initial uniform random set of samples in each case. Columns 2 and 3 show the sampling density after generation of 1100 and 2100 samples, respectively. The differences in the sampling density between the new and canonical versions of the NA are comparable to that between different random seeds.

For completeness, we briefly describe each inversion problem before presenting the experimental results.

6.1. Receiver Function Inversion

[64] The first example is the inversion of seismic receiver functions for the shear velocity structure of the crust. This is a well-known nonlinear problem in seismology [*Ammon et al.*, 1990] and was the example originally used to illustrate the Neighbourhood Algorithm [*Sambridge*, 1999a]. Many authors have subsequently applied the canonical form of the Neighbourhood Algorithm to this problem [e.g., [*Yoshizawa and Kennett*, 2002; *Reading et al.*, 2003]. The observations are multi-component seismic waveforms of distant earthquakes, from which receiver functions are produced with sensitivity only to Earth structure beneath the recording station. Here the shear wave velocity structure as a function of depth is represented by 24 unknowns and the Neighbourhood Algorithm is used to explore this space. In this case the relationship between observations and unknowns is nonlinear the numerical solution of the forward problem is relatively efficient [see *Shibutani et al.*, 1996; *Thomson*, 1950; *Haskell*, 1953].

6.2. Thermochronology

[65] The second example comes from a nonlinear inverse problem in low-temperature thermochronology [*Braun and Robert*, 2005; *Braun and van der Beek*, 2004]. Here the data are age distributions of rocks obtained from thermochronology and these are compared with predictions from a numerical calculation. Specifically each age prediction involves the solution of the heat transport equation

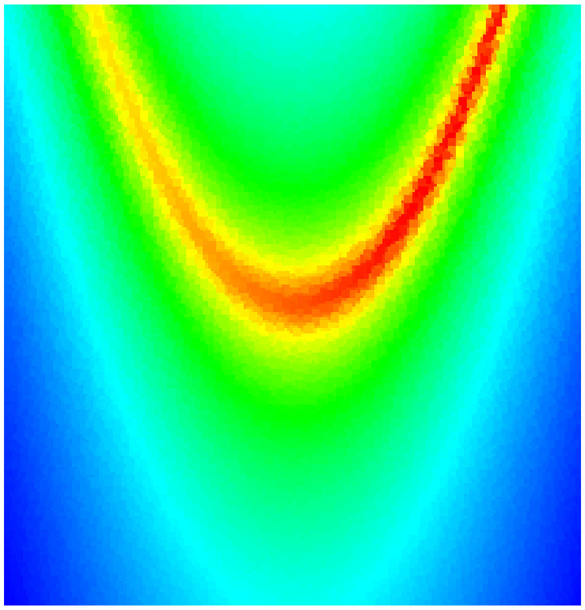


Figure 9. The 2-D Rosenbrock function used to illustrate sampling density of the original and new implementations of the NA (see Figure 8).

in the Earth's crust including the effects of finite amplitude topography and flexural isostasy. In this case the forward calculation involves the solution of a differential equation using a finite element method. The Neighbourhood Algorithm is used to search through a parameter space of three variables defining various tectonic-morphic scenario's (see *Braun and Robert* [2005] and *Braun and van der Beek* [2004] for details). This problem provides a contrast to the receiver function (RF) inversion, having a forward problem with many fewer unknowns that is much more computationally expensive (approximately by a factor of 5000).

6.3. Methodology

[66] The cluster used is a commodity Linux cluster of 120 Pentium 4 nodes (2.4 GHz) with fast (1066 MHz) memory and Gigabit Ethernet. Both codes (i.e., the canonical version and the new version) use Argonne National Labs implementation of MPI (MPICH 1.5.9). As already noted, MPI is inherently brittle to failure, so these results measure performance increases only. We simply make the point that the new algorithm, in addition to having better performance than the canonical version (as we will soon show), also has the advantage that it can execute in a messaging environment that continues in the event of node failure. The structure of the canonical implementation, in contrast, means that the computation will stall forever even if such a messaging substrate is available.

