Comput Econ
DOI 10.1007/s10614-013-9377-8

Nelder-Mead Simplex Optimization Routine for Large-Scale Problems: A Distributed Memory Implementation

Kyle Klein · Julian Neira

Accepted: 12 March 2013

Abstract The Nelder-Mead simplex method is an optimization routine that works well with irregular objective functions. For a function of n parameters, it compares the objective function at the $n + 1$ vertices of a simplex and updates the worst vertex through simplex search steps. However, a standard serial implementation can be prohibitively expensive for optimizations over a large number of parameters. We describe an implementation of the Nelder-Mead method in parallel using a distributed memory. For p processors, each processor is assigned $(n + 1)/p$ vertices at each iteration. Each processor then updates its worst local vertices, communicates the results, and a new simplex is formed with the vertices from all processors. We also describe how the algorithm can be implemented with only two MPI commands. In simulations, our implementation exhibits large speedups and is scalable to large problem sizes.

Keywords Parallel computing · Optimization algorithms · Nelder-Mead

JEL Classification C63

1 Introduction

The Nelder and Mead (1965) simplex method (Nelder-Mead) is an optimization routine that is commonly used in problems with irregular objective functions. It is a robust method that can overcome kinks, local solutions, indeterminacies, and discontinuities

Our computer code is available at Neira's website. We thank Hrishikesh Singhania for excellent comments and discussion.

K. Klein · J. Neira (✉)
U.C. Santa Barbara, Santa Barbara, CA, USA
e-mail: neira@econ.ucsb.edu

Published online: 31 March 2013

in functions being evaluated. However, a standard serial implementation can be prohibitively expensive for problems with a large number of parameters or with costly objective functions. In this paper, we propose a method to implement Nelder-Mead in parallel using a distributed memory implementation.

Nelder-Mead minimizes a function of n parameters by comparing function evaluations at the $n + 1$ vertices of a general simplex, and updating the worst vertex by moving it around a *centroid*. A centroid is a new vertex that is an average of the remaining (best) vertices. We modify Nelder-Mead in the following way. For p processors, each processor is assigned $(n + 1)/p$ vertices at each iteration. Each processor then updates its $k > 1$ worst *local* vertices, communicates the results, and a new simplex is formed with the vertices from all processors. We also describe how the algorithm can be implemented with only two Message Passing Interface (MPI) commands, the most widely used library for distributed memory parallelization.

There are two separate sources of speedups in our implementation. One source is the speedup achieved by updating more than one vertex per iteration. This gain arises from moving vertices in a better direction, as the worse vertices (those being updated) are not used to calculate the centroid. Notice this gain is not inherently parallel; an implementation with one processor would still converge faster due to better search directions. Lee and Wiswall (2007) explored a version of this source of speedups in Nelder-Mead. One contribution of our paper is to document that updating more than one vertex per iteration achieves much larger speedups for larger problem sizes than those considered by Lee and Wiswall (2007). The second source of speedups in our implementation comes from updating the worst local vertices to each processor. This aspect of the implementation reduces communication overhead between processors.

We test the performance of our implementation on two sample functions through simulations. To distinguish between the computational gains from both sources of speedups, we first perform the simulations on one processor and then on multiple processors. With one processor, we find that updating more than one vertex per iteration achieves a large speedup thanks to a reduction in the number of total function evaluations and the number of times the centroid is calculated. Using multiple processors, we then implement a distributed memory version of the method. We find that updating the worst local vertices to each processor significantly reduces communication overhead costs without increasing total function evaluations. Small communication overhead leads us to achieve almost linear speedup in the number of processors. Linear speedup implies the implementation is scalable to large-scale problems and the speedup is proportional to the number of processors.

Lee and Wiswall (2007) were the first to describe how significant speedups can be achieved in Nelder-Mead by updating more than one vertex per iteration. These authors considered optimizations over 100 and 200 parameters in their simulations. They found that their implementation performed up to 3 times less function evaluations before convergence. Using the same objective functions as Lee and Wiswall (2007), we find that updating more than one vertex per iteration yields significantly larger gains for larger problems than those considered by the authors. Specifically, function evaluations are reduced by up to a factor of 12 for an optimization over 1,000 parameters. This finding is relevant because problems in calibration and structural estimation often require optimizations over more than 1,000 parameters. For example,

Huggett et al. (2006) calibrate an empirical bivariate distribution. They create a mesh grid and minimize the distance between data and model at each grid-point. This requires a minimization over m^2 parameters, where m is the number of grid-points of the individual grids. If fine individual grids are desirable, the number of parameters in the minimization procedure can run in the thousands. Structural estimation of life-cycle models with several sources of heterogeneity can also yield optimizations over a large number of parameters. Lawver (2012) is one such example, with an estimation over more than 6,000 parameters.

Unconstrained optimization is common in economic problems involving calibration or structural estimation. There are two ways in which the literature has attempted to parallelize problems of this type. One way is to break down the objective function into separate regions and assigns each region to a different processor. Swann (2002), Creel (2005), Ferrall (2005), and Aldrich et al. (2011) are examples of this approach. A second way is to break down the minimization algorithm into parallel tasks, as is done in this paper (examples include Dennis et al. (1991) and Beaumont and Bradshaw (1995)). An advantage of parallelizing the minimization algorithm is that code does not need to be rewritten for every new problem. Rather, a generic parallel optimization algorithm can be applied to different problems.

