A Parallel Depth-first Search Algorithm for Global Optimization Using Interval Analysis: The Single Variable Case

Adam Baldwin & Asai Asaithambi
Computer Science Department
The University of South Dakota
Vermillion, South Dakota 57069
adam.baldwin@usd.edu
asai.asaithambi@usd.edu

Abstract

Finding the global minimum of an arbitrary differentiable function over an n-dimensional rectangle is an important problem in computational science, with applications in many disciplines. We have developed a depth-first search method to reliably obtain the global minimum of an arbitrary continuously differentiable function in the one-dimensional case. Our algorithm reliably computes the global minimum for standard test functions in the literature, and requires much less computational effort than previously used breadth-first search methods. A parallel implementation of the algorithm demonstrates the expected speed-up as the number of processors is increased. Our method can be extended to the multidimensional case, which will be reported in a future publication.

1 Introduction

A problem that has been important in computational science for a number of years is that of finding the minimum (and maximum) values of a function of several variables in an n-dimensional rectangle [1]. For the one-dimensional case, the problem can be defined as finding

$$f^* = \min_{x \in X} f(x)$$

where the objective function $f : \mathbb{R} \to \mathbb{R}$ is continuously differentiable and $X \subseteq \mathbb{R}$. The global minimum value is denoted by f^*. The value x for which the function will evaluate to the global minimum f^* is called the *minimizer*.

Also, finding good upper and lower bounds on the values of functions is important in fields such as sensitivity analysis, perturbation analysis, error analysis, and many other optimization problems [1]. The problem of finding the range of values of a function is solved by finding both the global minimum and global maximum for the function f over the initial domain X.

Since the global maximum value can be obtained by finding the minimum value f^* of $-f(x)$ and returning the value $-f^*$, we may reduce our primary problem of global optimization into one of obtaining a good lower bound of a function over a specified interval.

In 1966, R.E. Moore published a book *Interval Analysis* [3] in which he described a new method for numerical calculations, called interval arithmetic, which has now become a field in its own right called interval analysis. Interval analysis is widely used to address many concerns related to real-valued numerical calculations, particularly those related to rounding error introduced in machine computations. The appeal of interval analysis is that, rather than a machine-rounded approximation, our solutions are intervals that guarantee enclosure of the desired solution.

Existing methods for finding the range of values of a function use an exhaustive search with a pruning strategy. That is, the initial search domain is subdivided by a specific criteria, a pruning algorithm is applied to eliminate regions that are known to not contain f^*, and the search for the optimum values is carried out in those smaller subregions. For instance, in the Asaithambi, Zuhe, and Moore algorithm (hereafter referred to as the AZM algorithm) [1][4], the initial domain is bisected, and each subregion further explored using an interval extension based upon a monotonicity test and the mean value theorem. A list is used to store regions that may contain the correct minimum, and those regions that have been examined are removed from the list. The bounds of any monotonic regions can be easily found, and the remaining non-monotonic regions can continue to be bisected and searched for the minimizer. The AZM algorithm has been a benchmark for the evaluation and development of our new algorithm.

2 Initial Phase of Research

During the initial test phase of the AZM algorithm, it became very clear that the element that required a large amount of computational time was the list that stores regions yet to be examined for the global minimum value. During the execution of the AZM algorithm, regions are bisected and stored on the list if they must be bisected further to examine for the global minimum. These regions are inserted into the *list in the order of increasing lower bounds*. As the list size increases, the amount of time it takes to insert an element in the list in the proper location becomes quite large.

A key motivation for our research was to find a method that reduced our dependence on this list, in order to speed up the process of locating the global minimum. In addition, we anticipate that a parallelization of the AZM algorithm that uses a list may have an unreasonable amount of time devoted to the sending and receiving of elements to be inserted into the list. This would lead to a communication overhead for load balancing that would be inefficient. Thus, we began investigating alternative methods for obtaining the global minimum that did not rely on the management of list elements.

2.1 The Sequential Recursive Algorithm and Its Implementation

The result of our exploration of list dependency in the original AZM algorithm is the development of a recursive algorithm as a depth-first exhaustive search for the global minimum of a continuously differentiable function in one dimension. This algorithm, the Asaithambi-Baldwin algorithm (after A. Asaithambi and A. Baldwin), will be hereafter referred to as the AB algorithm. There are several differences between the AZM and AB algorithm. For instance, the AZM algorithm uses the monotonicity test form interval extension, an extension that is not used in the AB algorithm. In addition, AB does not use a list or insert function, but rather relies upon the system's stack, which handles recursive calls, to store regions that have yet to be examined. This eliminates our list dependency, and leads to a substantial improvement in performance when the AB algorithm is parallelized. Similarities involve the use of the natural interval extension on the first derivative of the function, referred to as $DF(X)$ to determine monotonicity, and a bisection method for generating subregions to be examined.