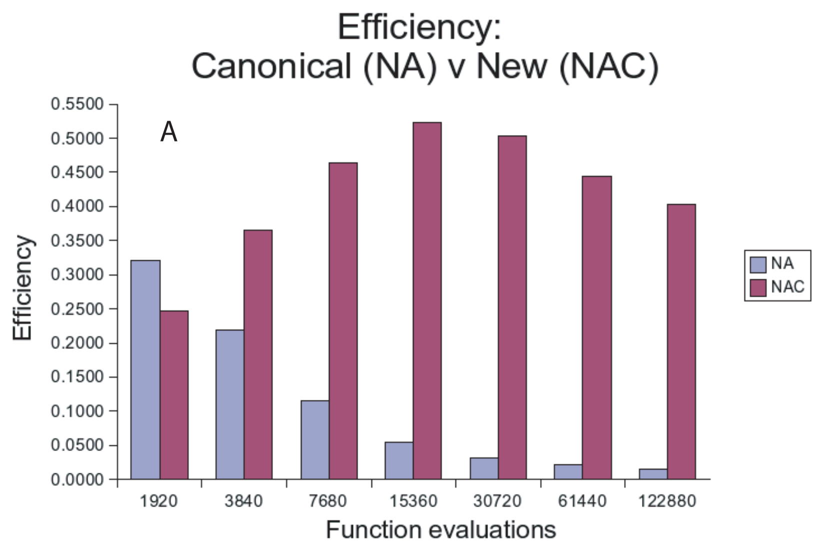
[67] The running time of both old and new versions of the Neighbourhood Algorithm were chosen so that, for both problems, the number of objective function evaluations was the same. Wall time and CPU time for each computation is measured through the use of standard library calls.

6.4. Results

[68] The left chart in Figure 10 compares the efficiency (as defined in section 3.2) of the new and old implementations on a cluster of 120 nodes for the receiver function of *Yoshizawa and Kennett* [2002]. We see that, beyond a small number of iterations, the new implementation is much more efficient than the old one. The discrepancy at 1920 function evaluations, where the old implementation is more efficient than the new one is due to the fact that the execution time is fast enough (<3 seconds) that some additional initial overhead in the new implementation is significant.

[69] From our analysis, we would expect the new algorithm to perform better on the RF problem for two reasons. First, this computation has a large number of iterations. Second, the objective function is fast to evaluate (≈ 0.05 seconds), so that transient factors introduce significant variation in the time taken to evaluate this function. This is exactly the case where we expect the largest performance gains, and we see in the right hand chart of Figure 10 that beyond 100000 function evaluations, the new implementation executes in less than one twentieth of the time, with efficiency around 0.4–0.5, whereas the canonical algorithm's efficiency quickly drops to less than 0.02.

[70] The thermochronology inversion has an objective function that takes much more computational effort to evaluate, of the order of 5 minutes for a single evaluation, but with significant variation, depending on the point in the domain where the function is evaluated. So, in some sections of the domain, computation time may be of the order of a minute faster or slower. This fact, and the consequence that a smaller number of iterations can be performed due to the computational requirements of object function evaluation, mean that we expect only minor improvements to performance for the new implementation of the Neighbourhood Algorithm. Specifically, it is only the variation in objective function evaluation time that allows the newer version to perform better. For this case Figure 5