The paper proceeds as follows. Section 2 describes the serial Nelder-Mead simplex method. Section 3 describes the modifications required to implement the algorithm in parallel using distributed memory. Section 4 discusses the experiments and results. Section 5 concludes.

2 Serial Nelder-Mead

This section describes the serial Nelder and Mead (1965) simplex method using an example. Precise notation for each step of the algorithm is introduced in the next section.

The idea behind Nelder-Mead is to "crawl" down to the minimum value by moving a simplex one vertex at a time.[1] The vertices are moved by performing four basic operations: Reflection, Expansion, Contraction, and Multiple Contraction (shrink). The steps the simplex can take are illustrated in Fig. 1.

To illustrate, suppose we are minimizing f, a function of two parameters θ_1 and θ_2,

$$f(\theta_1, \theta_2) = |\theta_1| + |\theta_2| \tag{1}$$

The solution is, trivially, $(\theta_1, \theta_2) = (0, 0)$. Nelder-Mead solves the problem in (1) as follows.

Step 1: Initialize the simplex

Given an initial guess, the method creates a simplex by adding a step of a given magnitude to each parameter. For n parameters, a simplex is an n-dimensional polytope which is the convex hull of its $n + 1$ vertices. For a two-parameter minimization,

[1] For this reason the algorithm is also known as the "amoeba" method.

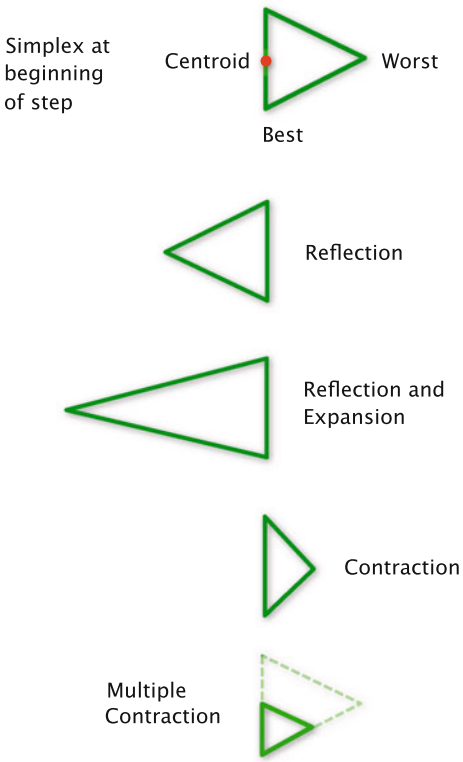

Fig. 1 Operations of Nelder-Mead for an optimization over 2 parameters. See Press et al. 2007 for an example when the simplex is a tetrahedron

a simplex is a triangle. For initial guess $(\theta_1, \theta_2) = (1, 1)$, the method can create the other vertices by adding a step of 1 in θ_1 and 2 in θ_2. The other two vertices of the triangle are then (2, 1) and (1, 3).

Step 2: Sort vertices and compute centroid

Evaluate the objective function at all vertices and rank them in increasing order (breaking ties in some arbitrary way). For example (1), the lowest (best) vertex is $A_0 = (1, 1)$ and the highest (worst) is $A_2 = (1, 3)$. Compute a centroid with all vertices except the worst. The centroid in example (1) is $M = \left(\frac{1+2}{2}, \frac{1+1}{2}\right) = (1.5, 1)$.

Step 3: Update the worst vertex or shrink simplex

Reflect the worst vertex. If the objective function has a lower value at the reflected vertex than the current best vertex, then the algorithm moves on to reflection and expansion and goes back to step 2. If it is not an improvement over the best vertex but it is an improvement over the second worst vertex, then the reflected vertex replaces the worst vertex and the algorithm goes back to step 2.

If the reflected vertex is not an improvement over the second worst vertex, the algorithm tries a contraction. If the contraction is better than the worst vertex, it replaces the worst vertex with the contracted vertex and the algorithm goes back to

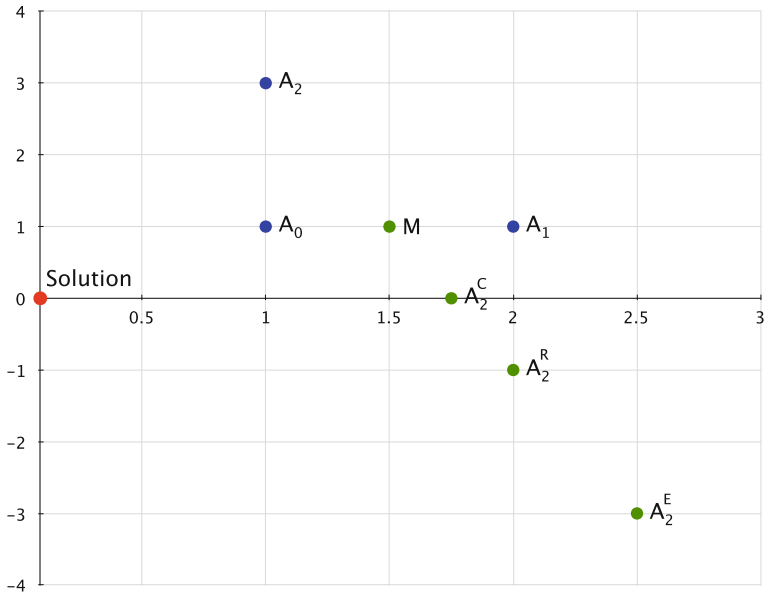

Fig. 2 Sample operations of Nelder-Mead in the first iteration for example (1).

step 2. Otherwise, if the contracted vertex is worse than the worst vertex, the algorithm shrinks the whole simplex around the current best vertex. The algorithm goes back to step 2 and iterates until some convergence criterion is reached.