The AB algorithm checks an interval for monotonicity (increasing or decreasing) of f upon receipt of an interval region to search. If f is not monotonic on the region tested, and the width of the interval is greater than some desired tolerance, AB bisects the interval into two smaller subintervals of equal width. AB is then recursively called on the two subintervals that were generated, provided that the termination criteria has not been reached. If the desired tolerance is reached, the minimum value is obtained by evaluating the natural interval extension of the function on the subregion of width less than the termination criteria. At each stage of the recursion, the global minimum is updated if the bound obtained is lower than obtained previously. The recursive call eliminates the need to store items in a list, as the system implementation of recursive calls will automatically take care of unchecked subregions. In addition, when the value of f^* is updated, we can also update the region in

which f^* was located, thus providing us with the minimizer of the function on the initial search domain.

The algorithm needs only to check for regions in which the function is non-monotonic. The global minimum can only occur on the interior of a region if it is located at a turning point of the function. That is to say that the global minimum has only a finite number of possible locations: at one of the endpoints of the region or at a turning point on the interior of the region. The AB algorithm exploits this property by only evaluating non-monotonic regions, and then comparing the final minimum value obtained on the interior of the initial region to the value of the function at the endpoints of the initial region. The AB algorithm returns the smallest of these three values; the global minimum.

Our experiments show that the AB algorithm works on all one-dimensional functions that we tested [6][5][2]. We selected 31 functions with a variety of characteristics for the testing of the our algorithm. The sequential AB algorithm is presented in Figure 1, in which $DF(X)$ denotes the natural interval extension of $f'(x)$.

Sequential AB Algorithm

```
 1 : input region X;
 2 : if DF(X) ≤ 0 && overline(DF(X)) ≥ 0
 5 :     if width(X) ≥ ε
 6 :         bisect X into X1, X2;
 7 :         lo1 = AB(X1);
 8 :         lo = lo1;
 9 :         lo2 = AB(X2);
10 :         if lo > lo2
11 :             lo = lo2;
12 :     else
13 :         lo = f(mid(X));
14 : return lo;
```

Figure 1: Sequential AB Algorithm

2.1.1 Sequential AB versus AZM

We tested both the AZM and AB algorithm on all of our one-dimensional test functions. The sequential AB algorithm obtained the minimum value in less time than the AZM algorithm that was used as our baseline for performance measurement. At times, this speedup achieved by using the sequential AB algorithm over the AZM algorithm was quite substantial. A table comparing the run time of these two sequential algorithms for several functions, as well as the reduction in run time is included below (a complete list of our one-dimensional test functions can be found in Appendix A):

Function	AZM (milliseconds)	AB (milliseconds)	% Reduction
$f(x) = \sin(x) - 2\cos(x^2 - 1)$	7813.84	962.41	88
$f(x) = x(1 - x)$	0.19	0.12	35
$f(x) = \frac{x^2}{4000} - \cos(x) + 1$	10.26	4.94	52
$f(x) = 24x^4 - 142x^3 + 303x^2 - 276x + 93$	80.08	25.29	68
$f(x) = \sin(x) + \sin\left(\frac{10x}{3}\right) + ln(x) - 0.84x$	1.20	0.69	42

Table 1: Speedup of Sequential AB over AZM

3 The Parallel Algorithm and Its Implementation

Due to the nature of the recursive AB algorithm, it is much easier to parallelize than the AZM algorithm. For this reason, we are able to see a speedup in the AB algorithm for many examples with a relatively straightforward parallelization of the algorithm. We note that our parallel implementation of the AB algorithm works on all one-dimensional test functions that were used during the preliminary testing of both the sequential AB and AZM algorithms.

3.1 Parallel AB

In order to assign tasks for multiple processors to simultaneously complete, we first generate a list of subregions of X to be distributed to all processors. We may use either the ideas from the original AZM algorithm or the AB algorithm to generate this list of subregions. These methods are discussed in more detail in section 3.1.1. The processors are sequentially ranked from 1 through k when k processors are used. Processor #1 acts as the master, assigning subregions to each processor, and collecting the global minimum from each of the local minimum values reported by each processor. Each of the other processors calls the AB algorithm using a bisection of the region that is assigned, hence two recursive calls for each iteration. The parallel AB algorithm is shown in Figure 2.