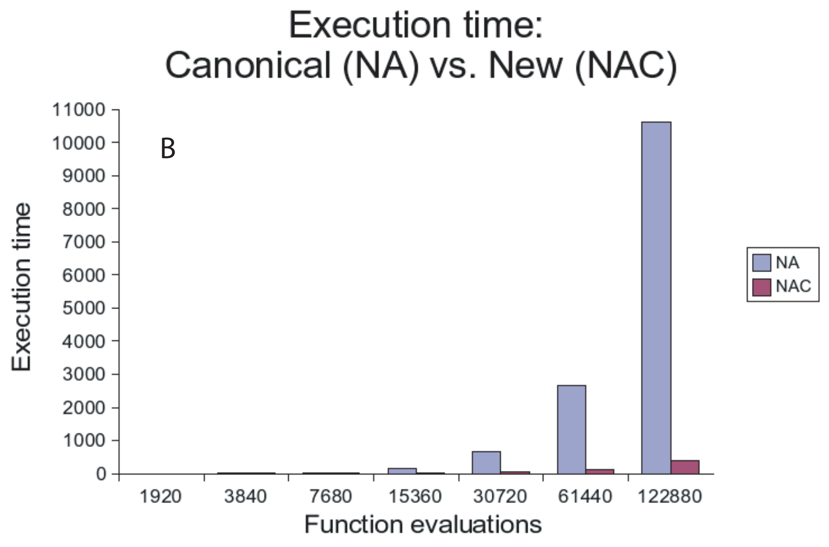


Figure 10. Comparison in efficiency of new and old implementations for the fast forward problem (i.e., short runtime) used in the work of *Yoshizawa and Kennett* [2002]. In Figure 10a we compare the efficiency of the new and old implementations. In Figure 10b we compare execution time as the number of function evaluations is increased. Here the time for the canonical implementation varies between 2.78 s (at 1920 evaluations) and 10611.49 s (at 120000 evaluations), while the new implementation varies between 3.62 s and 406.74 s.

suggests that we would get only a small improvement in performance with the new implementation. Figure 11 confirms that this is indeed the case, with a clear, but small, increase in efficiency and corresponding decrease in execution time.

7. Conclusion

[71] The computational power available to the scientific community has grown dramatically in the last decade, but the software available to efficiently use this computational power can still fail to make efficient use of these resources. Paying insufficient attention to algorithmic design issues is often a mistake, even if hardware improvements continue to make computing power faster and cheaper. Indeed, our specific analysis of a popular geophysical inversion method shows that, in many cases, parallel performance asymptotes very quickly for realistic problems.

[72] We believe that, in many cases, an analysis of an algorithm's distributed performance can allow for a redesign that is significantly more efficient. We have performed just such analysis for the

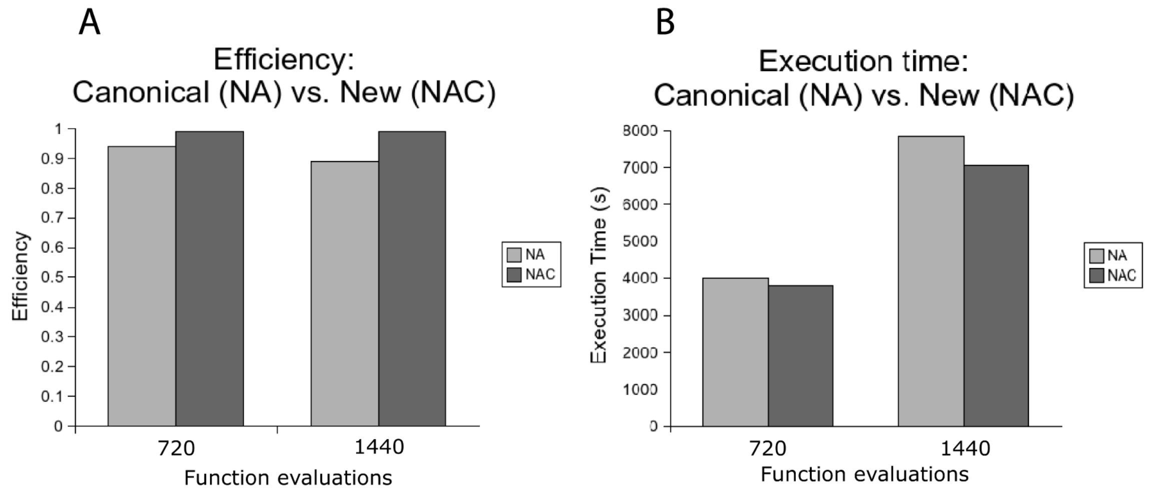


Figure 11. Comparison of efficiency of new and old implementations on a computationally expensive forward problem, taken from *Braun and van der Beek* [2004]. Figure 11a shows efficiency of the new and old implementations. Figure 11b shows time taken against number of function evaluations.