For example (1), the vertex $A_2^R = (2, -1)$ is the reflection of the worst vertex $A_2 = (1, 3)$. The vertex $A_2^E = (2.5, -3)$ is the reflected and expanded vertex. The contracted vertex is the midpoint between the reflected vertex and the centroid, $A_2^C = (1.75, 0)$. Since the reflected vertex is an improvement over the worst vertex but not the expansion, vertex A_2 is replaced by vertex A_2^R and the algorithm goes back to step 2. The range of steps for example (1) is illustrated in Fig. 2.

Once a solution is reached, the simplex should be re-initialized using the solution as an initial guess to reduce the probability of convergence to a local solution.

3 Parallel Nelder-Mead

We now describe the implementation in Lee and Wiswall (2007), and then describe our distributed memory implementation.

3.1 Lee and Wiswall's Implementation

Nelder-Mead updates one vertex per iteration. Lee and Wiswall (2007) modify Nelder-Mead by updating more than vertex per iteration. Each of those vertices could in theory be assigned to its own processor. The Lee and Wiswall (2007) implementation has the following steps.

Step 1 : Initialize the simplex.

Given an initial guess $A_0 = [\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n]$, and steps $s_1, s_2 \ldots, s_n$, create the vertices of the initial simplex as follows:

$$\begin{aligned} A_0 &= [\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n] \\ A_1 &= [\theta_1 + s_1, \theta_2, \ldots, \theta_{n-1}, \theta_n] \\ A_2 &= [\theta_1, \theta_2 + s_2, \ldots, \theta_{n-1}, \theta_n] \\ &\vdots \\ A_{n-1} &= [\theta_1, \theta_2, \ldots, \theta_{n-1} + s_{n-1}, \theta_n] \\ A_n &= [\theta_1, \theta_2, \ldots, \theta_{n-1}, \theta_n + s_n] \end{aligned}$$

Evaluate the objective function f at each vertex and re-label the simplex vertices so that

$$f(A_0) \leq f(A_1) \cdots \leq f(A_n).$$

Step 2 : Update the k worst vertices

Since each of the k updates is performed independently, we could assign a single vertex to each processor and use k processors. Or if we had $p < k$ processors, each could update k/p vertices. Since the simplex is sorted, the k worst vertices are $A_{n-k+1}, A_{n-k+2}, \ldots, A_n$. For each of the k worst vertices A_i, we compute an updated version A_i^* as follows.

Compute the centroid M as

$$M = \frac{1}{n} \sum_{i=0}^{n-k} A_i$$

For each vertex A_i among the k worst vertices calculate A_i's reflection as

$$A_i^R = M + \alpha(M - A_i)$$

where $\alpha > 0$ is an algorithm parameter, typically $\alpha = 1$. The objective function is evaluated at every reflection, and the instructions followed from here are nearly analogous to the serial Nelder-Mead.

If $f\left(A_i^R\right) < f(A_0)$, then compute the expansion

$$A_i^E = A_i^R + \gamma\left(A_i^R - M\right)$$

where $\gamma > 0$ is an algorithm parameter, typically $\gamma = 1$. If $f\left(A_i^E\right) < f(A_0)$, then $A_i^* = A_i^E$, otherwise $A_i^* = A_i^R$.

If $f\left(A_i^R\right) > f(A_0)$ and $f\left(A_i^R\right) < f\left(A_{i-1}^R\right)$, then $A_i^* = A_i^R$. Otherwise, calculate the contraction

$$A_i^C = \beta(M + \widetilde{A}_i)$$

where $\widetilde{A}_i = A_i^R$ if $f\left(A_i^R\right) < f(A_i)$, and $\widetilde{A}_i = A_i$ otherwise. $0 < \beta < 1$ is an algorithm parameter, typically $\beta = 1/2$. If $f\left(A_i^C\right) < f(A_i)$, then $A_i^* = A_i^C$. Otherwise $A_i^* = \widetilde{A}_i$, which is either A_i or A_i^R.

Step 3 : Update the simplex

Each updated vertex A_i^* belongs to the set $\{A_i, A_i^R, A_i^E, A_i^C\}$. If $A_i^* = A_i$ and $f\left(A_i^R\right) < f(A_{i-1})$ for at least one of the k vertices, then the new simplex is:

$$[A_0, A_1, \ldots, A_{n-k}, A_{n-k+1}^*, \ldots, A_n^*]$$

Otherwise, shrink the simplex toward the best vertex A_0. The shrunk simplex is

$$\begin{aligned}[A_0, (\tau A_0 + (1-\tau)A_1), \ldots, (\tau A_0 + (1-\tau)A_{n-k}),\\ (\tau A_0 + (1-\tau)\widetilde{A}_{n-k+1}), \ldots, (\tau A_0 + (1-\tau)\widetilde{A}_n)]\end{aligned}$$

where $0 < \tau < 1$ is an algorithm parameter, typically $\tau = 1/2$. Re-sort the simplex vertices and return to step 2 until the convergence criterion is met.

