Comput Econ (2007) 30:171–187
DOI 10.1007/s10614-007-9094-2

A Parallel Implementation of the Simplex Function Minimization Routine

Donghoon Lee · Matthew Wiswall

Accepted: 15 May 2007 / Published online: 26 June 2007

Abstract This paper generalizes the widely used Nelder and Mead (Comput J 7:308–313, 1965) simplex algorithm to parallel processors. Unlike most previous parallelization methods, which are based on parallelizing the tasks required to compute a specific objective function given a vector of parameters, our parallel simplex algorithm uses parallelization at the parameter level. Our parallel simplex algorithm assigns to each processor a separate vector of parameters corresponding to a point on a simplex. The processors then conduct the simplex search steps for an improved point, communicate the results, and a new simplex is formed. The advantage of this method is that our algorithm is generic and can be applied, without re-writing computer code, to any optimization problem which the non-parallel Nelder–Mead is applicable. The method is also easily scalable to any degree of parallelization up to the number of parameters. In a series of Monte Carlo experiments, we show that this parallel simplex method yields computational savings in some experiments up to three times the number of processors.

Keywords Parallel computing · Optimization algorithms

1 Introduction

We consider the computation of econometric estimators given by the optimization problem: $min_\theta \; f(\theta)$, where $\theta \in \Theta$ is a J dimensional vector of parameters. For optimization problems without closed form solutions, an iterative procedure is typ-

D. Lee · M. Wiswall (✉)
Department of Economics, New York University, 269 Mercer St., 7FL New York, NY 10003, USA
e-mail: mwiswall@nyu.edu

D. Lee
e-mail: donghoon.lee@nyu.edu

ically used. These algorithms successively evaluate the objective function $f(\theta)$ at different trial parameter vectors until a parameter vector achieves a convergence criterion. There are two main sources of computation time in these numerical procedures. First, for each trial vector of parameters, there is the computational cost of evaluating the objective function at this vector of parameters. Where the objective function involves simulation or the solution of a complex behavioral model, such as a dynamic programming model, the objective function level computation costs can be large. A second source of computational costs is that for a high dimensional and continuous parameter space Θ, a large number of vectors of parameters may need to be tried before the convergence criterion is achieved. This parameter level computation cost is increasing in the number of parameters J, which is generally related to the complexity of the underlying behavioral model.

Parallel computing techniques in economics have generally focused on using parallelization to decrease the first source of computation costs. For each set of candidate parameters, these objective function parallelization methods generally involve breaking down the objective function into several independent computation tasks. Each of the processors completes one of these tasks and communicates the results to the other processors. These results are then used to choose the next candidate set of parameters. Several common objective functions allow this type of parallelization. Swann (2002) describes the parallelization of a maximum likelihood objective functions where separate processors compute the likelihood function for subsets of sample observations, and these results are summed to calculate the full sample likelihood function. Creel (2005) describes a similar objective function parallelization for generalized method of moments objective functions where each processor computes moments for subsets of observations. Other possibilities for parallelization at the objective function level are based on the specific structure of the underlying model. For example, if the objective function is based on solving a behavioral model consisting of a finite number of agent types (finite mixtures models), the objective function could be broken down by agent type, and independent processors could be used to solve the model for each agent type (e.g. Ferrall 2005). The objective function is computed by combining the results for each type. Todd and Wolpin (2006) and van der Klaauw and Wolpin (2006) use this approach in their empirical applications.

In this paper, we develop an alternative parallel optimization algorithm designed to limit the second source of computation cost, parameter level computation cost. Instead of breaking down the objective function into several independent tasks, we keep the objective function whole and parallelize the optimization problem at the parameter level. We develop a generalization of the Nelder and Mead (1965) simplex algorithm for parallel processors. Our parallel Nelder–Mead (NM) simplex algorithm does not break the objective function into parts, but instead in each iteration, a separate vector of parameters is assigned to each processor corresponding to a point on a simplex. The processors then conduct the simplex search steps for improved points, communicate the results, and a new simplex is formed. This algorithm continues until a convergence criterion is met. In a series of numerical exercises, we show that this parallel version of the NM simplex method achieves substantial gains in computation time relative to the non-parallel simplex method. In some of our examples, these gains are up to three times the number of processors.

Simple parallelization at the parameter level could be implemented with several types of numerical optimization algorithms. For example, a grid search algorithm could make use of parallel processing by assigning each processor a subset of the parameter space to evaluate. In addition, other numerical algorithms could use parallel processors by having each processor independently solve the optimization problem using a different set of starting parameters. While these simple parallelization methods may achieve some computational gains over single processor serial approaches, algorithms specifically designed to make use of parallel processing could exhibit larger gains. Beaumont and Bradshaw (1995), for example, develop a parallel genetic optimization algorithm, which uses each processor as an "island" to introduce greater "genetic" diversity in the stochastic search algorithm. Dennis and Torczo (1991) develop a parallel simplex method in which each processor searches a grid of points along the reflection point of each of the initial simplex vertices. This method is intended for optimization problems with small numbers of parameters (small J).