Neighbourhood Algorithm, shown how a reformulation is possible, and, more constructively, implemented the reformulated scheme. The result is a fault-resistant algorithm, that does not suffer from the deficiencies outlined in sections 4.2 and 4.1. For a large class of problems, it is substantially more efficient than the canonical version, and becomes increasingly more efficient as the size of the cluster increases. We have achieved this efficiency increase by relaxing some nonessential properties of the canonical implementation of the Neighbourhood Algorithm.

[73] Given trends in computer performance and price, distributed computing offers great potential to scientists who are able to harness it, but much current software scales poorly over a cluster, and little software is designed to survive frequent node failure on large clusters. Simply stated, we believe that a passive approach, where we wait for hardware improvements to translate into faster computation times, is often a mistake, and a more active approach to algorithm design is needed to allow us to run efficiently on large cluster computers. We have illustrated such an approach in the redesign of a common inversion algorithm, and showed not only significant increases in performance, but that the efficiency of the new implementation will increase as the size of the cluster increases, compared with the original implementation.

[74] One issue that arises in designing algorithms to be hardware fault intolerant is that at present many message passing libraries (such as MPI) are themselves not fault tolerant. Hence this aspect of the new algorithm is lost if an implementation makes use of currently available MPI libraries. It is possible, however, to implement the new algorithm using standard sockets programming, or a failure-resistant message passing library, and hence obtain a fault-tolerant program. This is not the case for the old implementation. As cluster sizes increase, failure-tolerant message-passing libraries will become the norm.

Acknowledgments

[75] We thank Jean Braun for providing software for the thermochronology example used in section 6.2. Jeannot Trampert and an anonymous reviewer provided constructive comments on an earlier draft of this manuscript. An implementation of the Neighbourhood Algorithm based on the ideas described in this paper (including fault tolerance) is available from the authors.

References

Agostinetti, N. P., G. Spada, and S. Cianetti (2004), Mantle viscosity inference: A comparison between simulated annealing and neighbourhood algorithm inversion methods, *Geophys. J. Int.*, *157*(2), 890–900, doi:10.1111/j.1365-246X.2004.02237.x.

Amdahl, G. M. (1967), Validity of the single-processor appoach to achieving large scale computing capabilities, in *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, AFIPS Press, Reston, Va.

Ammon, C. J., G. E. Randall, and G. Zandt (1990), On the nonuniqueness of receiver function inversions, *J. Geophys. Res.*, *95*(B10), 15,303–15,318.

Braun, J., and X. Robert (2005), Constraints on the rate of post-orogenic erosional decay from low-temperature thermochronological data: Application to the Dabie Shan, China, *Earth Surf. Processes Landforms*, *30*(9), 1203–1225.

Braun, J., and P. van der Beek (2004), Evolution of passive margin escarpments: What can we learn from low-temperature thermochronology?, *J. Geophys. Res.*, *109*, F04009, doi:10.1029/2004JF000147.

Gill, P. E., W. Murray, and M. H. Wright (1981), *Practical Optimization*, Elsevier, New York.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Boston, Mass.

Haskell, N. A. (1953), The dispersion of surface waves in multilayered media, *Bull. Seismol. Soc. Am.*, *43*, 17–34.

Holland, J. H. (1992a), *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*, MIT Press, Cambridge, Mass.

Holland, J. H. (1992b), Genetic algorithms, *Sci. Am.*, *267*, 66–72.

Kennett, B. L. N., K. Marson-Pidgeon, and M. Sambridge (2000), Seismic source characterisation using a Neighbourhood Algorithm, *Geophys. Res. Lett.*, *27*(20), 3401–3404.

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi (1983), Optimization by simulated annealing, *Science*, *220*(4598), 671–680.