3.2 A Distributed Memory Implementation

As parallel problems grow larger, it eventually becomes necessary to distribute the memory amongst the processors. This case is no different. A problem size of n parameters consumes $O(n^2)$ memory to store the simplex, quickly requiring more memory capacity than a single machine can handle.[2] Thus, in order to effectively scale this method to large problems, we distribute the simplex vertices across the processors. In particular, if there are p processors each processor is assigned $(n+1)/p$ vertices. The remainder of this section discusses how the algorithm must be adapted to account for the fact that each processor only has access to part of the simplex.

Our implementation is written in $C++$ and uses the MPI library for communication between processors. From this library we only need two routines: `MPI_AllReduce` and `MPI_Bcast`. We briefly describe them here.

`MPI_AllReduce` instructs each processor to calculate a value. It then combines values from all processors and distributes the result back to all processes. All processors call this routine simultaneously. For example, to determine the best value of the objective function across the entire simplex, each processor calls this routine and provides the best vertex in the piece of the simplex it holds. An argument to `MPI_AllReduce` specifies that the minimum value needs to be found. The minimum among all processors is picked and then distributed to all processors. This may also be used to sum the value of a parameter (or sum entire vectors) across all processors and distribute the answer.

[2] Big O notation is used to describe the limiting behavior of a function as its argument tends toward infinity. Essentially, it allows one to express the function in terms of only its dominant terms. Formally, a function $h(n) = O(g(n))$ if and only if there exists a positive real number M and a real number n_0 such that $|h(n)| \leq M \cdot |g(n)|$ for all $n \geq n_0$.

`MPI_Bcast` designates one processor as the root node. The root node then broadcasts a value to all other processors. This routine is invoked when we need to shrink the simplex. The processor with the best vertex is designated as the root node and it broadcasts the vertex to all the other processors.[3]

The Lee and Wiswall (2007) implementation updates the k worst vertices in parallel. Implementing this exact algorithm with distributed memory can be a computationally expensive task, as we first need to sort the distributed data. Further, the data must then be distributed such that each of the p processors has k/p of the worst vertices. In order to avoid this costly computation, we modify Lee and Wiswall (2007) and update the k/p worst vertices that are *local* to each processor. This introduces a tradeoff: On the one hand, we reduce costly communication between processors. On the other hand, some vertices not within the k worst *global* vertices will be updated, and vertices in the worst k global vertices will be used to calculate the centroid. This can increase the number of iterations before convergence. For our simulations, we find that updating local vertices has a minimal effect on performance, and thus this is an effective design decision.

Next, we describe the portions of the Lee and Wiswall (2007) implementation which require changes in order to use distributed memory.

Centroid: Each processor now calculates a local centroid among its assigned vertices. A global centroid is then calculated by adding the centroids from each processor, which is then communicated to all processors. Specifically, all but the k worst vertices are involved in computing the centroid, which are spread across the p processors. Each of the p processors computes the centroid of its $(n-k)/p$ vertices. Denote a local centroid for processor j as $M_j = \frac{1}{n}\sum_{i=(n-k)/p}^{n/p} A_i$. The global centroid is then $M = \sum_{j=1}^{p} M_j$. In this process, all processors call `MPI_Allreduce` to sum all M_j to obtain M and distribute it to all processors.

Reflection, Expansion, Contraction: All of these computations involve only the global centroid, the vertex being updated, and the value of the objective function at the global best vertex. The global centroid and the vertex being updated are already available on each processor. The value of the best vertex is distributed to all processors before each processors updates its k/p vertices as follows. All processors know the minimum of their n/p vertices, denote it $f(A_0^j)$. If A_0 is the global minimum it must be that $f(A_0) = \min\left(f\left(A_0^1\right), f\left(A_0^2\right), \ldots, f\left(A_0^p\right)\right)$. Thus, all processors call `MPI_AllReduce` to determine which $f(A_0^j)$ is the minimum, and distribute the value $f(A_0)$ to all processors.

Shrink: Each of the p processors keeps a parameter that equals 1 if a vertex in the simplex has been updated, and 0 otherwise. All processors then call `MPI_AllReduce` to find the sum of this parameter across all processors. If the sum is not zero, a shrink is not necessary because some processor updated a vertex and the next iteration may start. If the sum is zero, then each processor needs A_0 in order shrink the simplex. The value of A_0 is distributed to all processors using `MPI_Bcast`. First, however, the processor containing A_0 must first find out that it contains the best vertex, and all

[3] While it is possible to accomplish our entire algorithm using only `MPI_AllReduce`, it is simpler to use `MPI_Bcast` in the shrink step.

other processors must know which processor contains A_0 to know who to receive from. This can be done with a slightly modified version of the `MPI_AllReduce` which also identifies the processor which A_0 came from. Once A_0 has been distributed, the processors perform the shrink in parallel.

4 Experiments

All code was written in C++, compiled with the `mpicxx` command, and run on the Triton cluster in the San Diego SuperComputer Center. Triton uses the pgCC 10.5-0 compiler. The executions were timed with the MPI command `MPI_Wtime`; we ignored the initial time it takes to allocate the needed memory. Each experiment was repeated five times and time-data was averaged. We pick a convergence criterion of 10^{-6}.