A considerable body of research exists in the computer science fields on parallel simplex algorithms for linear programming problems (e.g. Barr and Hickman 1994; Bixby and Martin 2000; Klabjan et al. 2000). Our approach is intended for unconstrained problems, which is more typical of computation in econometrics (e.g. maximum likelihood and generalized method of moments estimators). In addition, much of this prior literature discusses parallelization within particular parallel computing hardware structures.[1] Although there may be scope to adapt our algorithm to specific hardware structures, the analysis of our algorithm ignores these hardware issues of communication across processors and data storage because in our algorithm, processors only communicate a handful of scalar values, and communicate this information only once after a relatively small number of iterations. This is not the case in general for objective function parallelization where processors often communicate much larger objects more frequently.

The main advantage of parallelization at the parameter level is that the objective function does not need to be broken down into independent tasks in order to take advantage of parallel computing. Like its non-parallel counterpart, our parallel NM algorithm is a generic procedure which does not depend on the specific objective function or the number of available processors. This advantage is two-fold. First, for objective functions which cannot be broken down into independent tasks, and therefore parallelization is not possible at the objective function level, the parallel NM algorithm is still applicable. For example, objective functions based on models of interacting agents may not allow the objective function to be divided into separate tasks. In Lee and Wolpin's (2006) general equilibrium labor market model, individuals interact through equilibrium skill prices. Implementation of objective function parallelization for these types of models is not readily obvious. Even if some parallelization of the objective function is possible, it may not be possible to break down the objective function into enough independent tasks to fully use all of the available processors. This is the case, for example, if the objective function is broken down by agent types, but the researcher has more processors available than agent types.

[1] For an introduction to parallel programming and hardware in general, see Swann 2002.

Second, even in cases where parallelization at the objective function level is possible, our parallel NM simplex algorithm avoids the need to re-write computer code for each version of a model and objective function or as more processors become available. The benefit of not re-writing computer code should not be underestimated. This may be particularly important in the context of parallel methods as the difficulty of writing parallel code may explain its relative lack of use among economists.

It should be noted that our parallelization at the parameter level can be combined with parallelization at the objective function level. The two types of parallelization can be nested in which a set of available processors can be divided into groups of several processors. Each group of processors is given a vector of parameters as dictated by the parallel simplex algorithm, and the objective function evaluation tasks are further divided among each processor within the group. Given a fixed number of processors available, researchers will need to make a case-by-case decision about how to allocate their computation resources across the two types of parallelization. In general, the advantage to parameter level parallelization over objective function parallelization is increasing in the size of the parameter space Θ.

We argue that as economists attempt to estimate ever more complex behavioral models involving ever more parameters, parallelization at the parameter level will increasingly become a source of computation time savings. Economists have developed a number of methods to limit the computation time involved in solving some types of behavioral models, as for example in solving dynamic programming models (e.g. Hotz and Miller 1993; Keane and Wolpin 1994). These methods decrease the computation cost associated with evaluating the objective function at a given set of parameters. However, other than intuition about good potential starting parameters, there is little opportunity for researchers to reduce computational costs at the parameter level outside of using greater computational resources and algorithms designed to use these resources.

This paper is organized as follows. Section 2 describes the non-parallel Nelder–Mead (NM) simplex method. Section 3 describes our parallel version of the NM simplex method. Section 4 provides a series of Monte Carlo experiments in which we compare the performance of the non-parallel and parallel simplex methods using various numbers of processors and numbers of parameters. Section 5 discusses how some of the potential advantages of our parallel NM algorithm. Section 6 concludes.

2 Non-Parallel Nelder–Mead Simplex

As a baseline to compare our parallel simplex algorithm, we first describe the original Nelder–Mead (1965) simplex method.[2] The algorithm computes the solution to the problem $\min_{\theta \in \Theta} f(\theta)$, where θ is a J dimensional vector.

[2] The NM simplex can be found in the Fortran IMSL subroutines UMPOL (for single precision) and DUMPOL (for double precision). The source code can be obtained from Allan J. Miller's software homepage at http://users.bigpond.net.au/amiller/minim.f90. Other packages, such as Matlab's fminsearch or fmins command, have somewhat different simplex algorithms than described here.

Step 1: Create the Initial Simplex.

The researcher first chooses a J dimensional vector of starting parameters: $A_0 = [\theta_1, \theta_2, \ldots, \theta_J]$. The NM algorithm forms an initial simplex of $J + 1$ points, $A_0, A_1, \ldots, A_J$.

$$A_0 = [\theta_1, \theta_2, \ldots, \theta_J]$$

$$A_1 = [\theta_1 + s_1, \theta_2, \ldots, \theta_J]$$

$$A_2 = [\theta_1, \theta_2 + s_2, \ldots, \theta_J]$$

$$\vdots$$

$$A_J = [\theta_1, \theta_2, \ldots, \theta_J + s_J],$$

where $s_1, s_2, \ldots, s_J$ are the initial step sizes.