Lohman, R. B., M. Simons, and B. Savage (2002), Location and mechanism of the Little Skull Mountain earthquake as constrained by satellite radar interferometry and seismic waveform modeling, *J. Geophys. Res.*, *107*(B6), 2118, doi:10.1029/2001JB000627.

McKee, S. (1995), Hitting the memory wall: Implications of the obvious, *Comput. Archit. News*, *23*(1), 20–24.

Message Passing Interface Forum (1994), MPI: A message-passing interface standard, *Int. J. Supercomput. Appl.*, *8*, 165–414.

Moore, G. E. (1965), Cramming more components onto integrated circuits, *Electronics*, *38*(8), 114–117.

Mosegaard, K., and M. Sambridge (2002), Monte Carlo analysis of inverse problems, *Inverse Problems*, *18*, r29–r54.

Okabe, A., B. Boots, and K. Sugihara (1992), *Spatial tessellations: Concepts and applications of Voronoi diagrams*, John Wiley, Hoboken, N. J.

Pritchard, M. E., and M. Simons (2002), A satellite geodetic survey of deformation of volcanic centres in the central Andes, *Nature*, *418*, 167–171.

Reading, A., B. Kennett, and M. Sambridge (2003), Improved inversion for seismic structure using transformed, S-wave-vector receiver functions: Removing the effect of the free surface, *Geophys. Res. Lett.*, *30*(19), 1981, doi:10.1029/2003GL018090.

Sambridge, M. (1998), Searching multi-dimensional landscapes without a map, *Inverse Problems*, *14*, 427–440.

Sambridge, M. (1999a), Geophysical inversion with a neighbourhood algorithm—i. Searching a parameter space, *Geophys. J. Int.*, *138*, 479–494.

Sambridge, M. (1999b), Geophysical inversion with a neighbourhood algorithm—ii. Appraising the ensemble, *Geophys. J. Int.*, *138*, 727–746.

Sambridge, M., and K. Mosegaard (2002), Monte Carlo methods in geophysical inverse problems, *Rev. Geophys.*, *40*(3), 1009, doi:10.1029/2000RG000089.

Sen, M. K., and P. L. Stoffa (1991), Nonlinear one-dimensional seismic waveform inversion using simulated annealing, *Geophysics*, *56*, 1624–1638.

Sen, M. K., and P. L. Stoffa (1992), Rapid sampling of model space using genetic algorithms, *Geophys. J. Int.*, *108*, 281–292.

Sen, M. K., and P. L. Stoffa (1995), *Global Optimization Methods in Geophysical Inversion*, *Adv. Explor. Geophys.*, vol. 4, Elsevier, New York.

Sherrington, H. F., G. Zandt, and A. Frederiksen (2004), Crustal fabric in the Tibetan Plateau based on waveform inversions for seismic anisotropy parameters, *J. Geophys. Res.*, *109*, B02312, doi:10.1029/2002JB002345.

Shibutani, T., M. Sambridge, and B. Kennett (1996), Genetic algorithm inversion for receiver functions with application to crust and uppermost mantle structure beneath eastern Australia, *Geophys. Res. Lett.*, *23*(14), 1829–1832.

Snir, M., S. W. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra (1998), *MPI—The Complete Reference: Volume 1, The MPI Core*, 2nd ed., MIT Press, Cambridge, Mass.

Subbey, S., M. Christie, and M. Sambridge (2004a), Prediction under uncertainty in reservoir modelling, *J. Pet. Sci. Eng.*, *44*, 143–153.

Subbey, S., M. Christie, and M. Sambridge (2004b), The impact of uncertain centrifuge capillary pressure on reservoir simulation, *SIAM J. Sci. Comput.*, *26*(2), 537–557.

Thomson, W. T. (1950), Transmission of elastic waves through a stratified solid, *J. Appl. Phys.*, *21*, 89–93.

Yoshizawa, K., and B. Kennett (2002), Non-linear waveform inversion for surface waves with a neighbourhood algorithm—Application to multimode dispersion measurements, *Geophys. J. Int.*, *149*, 118–133.