We divide our experiments into two categories. First, we examine the effects of updating multiple vertices per iteration on a single processor. We document that the speedups that can be obtained are much larger than previously thought, resulting in fewer iterations and faster convergence. Second, we examine the scalability of our distributed memory implementation, and demonstrate that the communication overhead between processors is minimal. Furthermore, updating local vertices has little impact on the number of iterations needed to converge to a solution.

We consider the two objective functions considered by Lee and Wiswall (2007), denoted by f_1 and f_2. Given an n dimensional vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, we minimize the following functions:

$$f_1(\mathbf{x}) = \sum_{i=1}^{n} \frac{x_i^2}{n}$$

$$f_2(\mathbf{x}) = \sum_{i=1}^{n} \frac{|x_i|}{n}.$$

Notice that the solution to both minimization problems is the zero vector.

4.1 Single Processor

The first experiment with a single processor studies the convergence rate for different values of k. Specifically, for $n = 1{,}000$ we progressively increase k from 1 to 750 and plot the best value of the objective function versus the total number of vertices updated. Figure 3a shows the results for f_1 and Fig. 3b shows those for f_2. The results show that the speed of convergence increases with k. Furthermore, the rate of this increase is also increasing in k. However, we do notice that this is only effective to a k of size about half of n.

The second experiment with a single processor compares the speedup in execution time needed to reach the convergence criterion as the number of vertices updated in a single iteration is increased. The results of this experiment are plotted in Fig. 4a.

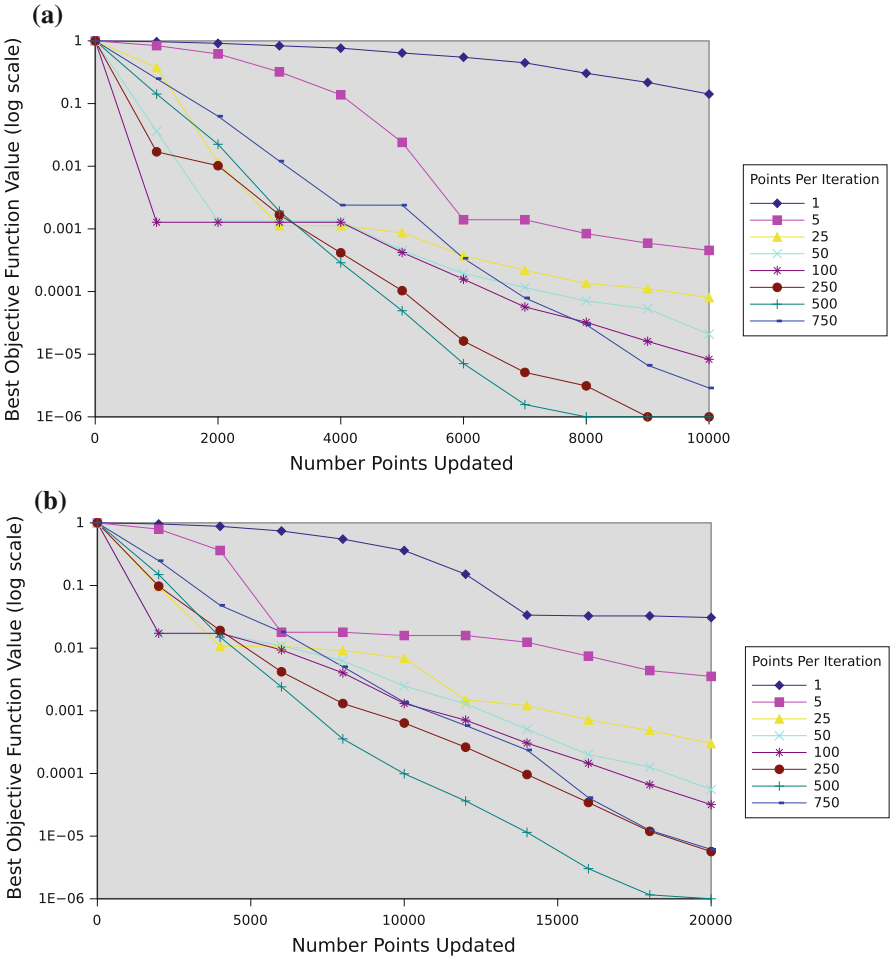

Fig. 3 Plots of the number of vertices updated versus the current best value of the objective function. Each *line* denotes a different number of vertices updated in a single iteration. f_1 is depicted in (a) and f_2 in (b).

Figure 4b contains the results of our third experiment, which looks at the total number of vertices updated before the functions converge.[4]

Figure 4 shows that for both objective functions there is an increase in both, the number of vertices updated and the execution time as we move from $k = 500$ to $k = 750$. Compared to updating one vertex per iteration, the speedups achieved were over 200 times for f_1 and nearly 3,000 times for f_2 when $k = 500$. This is due to two reasons. First, Fig. 4(b) shows that the number of vertices updated before convergence decreases drastically with k. In particular, with one vertex updated per iteration, f_1 and f_2 updated around 66,000 vertices and 228,000 before convergence,

[4] It should be made clear that if the k worst vertices are updated each iteration, we count that as updating k vertices. Similarly if 1 vertex is updated each iteration, it would take k iterations to update k vertices.