The NM simplex calculates the objective function at each of the $J + 1$ simplex points: $f(A_0), f(A_1), \ldots, f(A_J)$. Without loss of generality, re-order the points on the initial simplex as

$$f(A_0) < f(A_1) < \cdots < f(A_J).$$

Step 2: Calculate the Reflection Point

The NM algorithm replaces the worst point on the simplex A_J with another point which has a lower objective function evaluation. For ease of exposition, we say that the point A_j is an "improvement" over or "better" than the point A_k if $f(A_j) < f(A_k)$. The first candidate improvement point is the reflection point. The NM simplex calculates the reflection point A_J^R as the reflection of the worst point, A_J, through the centroid M of the remaining points, $A_0, A_1, \ldots, A_{J-1}$. The centroid is

$$M = \frac{1}{J} \sum_{j=0}^{J-1} A_j$$

The reflection point is then

$$A_J^R = M + \alpha(M - A_J),$$

where $\alpha > 0$ is an algorithm parameter. The default value is typically $\alpha = 1$.

Step 3: Update the Simplex

We next evaluate the objective function at the reflection point to obtain $f(A_J^R)$. There are three cases depending on the objective function value at the reflection point relative to the previously calculated values of the objective function at the other points on the existing simplex.

Case 1: If A_J^R is an improvement over the initial best point A_0, then we continue to move in the same direction by calculating the expansion point A_J^E

$$A_J^E = A_J^R + \gamma(A_J^R - M),$$

where $\gamma > 0$ is an algorithm parameter, typically $\gamma = 1$.

If A_J^E is an improvement over A_0, then A_J is replaced by A_J^E. The new simplex, including the expansion point, is re-ordered, and we return to Step 2. If A_J^E is not an improvement over A_0, then A_J is replaced by A_J^R, and we re-order the simplex and return to Step 2.

Case 2: If A_J^R is not an improvement over A_0, and A_J^R is better than the next worst point A_{J-1}, then A_J is replaced by A_J^R, and we re-order the simplex and return to Step 2.

Case 3: If A_J^R is not an improvement over A_0, and worse than the next worst point A_{J-1}, then we calculate the contraction point A_J^C as

$$A_J^C = M + \beta(\widetilde{A_J} - M),$$

where $\widetilde{A}_J = A_J^R$ if $f(A_J^R) < f(A_J)$, and $\widetilde{A}_J = A_J$ otherwise. $0 < \beta < 1$ is an algorithm parameter, typically $\beta = 1/2$.

If A_J^C is an improvement over A_J, then A_J is replaced by A_J^C, and we re-order the simplex and return to Step 2. If A_J^C is not an improvement over A_J, then we shrink the entire simplex toward the best point A_0. The new simplex is defined by these $J + 1$ points

$$[A_0, (\tau A_0 + (1-\tau)A_1), (\tau A_0 + (1-\tau)A_2), \ldots, (\tau A_0 + (1-\tau)A_{J-1}),$$
$$(\tau A_0 + (1-\tau)\widetilde{A}_J)],$$

where $0 < \tau < 1$ is an algorithm parameter, typically $\tau = 1/2$. Using this new simplex, we return to Step 2.

3 Parallel Simplex

We now describe our parallel version of the Nelder–Mead simplex algorithm. The degree of parallelization is P. The degree of parallelization need not equal the number of available processors. The parallel algorithm we describe here can be implemented using a single processor. For a single processor using P degrees of parallelization, each task assigned to a separate processor would instead be sequentially completed by a single processor. However, we call each unit performing each separate operation a "processor" to make clear how tasks can be divided among independent processors, if they are available.

Step 1: Create the Initial Simplex.

Step 1 for the parallel case is identical to Step 1 for the non-parallel NM simplex algorithm. We create a $J + 1$ size simplex using the step sizes $s_1, s_2, \ldots, s_J$, evaluate the objective function at each point, and order the simplex points from best to worst: $f(A_0) < f(A_1) < \cdots < f(A_J)$.

Step 2: Assign Each Processor One of the P Worst Points

We assume that the number of parameters is at least as great as the degree of parallelization, $J \geq P$. The parallel algorithm reflects the P worst points (A_{J-P+1}, A_{J-P+2}, ..., A_J) through the centroid M of the remaining points (A_0, $A_1, \ldots, A_{J-P}$). The centroid M is defined as

$$M = \frac{1}{J}\sum_{j=0}^{J-P} A_j.$$

Each processor is assigned one of the P worst points on the simplex. The P worst points are denoted A_j for $j = J - P + 1, J - P + 2, \ldots, J$. Each processor then calculates the reflection point for their assigned point as

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

The set of P processors then contains a set of P reflection points:

$$\begin{aligned} A_{J-P+1}^R &= M + \alpha(M - A_{J-P+1}) \\ &\vdots \\ A_J^R &= M + \alpha(M - A_J), \end{aligned}$$

Note that with one degree of parallelization ($P = 1$), this procedure collapses to the original non-parallel NM simplex approach as only the worst point A_J is reflected.