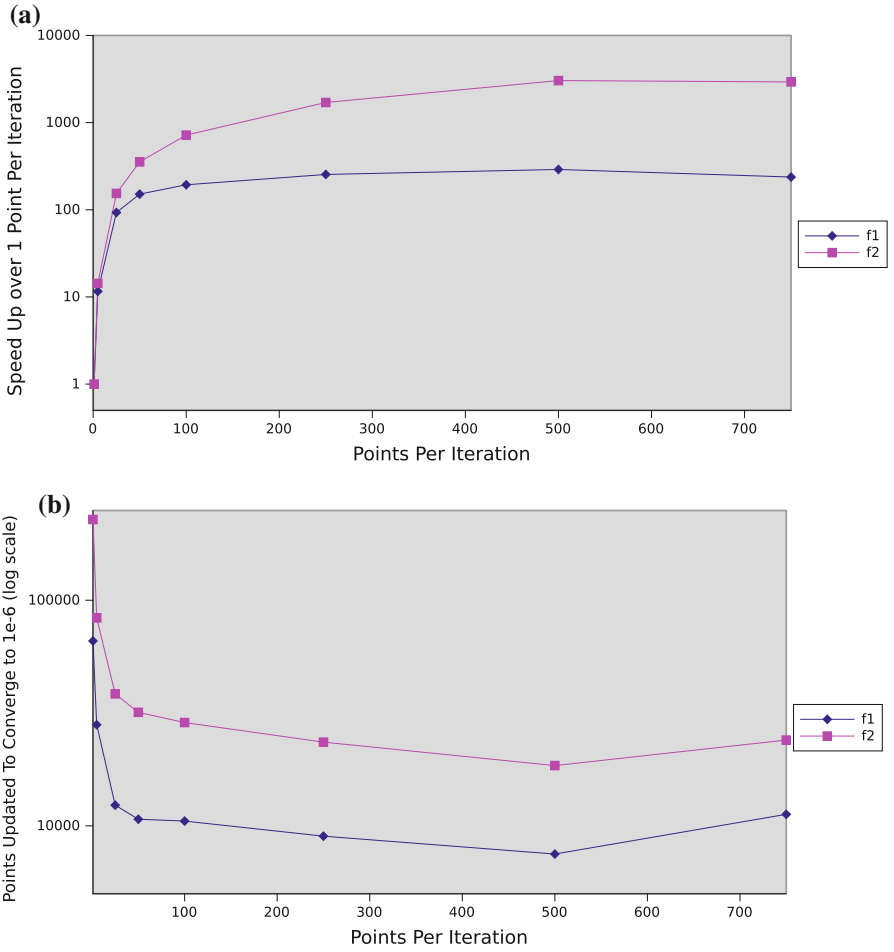

Fig. 4 (a) Shows the speedup obtained by increasing the number of vertices updated in a single iteration and (b) shows the total number of vertices that were updated before convergence

respectively. With 500 vertices updated per iteration, each function updated around 7,500 and 18,500 vertices before convergence, respectively. Since only the best vertices are used to compute the centroid, the search direction of each updated vertex is likely better with multiple vertices updated per iteration. However, the number points updated decreased at most by a factor of 12. The remainder of the speedup comes from the fact that, when $k = 500$, the number of times the centroid is computed and sorted is reduced by a factor of 500 times.[5] This explains the enormous speedups, as well as the larger speedup achieve by f_2, as f_2 had a larger reduction in the number of required iterations.

[5] These operations are $O(n^2)$ and $O(n \log n)$, respectively, as opposed to the remaining $O(n)$ cost of updating a single vertex.

4.2 Multiple Processors

The first experiment we perform on the distributed memory implementation is commonly called *strong scaling*. Strong scaling means that for a fixed problem size and fixed number of vertices updated per iteration, $n = 10{,}000$ and $k = 5{,}000$, we scale the number of processors from $p = 1$ to $p = 32$. *Parallel efficiency* is defined as $T^*/(p * T_p)$, where T^* is the time taken to reach the convergence criterion on a single processor and T_p is the time taken on p processors. Figure 5a depicts the parallel efficiency of our distributed memory implementation as we scale the number of processors.

Next, we scale the problem with the number of processors such that each processor gets the same amount of work. This is commonly called *weak scaling*. The expectation is that the execution time should remain roughly the same, as each processor has the same amount of work to do. The first thing to note is that as the problem size increases, the amount of iterations required to converge will increase. Thus, we measure the amount of time it takes to execute a fixed number of iterations. Next, we scale the problem size appropriately to keep the amount of memory used by each processor the same.[6] The results of scaling to 32 processors are shown in Fig. 5b.

A potential loss of efficiency of our implementation is that we update the worst k/p vertices on each processor, instead of the k worst global vertices. In order to understand the magnitude of this loss of efficiency, we compare the number of iterations required for convergence on a single processor to the number required when we parallelize the problem. Specifically, for a problem of size $n = 10{,}000$ we update $k = 5{,}000$ vertices per iteration and look at the number of iterations required to converge on varying numbers of processors. Notice, that the single processor will always update the k worst vertices. Thus, the number of iterations taken by the single processor equals the number of iterations the parallel implementation would take when the k worst global vertices are updated.