Step 3: Each Processor Returns an Updated Point

Each of the processors evaluates the objective function at their assigned reflection point A_j^R. Each processor then follows a set of instructions analogous to the non-parallel simplex algorithm. Let A_j^* for $j = J - P + 1, J - P + 2, \ldots, J$ denote the point each processor returns at the end of Step 3. Analogous to the non-parallel simplex, each updated point is from the set $\{A_j, A_j^R, A_j^E, A_j^C\}$, where A_j is one of the worst P points on the original simplex, and A_j^R, A_j^E, and A_j^C are the reflection, expansion, and contraction points, respectively.

The specific set of instructions for each processor are as follows:

Case 1: If the calculated reflection point A_j^R is an improvement over the initial best point A_0, then we continue to move in the same direction by calculating the expansion point A_j^E

$$A_j^E = A_j^R + \gamma(A_j^R - M),$$

If A_j^E is an improvement over A_0, then the processor returns the expansion point $A_j^* = A_j^E$. If A_j^E is not an improvement over A_0, then the processor returns the reflection point $A_j^* = A_j^R$.

Case 2: If A_j^R is not an improvement over A_0, but A_j^R is better than the next worse point A_{j-1}, then the processor returns $A_j^* = A_j^R$. For the discussion of computation time below, notice that in this case, each processor is only evaluating the objective function once, rather than twice as in Cases 1 and 3. For processors for which Case 2 applies, they then remain idle if Cases 1 or 3 apply to any of the other processors.

Case 3: If A_j^R is worse than the next worst point A_{j-1}, then the processor calculates the contraction point A_j^C as

$$A_j^C = \beta(M + \widetilde{A}_j),$$

where $\widetilde{A}_j = A_j^R$ if $f(A_j^R) < f(A_j)$, and $\widetilde{A}_j = A_j$ otherwise.

If A_j^C is an improvement over $\widetilde{A}_j$, then the processor returns the contraction point $A_j^* = A_j^C$. If the contraction point is not an improvement, then the processor returns $A_j^* = \widetilde{A}_j$, which is either A_j^R or A_j.

Step 4: Form the New Simplex

The P processors return P points, all of which could be new points as determined above. If for any processor, Case 1 or 2 apply or if Case 3 applies and $A_j^* = A_j^C$, then the new $J+1$ dimension simplex is formed from the P points $A_{J-P+1}^*, A_{J-P+2}^*, \ldots, A_J^*$ defined above and the original best points $A_0, A_1, \ldots, A_{J-P}$. The new $J+1$ dimension simplex is then

$$[A_0, A_1, \ldots, A_{J-P}, A_{J-P+1}^*, A_{J-P+2}^*, \ldots, A_J^*].$$

On the other hand, if Case 1, Case 2, or Case 3 and $A_j^* = A_j^C$ do not apply to any of the processors, then the entire simplex shrinks toward the best point A_0. The new simplex is defined by these $J+1$ points:

$$[A_0, (\tau A_0 + (1-\tau)A_1), (\tau A_0 + (1-\tau)A_2), \ldots, (\tau A_0 + (1-\tau)\widetilde{A}_{J-P+1}), \ldots, \\ (\tau A_0 + (1-\tau)\widetilde{A}_{J-1}), (\tau A_0 + (1-\tau)\widetilde{A}_J)].$$

After the new simplex is formed, we re-order the points in the new simplex by their objective function values and return to Step 2.

4 Monte Carlo Evidence

In the following Monte Carlo experiments, we examine the performance of the parallel simplex method for three different optimization problems. The optimization problems we consider have a large number of parameters: $\dim(\Theta) = J \geq 100$, which are intended to mimic economic problems with high dimensional parameter spaces. However, aside from the high dimension of the parameter space, the objective functions are relatively simple in order to make a large number of Monte Carlo trials computationally feasible.

For each optimization problem, we compare the performance of the algorithm across different degrees of parallelization from the non-parallel NM simplex case ($P = 1$) to higher degrees of parallelization ($P > 1$). We draw 100 random initial vectors of starting parameters, each of which is independently and identically distributed standard normal. For each of the 100 starting vectors, we use the parallel simplex algorithm to solve the optimization problem. The performance results are averages over the 100 different starting parameters. To minimize simulation noise, we use the same 100 starting vectors across all degrees of parallelization.[3]

We measure computation time as the number of iterations required to reach a particular objective function value ($f(\theta) = 0.1$ for the objective functions considered below). A single iteration of the parallel simplex algorithm with P degrees of parallelization consists of a set of up to P objective function evaluations. For the non-parallel simplex algorithm ($P = 1$), each iteration is one objective function evaluation. The computational savings of our parallel algorithm is that each iteration for the parallel simplex can involve up to P simultaneous objective function evaluations, which takes about the same clock time as a single objective function evaluation. For the parallel simplex algorithm with degree of parallelization P, every other iteration after the formation of the initial simplex involves P objective function evaluations at the assigned P reflection points. The next iteration involves between 1 and P objective function iterations as some processors are idle after calculating the objective function at their assigned reflection point (Case 2), while other processors are evaluating the objective function at expansion (Case 1) or contraction points (Case 3).[4] The idle processors with $P > 1$ is a potential drag on the computational savings from the parallel simplex algorithm.

The cost of communication time among processors is typically a factor in determining the total computational savings from parallel processing algorithms. We ignore communication time in our algorithm because processors only communicate a handful of scalar objective function values, and communicate this information only once after each iteration. For optimization problems in which an econometric estimator is formed from the solution to an often complex behavioral model, the communication time in our algorithm is trivial relative to the time spent evaluating the objective function at a given set of parameters.

[3] For both the parallel and non-parallel algorithms, we use the following algorithm parameters: step sizes $s_j = 1$ for all j, $\alpha = 1$, $\gamma = 1$, $\beta = 1/2$, and $\tau = 1/2$. Re-starting the simplex when it becomes small seems to offer an important improvement. For all experiments, when the difference between the maximum point and the minimum point on the simplex is less than 0.001, we re-start simplex by forming a new initial simplex using the best point on the current simplex.

[4] Note also that the shrink procedure fully uses all P processors, as each processor evaluates a new simplex point.

Table 1 (Example 1): Mean function value by degree of parallelization (P)

	Function evaluations						
P	0	500	1000	1500	2000	3000	5000
1	0.938	0.510	0.314	0.184	0.102	0.034	0.011
2	0.938	0.459	0.308	0.178	0.101	0.036	0.013
3	0.938	0.427	0.295	0.172	0.096	0.036	0.013
5	0.938	0.367	0.251	0.134	0.078	0.032	0.012
10	0.938	0.290	0.202	0.104	0.065	0.029	0.010
20	0.938	0.249	0.170	0.091	0.062	0.032	0.013
30	0.938	0.237	0.169	0.094	0.065	0.036	0.015
50	0.938	0.178	0.087	0.048	0.034	0.022	0.010
80	0.938	0.216	0.066	0.035	0.027	0.020	0.008
90	0.938	0.322	0.191	0.156	0.149	0.133	0.053

Notes: P is degree of parallelization. Results from Example 1 minimizing the objective function $\sum_{j=1}^{100}\theta_j^2/100$. Function values are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

Example 1

In this example, we consider minimizing the following objective function

$$f(\theta)=\sum_{j=1}^{100}\theta_j^2/100,$$

where θ is a vector of $J = 100$ parameters. The true minimum of this function is $\theta_j = 0$ for all j.

Table 1 reports the function value after 0 through 5,000 function evaluations for various degrees of parallelization (P). We count function evaluations following the formation of the initial simplex. For each degree of parallelization, the objective function value at 0 is $f(\theta) = 0.938$ because the same starting parameters are used and the initial simplex is invariant to the degree of parallelization. For the single processor case ($P = 1$), the function value falls to $f(\theta) = 0.51$ after 500 function evaluations, and to $f(\theta) = 0.01$ after 5,000 function evaluations. For higher degrees of parallelization from $P = 2$ to $P = 80$, there is steeper drop in the objective function value over the first several thousand function evaluations. This indicates a higher level of performance in reducing the objective function as the degree of parallelization increases. However, with $P = 90$ degrees of parallelization, the performance of the algorithm is noticeably declining.

Table 2 provides our direct measure of the computation time to reach a minimum objective function value of $f(\theta) = 0.1$. The first columns of Table 2 measure the mean total number of function evaluations required to reach $f(\theta) = 0.1$. For example, for the non-parallel case ($P = 1$), Table 2 indicates that 2,011 objective function evaluations are required to reach $f(\theta) = 0.1$. The Gain column measures the extent to

Table 2 (Example 1): Computation required to reach $f(\theta) = 0.1$

P	Mean funct. evals.	Gain	Mean iterations	Gain
1	2011	1	2011	1
2	2010	1.00	1100	1.83
3	1978	1.02	763	2.64
5	1729	1.16	422	4.77
10	1542	1.30	200	10.0
20	1430	1.41	96.9	20.8
30	1463	1.38	67.0	30.0
50	887	2.27	23.2	86.6
80	795	2.53	12.6	160
90	3521	0.571	48.7	41.3

Notes: P is degree of parallelization. Gain is the multiple increase over $P = 1$, e.g. Gain of 2 indicates that this degree of parallelization is twice as productive as $P = 1$. Results from Example 1 minimizing the objective function $\sum_{j=1}^{100} \theta_j^2/100$. Mean Function Evaluations and Mean Iterations are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

which higher degrees of parallelization ($P > 1$) improve on the non-parallel ($P = 1$) algorithm. A gain greater than 1 indicates improvement over the non-parallel simplex, and a gain of 1 indicates no improvement. The number of objective function evaluations required to reach $f(\theta) = 0.1$ measures the performance of the parallel algorithm if the algorithm were implemented sequentially using a single processor. On a single processor, only one objective function evaluation can be performed at a time. As we discuss below, parallelization of the simplex algorithm may provide computational savings, even if performed on a single processor, if parallelization improves the search direction of the algorithm.