Figure 6a shows that not updating the k worst global vertices has little effect on convergence. For example, f_1 converges with fewer vertices updated in every case but where $n = 8$, and then it still only updates about 10 % more vertices. On the other hand, f_2 updates more vertices almost every time, but in all cases but one about 10 % extra vertices are updated, again with the exceptional case when $n = 8$ where 23 % extra vertices are updated. Overall, this is a small cost to pay relative to the possibly large amounts of communication overhead cost to distribute k/p of the worst vertices to each processor, in addition to the cost of the distributed sort to find them.

The last experiment assesses the communication overhead in the distributed memory implementation. A convenient toolkit to measure communication overhead

[6] Notice that the memory required for a problem size of n is $O(n^2)$. Hence, to keep the amount of memory used by each processor the same, if we have p processors we scale the problem size by a factor of $\sqrt{p}$. However, with exception to the infrequent sorting and centroid computations, the work done by each processor is $O(n)$. Since we only increase the problem size by a factor of $\sqrt{p}$, we also increase the number of iterations by a factor of $\sqrt{p}$ to equalize the work performed. Specifically, for a single processor we execute 32, 000 iterations on a problem of size $n = 4{,}000$ and update $k = n/4$ vertices per iteration. When we scale to $p = 2$, we then execute $32{,}000 \cdot \sqrt{2}$ iterations on a problem of size $n \cdot \sqrt{2}$ and update $k = (n\sqrt{2})/(4)$ vertices per iteration (the vertices updated per iteration are split amongst the processors).

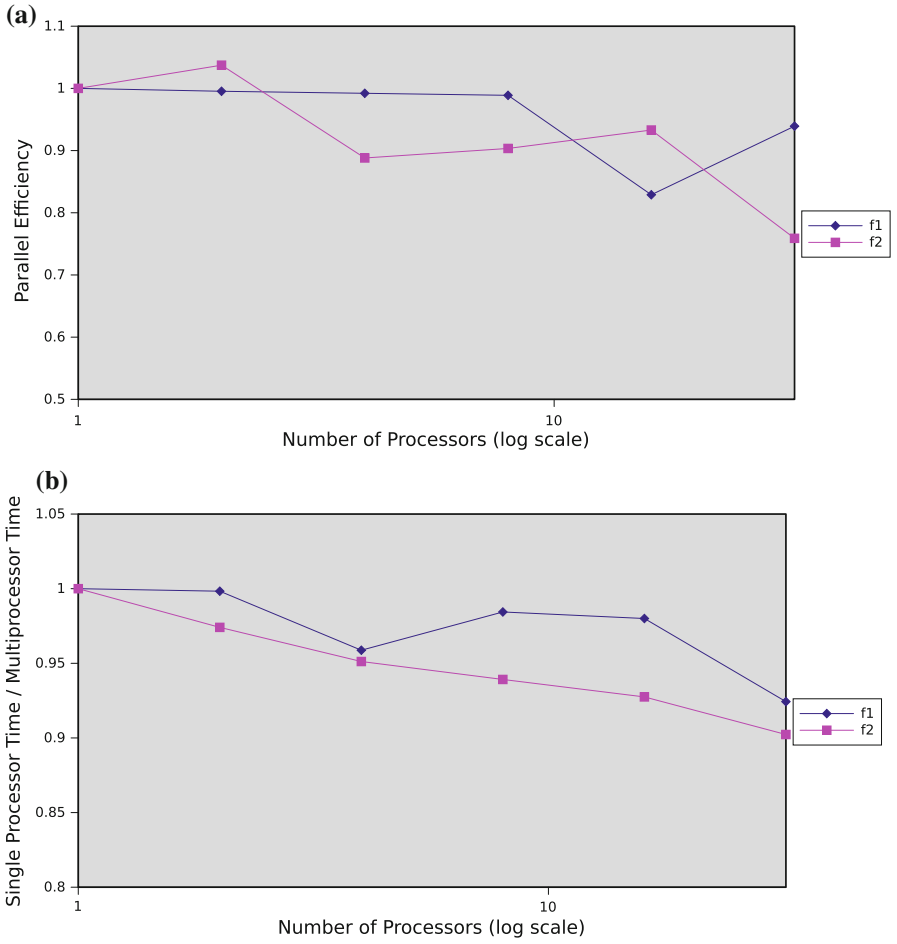

Fig. 5 **a** Shows the results of the *strong* scaling analysis and **b** shows the results of the *weak* scaling analysis

is TAU profiling. Specifically, TAU determines how much time each processor spends on each function on a given run. We run the same problem sizes as in the weak scaling experiments except we remove the maximum iteration cap and instead wait for the algorithm to converge. We then use the profiling information generated by TAU to see what percentage of time the code spends in MPI routines, as all other code is part of the serial implementation. These results are shown in Fig. 6b. Again, we run each experiment five times and report the average.