In this experiment, we find that increasing the degree of parallelization up to $P = 80$ degrees of parallelization reduces the number of objective function evaluations required to reach $f(\theta) = 0.1$. With 80 degrees of parallelization, only 795 objective function evaluations are required, a 2.5-fold gain over the non-parallel simplex algorithm. However, the performance of the processors is far lower for 90 degrees of parallelization. On average, 3,521 function evaluations are required for $P = 90$ degrees of parallelization, which is more than is required for the non-parallel case. If implemented on a single processor, the parallel algorithm performs worse than the non-parallel algorithm for degrees of parallelization close to the number of parameters.

The second half of Table 2 reports the performance gain of the parallel algorithm if implemented using the same number of processors as the degree of parallelization, i.e. using P processors. For P processors, each iteration consists of up to P objective function evaluations. Using a single processor with one degree of parallelization, by definition the number of iterations is equal to the number of objective function evaluations. For two processors ($P = 2$), the mean number of iterations needed is 1,100, a 1.8-fold reduction in computation time relative to $P = 1$. As the degree of parallelization increases, the mean number of iterations required falls. For degrees of parallelization $10 \leq P \leq 80$, the parallel algorithm reduces computation time at a rate greater than P. For 80 processors, only 12.6 iterations are required to achieve $f(\theta) = 0.1$. This is a 160-fold or $2P$ reduction in computation time over a single processor. However, the reduction in computation time is not monotonically increasing. 90 processors requires on average 48.7 iterations to reach $f(\theta) = 0.1$. This is still a 41-fold reduction in computation time relative to $P = 1$. But the gain for $P = 90$ is less than P, and less than the gains using fewer degrees of parallelization.

Example 2

For our second example, we consider the sum of absolute values with $J = 100$ parameters:

$$f(\theta) = \sum_{j=1}^{100} |\theta_j|/100.$$

The true minimum is again $\theta_j = 0$ for all j. In this example, gradient based methods are not applicable since the derivative of $f(\theta)$ at the minimum does not exist. We conduct the experiment for this objective function identically to the first example.

As in Example 1, Tables 3 and 4 indicate that the parallel algorithm produces substantial computational savings over the non-parallel algorithm if implemented using P processors. However, unlike in Example 1, there are generally no gains to implementing the parallel algorithm in a serial fashion on a single processor. The one exception is in the case for $P = 50$ processors, where the parallel algorithm outperforms the non-parallel algorithm by 50%.

The second half of Table 4 measures the computational savings to using the same number of processors as degrees of parallelization. The computational gains are monotonically increasing in the degree of parallelization. However, the gains are more modest than those in Example 1. For instance, using $P = 50$ processors provides a 53-fold gain over the non-parallel algorithm for Example 2, but an 87-fold gain for Example 1.

Example 3

Example 3 uses the same objective function as Example 1, but with twice as many parameters ($J = 200$) (Table 5).

$$f(\theta) = \sum_{j=1}^{200} \theta_j^2/200$$

This example is conducted in the same way as the previous examples.

As with the previous examples, there is a substantial gain to parallelization. Table 6 indicates that the gains to parallelization are highest at $P = 150$. For a single processor, on average 7,071 iterations are required to achieve $f(\theta) = 0.1$. Using 150 processors, requires on average only 15 iterations, a 472-fold reduction in computation time. This

Table 3 (Example 2): Mean function value by degree of parallelization (P)

	Function evaluations						
P	0	500	1000	1500	2000	3000	5000
1	0.780	0.556	0.365	0.310	0.263	0.195	0.127
2	0.780	0.541	0.358	0.310	0.264	0.199	0.134
3	0.780	0.525	0.356	0.304	0.260	0.196	0.131
5	0.780	0.490	0.350	0.300	0.260	0.200	0.135
10	0.780	0.429	0.345	0.298	0.262	0.206	0.147
20	0.780	0.387	0.330	0.283	0.250	0.203	0.151
30	0.780	0.337	0.286	0.242	0.214	0.176	0.137
50	0.780	0.402	0.253	0.198	0.163	0.128	0.099
80	0.780	0.396	0.310	0.248	0.210	0.173	0.145
90	0.780	0.539	0.438	0.370	0.328	0.287	0.247

Notes: P is degree of parallelization. Results from Example 2 minimizing the objective function $\sum_{j=1}^{100} |\theta_j|/100$. Function values are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

Table 4 (Example 2): Computation required to reach $f(\theta) = 0.1$

P	Mean funct. evals.	Gain	Mean iterations	Gain
1	5652	1	5651.88	1
2	5935	1.05	3220	1.76
3	5797	0.975	2233	2.53
5	5933	0.953	1503	3.76
10	6374	0.887	924	6.12
20	6078	0.930	476	11.8
30	5370	1.05	280	20.2
50	3763	1.50	107	52.7
80	5926	0.954	99.3	56.9
90	6879	0.823	95.9	58.9

Notes: P is degree of parallelization. Results from Example 2 minimizing the objective function $\sum_{j=1}^{100} |\theta_j|/100$. Mean Function Evaluations and Mean Iterations are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

reduction in computation time is more than three times the degree of parallelization. Substantial, yet less impressive gains, are evident at lower degrees of parallelization. For $P = 50$, the mean number of iterations required is 100, or a 71-fold reduction in computation time relative to a single processor. As is evident in Examples 1 and 2, high degrees of parallelization approaching the number of parameters (P close to J) is not optimal. For $P = 190$, the mean number of iterations required is 54.7. This is still much smaller than that required for a single processor, but is larger than that required for 150 processors.

Table 5 (Example 3): Mean function value by degree of parallelization (P)

	Function evaluations						
P	0	500	1000	1500	2000	3000	5000
1	0.965	0.878	0.758	0.654	0.545	0.387	0.188
2	0.965	0.872	0.743	0.620	0.503	0.392	0.208
5	0.965	0.857	0.687	0.513	0.439	0.366	0.203
s10	0.965	0.839	0.594	0.418	0.383	0.271	0.143
20	0.965	0.830	0.472	0.342	0.318	0.184	0.098
50	0.965	0.812	0.322	0.256	0.231	0.138	0.069
100	0.965	0.819	0.354	0.209	0.162	0.107	0.063
150	0.965	0.826	0.430	0.125	0.087	0.049	0.031
180	0.965	0.831	0.539	0.263	0.220	0.157	0.138
190	0.965	0.828	0.635	0.449	0.374	0.326	0.306

Notes: P is degree of parallelization. Results from Example 3 minimizing the objective function $\sum_{j=1}^{200} \theta_j^2/200$. Function values are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

Table 6 (Example 3): Computation required to reach $f(\theta) = 0.1$

P	Mean funct. evals.	Gain	Mean iterations	Gain
1	7071	1	7071	1
2	7360	0.961	3811	1.86
5	7304	0.968	1664	4.25
10	6132	1.15	728	9.72
20	5194	1.36	319	22.1
50	3957	1.79	99.7	71.0
100	3315	2.13	37.8	187
150	1918	3.69	15.0	472
180	8114	0.871	51.8	137
190	9834	0.719	54.7	129

Notes: P is degree of parallelization. Results from Example 3 minimizing the objective function $\sum_{j=1}^{200} \theta_j^2/200$. Mean Function Evaluations and Mean Iterations are averages over 100 experiments using different starting values. The same starting values are used for each degree of parallelization

5 Discussion

In understanding the performance of our parallel simplex algorithm it is important to note that the parallel algorithm potentially has a different search path through the parameter space than the non-parallel algorithm. In each iteration of the non-parallel NM simplex algorithm the worst point on the simplex is reflected and replaced by an updated point (the reflection, expansion, or contraction point). The search direction

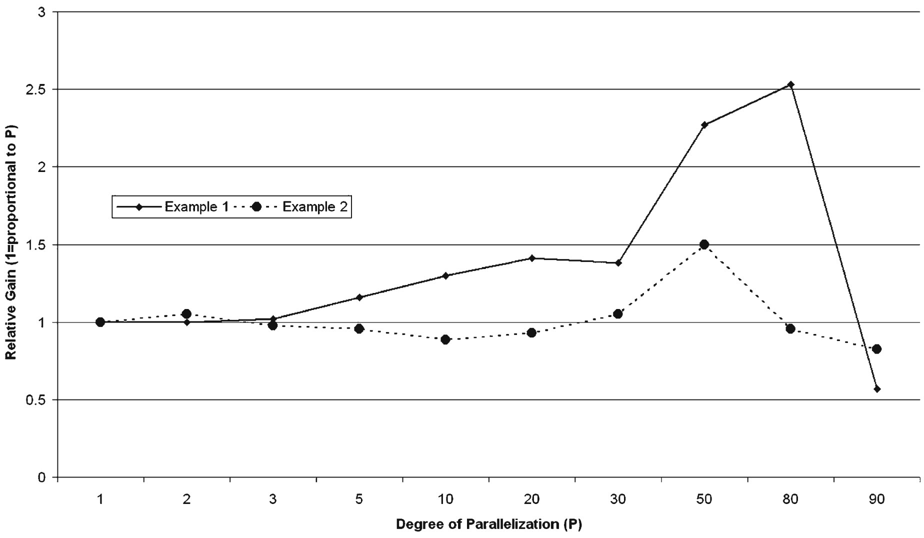

Fig. 1 Comparison of relative gains from examples 1 and 2. *Notes:* Results from Example 1 (Table 2) minimizing the objective function $\sum_{j=1}^{100} \theta_j^2/100$ and Example 2 (Table 4) minimizing the objective function $\sum_{j=1}^{100} |\theta_j|/100$

of the algorithm is away from the one worst point and toward the better points on the simplex. In the parallel simplex algorithm with P degrees of parallelization, in addition to the worst point, the next $P-1$ worst points on the simplex are also replaced. In our parallel algorithm, the reflection point is not the same as it is in the non-parallel algorithm. This is because as the degree of parallelization increases, the centroid is calculated using fewer and better simplex points as the centroid excludes the additional $P-1$ worst points. This "purer" centroid may be a better or worse search direction than the centroid calculated from all but the one worst simplex point in the non-parallel algorithm.