The strong scaling results show a steady parallel efficiency over 90 % until the number of processors reaches around 32, at which point we see a drop in f_2 to around 80 %. The dips in efficiency are probably due to the result of the problem size becoming too small relative to the number of processors. This conclusion is supported by the results of the weak scaling, where we see that increasing the time to solve a large problem on 32 processors is within 90 % of the time needed to solve the problem on

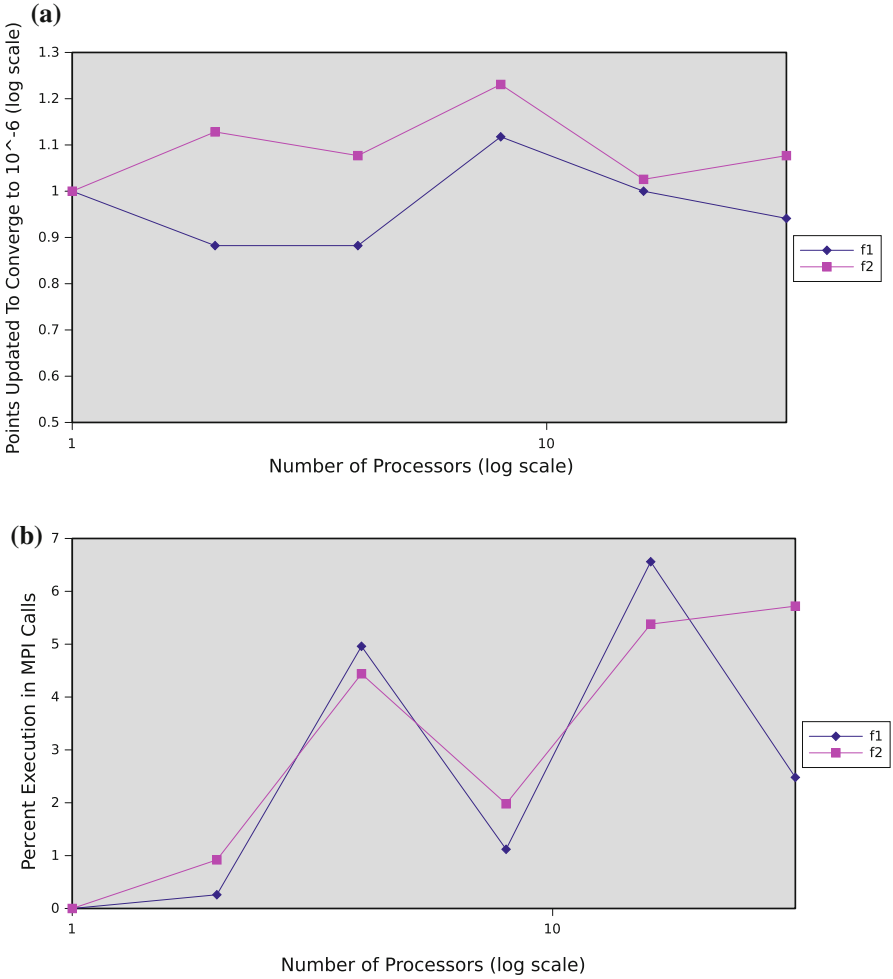

Fig. 6 **a** Shows the change in the number of vertices updated by our distributed memory implementation versus an implementation which updates the worst global k vertices, and **b** shows the overhead of communication obtained by profiling with TAU

a single processor. This result implies that the algorithm can be scaled to a number of processors much larger than 32 while maintaining high parallel efficiency, allowing much larger problems to be solved. The scalability of the algorithm is further confirmed by the profiling performed by `TAU`. Figure 6b shows that as the number of processors is increased to 32 the percentage of time spent communicating for both objective functions increases to only 7 %. Notice this is consistent with the results of the weak scaling analysis. This communication time is nearly exclusively from `MPI_AllReduce`, as the need to shrink the simplex is rare when updating so many vertices.

5 Conclusion

In this paper we described a method to implement the Nelder-Mead simplex method using a distributed memory implementation. For various examples we documented how our implementation exhibits large speedups and is scalable to large problem sizes. Our hope is that previously unfeasible large calibration and structural estimation problems become within computational reach without requiring the author to rewrite computer code.

References

Aldrich, E. M., Fernandez-Villaverde, J., Ronald Gallant, A., & Rubio-Ramirez, J. F. (2011). Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors. *Journal of Economic Dynamics and Control*, *35*, 386–393.

Beaumont, P. M., & Bradshaw, P. M. (1995). A distributed parallel genetic algorithm for solving optimal growth models. *Computational Economics*, *8*, 159–179.

Creel, M. (2005). User-friendly parallel computations with econometric examples. *Computational Economics*, *26*, 107–128.

Dennis, J. E, Jr, & Torczon, V. (1991). Direct search methods on parallel machines. *SIAM Journal on Optimization*, *1*, 448–474.

Ferrall, C. (2005). Solving finite mixture models: Efficient computation in economics under serial and parallel execution. *Computational Economics*, *25*, 343–379.

Huggett, M., Ventura, G., & Yaron, A. (2006). Human capital and earnings distribution dynamics. *Journal of Monetary Economics*, *53*, 265–290.

Lawver, D. (2012). *Measuring quality increases in the medical sector*. Santa Barbara: University of California Santa Barbara.

Lee, D., & Wiswall, M. (2007). A parallel implementation of the simplex function minimization routine. *Computational Economics*, *30*, 171–187.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, *7*, 308–313.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical recipes: The art of scientific computing* (3rd ed.). Cambridge University Press.

Swann, C. A. (2002). Maximum likelihood estimation using parallel computing: An introduction to MPI. *Computational Economics*, *19*, 145–178.