In all of the Monte Carlo experiments and for all degrees of parallelization, the parallel algorithm exhibited substantial computational savings over the non-parallel algorithm. However, the performance did vary substantially across experiments and degrees of parallelization. In some cases, in particular in Example 2, the gains in computational time are lower than the degree of parallelization P. One interpretation of this finding is that in some cases the additional processors used in the parallel algorithm are not as productive as the one processor used in the non-parallel case. That is, the search direction and the associated evaluation points assigned to each processor are not optimal.

Figure 1 displays the gains to parallelization for Examples 1 and 2. The optimal degree of parallelization was around $P = 80$ in Example 1 and $P = 50$ in Example 2. This suggests that the optimal degree of parallelization is related to the smoothness of the objective function near the optimum. For a known non-smooth function, a lower degree of parallelization may be warranted.

This relative slowdown in performance with additional degrees of parallelization is not in general a feature of parallelization at the objective function level. In objective function parallelization, each processor assigned a part of the objective function, such as a subset of observations, has the same productivity or performance in completing its task as any other processor. The gains to objective function parallelization is generally $1P$ proportional to the degree of parallelization.

In some of our Monte Carlo experiments we find that the computational gain using the parallel algorithm exceeds the degree of parallelization P. In Examples 1 and 3, we find that for degrees of parallelization $10 \leq P \leq 80$ and $20 \leq P \leq 150$, respectively, the computational gains for the parallel algorithm is greater than $1P$. At the optimal level of parallelization $P = 80$ and $P = 150$, the gains are $2P$ and $3P$, respectively. Our interpretation of this finding is that the parallel algorithm's search direction using the purer centroid purged of the P worst points is superior to that of the non-parallel algorithm. The superior search direction is especially evident in the finding that the parallel algorithm has considerable computational gains over the non-parallel algorithm, even if implemented on a single processor in a serial fashion.

As an additional advantage to parallelization, parallel methods might better cope with objective functions that have multiple local minima, which are inherent to many optimization problems based on complex economic models. In our examples, there are no local minima aside from the unique global minima for each example. The relative performance of parallel simplex algorithms in finding local minima may be a fruitful topic for future research.

6 Conclusion

This paper develops a generalization of the Nelder–Mead (1967) simplex algorithm for parallel processors. In contrast to the more common parallelization at the objective function level in which each processor is assigned part of the objective function to evaluate at given vector of parameters, our parallel approach uses parallelization at the parameter level and assigns each processor a different set of parameters. The advantage of this approach is that our parallel simplex algorithm is generic and can be applied, without re-writing computer code, to any optimization problem to which the NM simplex is applicable.

We show in a series of Monte Carlo experiments the potentially large gains to parallelizing the simplex algorithm. In general, the gains in computer time are increasing in the degree of parallelization. These gains varied across our examples and the degree of parallelization. For some of our experiments, we find that the gains to parallelization could be as high as three times the number of processors. As economic models become increasingly complex and involve larger numbers of parameters, using parameter level parallelization algorithms will become increasingly important.

References

Barr, R. S., & Hickman, B. L. (1994). Parallel simplex for large pure network problems: Computational testing and sources of speedup. *Operations Research, 42*(1), 65–80.

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

Bixby, R. E., & Martin, A. (2000). Parallelizing the dual simplex method. *INFORMS. Journal on Computing, 12*(1), 45–56.

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

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

Hotz, V. J., & Miller, R. A. (1993). Conditional choice probabilities and the estimation of dynamic models. *Review of Economic Studies, 60*(3), 497–529.

Dennis, J. E. J., & Torczo, V. (1991). Direct search methods on parallel machines. *SIAM Journal of Optimization, 1*(4), 448–474.

Keane, M. P., & Wolpin, K. I. (1994), The solution and estimation of discrete choice dynamic programming models by simulation and interpolation: Monte Carlo evidence. *Review of Economics and Statistics, 76*(4), 648–672.

Klabjan, D., Johnson, E. L., & Nemhauser, G. L. (2000). A parallel primal-dual simplex algorithm. *Operations Research Letters, 27*, 47–55.

Lee, D., & Wolpin, K. (2006), Intersectoral labor mobility and the growth of the service sector. *Econometrica, 74*(1), 1–46.

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

Swann, C. A. (2002). Maximum likelihood estimation using parallel computing: An introduction to MPI. *Computational Economics, 19*(2), 145–178.

Todd, P., & Wolpin, K. (2006). Assessing the impact of a school subsidy program in Mexico: Using a social experiment to validate a dynamic behavioral model of child schooling and fertility. *American Economic Review, 96*(5), 1384–1417.

van der Klaauw, W., & Wolpin, K. I. (2006). Social security and the retirement and savings behavior of low income Households. Working paper.