3.1.1 Generating List of Subregions

There are several ways in which the master processor can generate a list of subregions to be distributed amongst the other processors. For instance, we began by using the original AZM algorithm to generate a list of subregions equal to the number of processors. Since the AZM algorithm keeps a working list of regions that may contain the global minimum, we simply terminate the AZM algorithm when we have generated a desired list size. The minimizer is guaranteed to be contained in one of the elements generated by AZM [1][4]. Each processor then has some number of regions on which to run the recursive AB algorithm. Another approach to further reduce the run time is to generate the list of subregions by

```
                    Parallel AB Algorithm
 1 : input region X;
 2 : if id == master
 5 :     Generate list of subregions
 6 :     Broadcast list to all processors
 7 : else
 9 :     Run AB on assigned subregions
10 :     Update local minimum at each completion of AB
11 : global_min = min(local_min)
12 : return global_min
```

Figure 2: Parallel AB Algorithm

checking a region for monotonicity of f; if it is non-monotonic on a region, we bisect it into two subregions, and push both subregions to the end of our working list. We then test the new front element on our list for monotonicity and repeat the process until our list reaches a desired size. Our list elements are stored in the order obtained, as the order in which the subregions are distributed to the processors will be unimportant. Since we are only eliminating regions that are monotonic, we are able to guarantee that the minimizer is not discarded by this process. Moreover, there is no need to look for a proper place in the list to insert the newly obtained subregions.

4 Numerical Results

We experimented extensively on all example functions (included in Appendix A) and found that there were some characteristics of functions that led to a substantial decrease in run time as the number of processors was increased. These properties, as well as examples are provided in this section.

4.1 Performance Increase for Highly Multimodal Functions

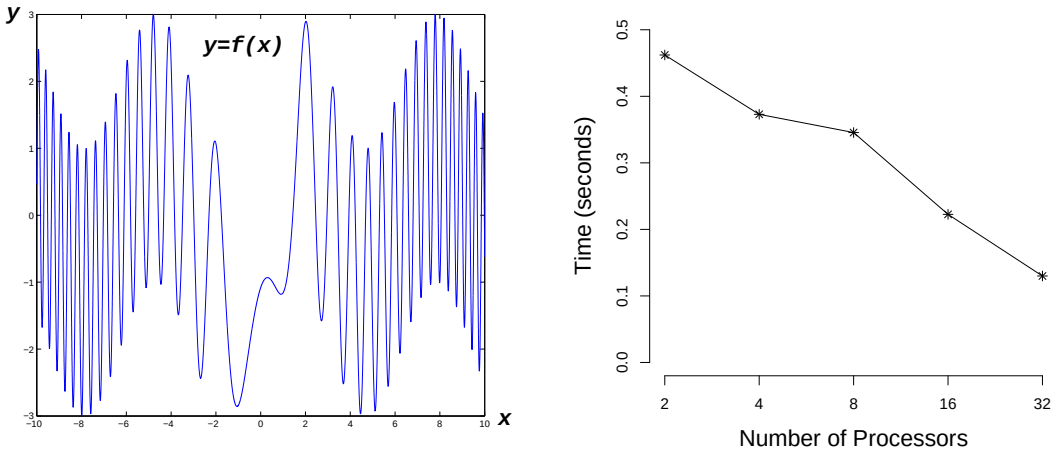
As we might expect, the method used to develop the parallel AB algorithm does not provide speedup for functions that have a small number of local extrema. This is of little concern, however, since obtaining the minimum values on those functions is trivial. Therefore, our primary focus was devoted to achieving the highest speedup possible for more complicated, highly multimodal functions. The increased complexity of finding the global minimum value on these types of functions allowed us to heavily test our new algorithm and devise a method that works very well. In addition, our experience with the complicated one-dimensional functions leads us to believe that we will see similar results in the multi-dimensional case, where the number of subregions to be considered is often times larger than the single variable case.

The decrease in run time for functions with a large number of local extrema can be justified

rather intuitively. Each of the bisected regions produced by our generation of subregions to be considered will be non-monotonic due to the high degree of multimodality of the function. Therefore, each subregion listed will require further consideration in obtaining the global minimum. Thus, each of the processors is assigned meaningful work. Increasing the number of processors reduces the amount of work that needs to be done by each processor. The performance increase for two of our test functions is discussed in the next sections.

4.2 Example: $f(x) = \sin(x) - 2\cos(x^2 - 1)$

This function was tested on the search domain $[-100, 100]$, in which the function is highly multimodal. The global minimum value for this function is $f^* = -3$. Due to the presence of a large number of local extrema, we were able to achieve substantial speedup over both the original AZM algorithm and the sequential AB algorithm. The run time for the sequential AZM algorithm was 7,813 milliseconds, compared to the sequential AB algorithm run time of 962 milliseconds. When we increased the number of processors to 32 (including the master processor), we were able to further reduce the run time to 130 milliseconds. A plot of the function demonstrating is high degree of multimodality on the search domain $[-10, 10]$, as well as the reduction in run time as the number of processors was increased is shown in Figure 3:



(a) Graph of Function 1

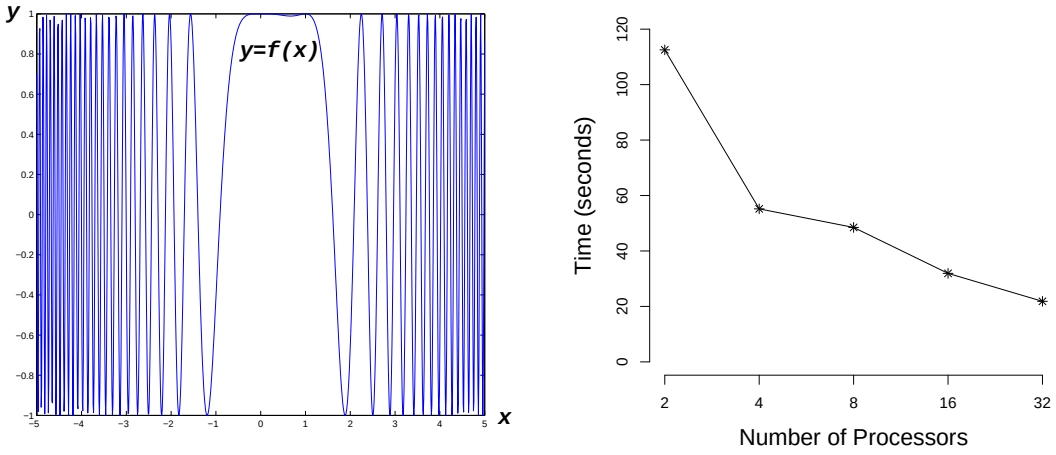
(b) Speedup

Figure 3: $f(x) = \sin(x) - 2\cos(x^2 - 1)$

4.3 Example: $f(x) = \cos(x^2 - x^3)$

This function was also tested on the search domain $[-100, 100]$; a domain in which the function is highly multimodal. The global minimum value for this function is $f^* = -1$. We were unable to determine the run time for the sequential AZM algorithm due to a list

overflow. This was the result of a list size that was not manageable by our hardware, as the list size was in excess of 200,000 elements. However, when we ran the sequential AB algorithm for this function, we obtained a run time of 112,536 milliseconds. When we increased the number of processors to 32 (including the master processor), we were able to further reduce the run time to 21,834 milliseconds. Thus, compared to the baseline AZM algorithm, for this example, we were able to demonstrate that some test domains that are unmanageable due to the AZM algorithm's list dependence were quickly solved by our use of the system's stack to handle recursive calls. A plot of the function on the search domain $[-5, 5]$, as well as the reduction in run time as the number of processors was increased is shown in Figure 4:



(a) Graph of Function 7

(b) Speedup

Figure 4: $f(x) = \cos(x^2 - x^3)$

4.4 Further Reduction in Run Time

For both of the examples discussed in the previous section, we were able to obtain a further reduction in computational effort by applying a load balancing strategy (discussed in section 5.1.1). For the example in section 4.2, generating a larger list of subregions to distribute amongst the processors led to decrease the run time from 130 milliseconds to only 20 milliseconds. Thus, compared to the baseline AZM algorithm, for this example, we were able to reduce the run time by 99.7 percent. For the example in section 4.3, generating a larger list of subregions first to distribute amongst the processors (providing 190 subregions per processor), we were able to decrease the run time to only 893 milliseconds. This marks a substantial improvement, as the search domain was too large for the AZM algorithm to successfully terminate with the global minimum.

5 Future Work

5.1 Load Balancing Techniques

In order to ensure that no processor remains idle for an unnecessarily long time, it is important for any parallel algorithm to balance the work load appropriately. In the case of the parallel AB algorithm, each processor is sent subregions on which f is non-monotonic. We were able to increase the efficiency of the algorithm by sending each processor more than one subregion. Given the nature of this load balancing technique, we are able to generate a list of subregions that is at most equal to the number of extreme points of the function on the initial search domain. Thus, we may need to terminate our subregion list generation when it is no longer possible to increase the list size.

5.1.1 Better Load Balancing

In an attempt to balance the load more efficiently, we experimented by letting the master processor generate lists of non-monotonic subregions of various sizes. We then distribute these lists to the other processors and have them each obtain the minimum on a subset of those regions listed. We tested this list size up to 128 times the number of processors used in order to see how the load balancing effected the performance of the algorithm. We did see improvement in the performance as the number of subregions in our distributed list became larger for some functions, however, there appeared to be a limit as to how many subregions could be used. That is to say, at a certain point (different for each function tested), increasing the number of subregions per processor provided no further increase in efficiency. The optimum number of subregions per processor is contingent upon the function being tested; more specifically, on the number of extreme points of the function in the initial search domain. In some cases, we were able to reduce the run time by as much as 96 percent over the speedup already achieved by simply increasing the number of processors. Two examples of this are seen in sections 4.2 and 4.3 This load balancing technique shows promise for increasing the efficiency of the parallel AB algorithm, and is being further investigated at this time.

5.2 The Multidimensional Case

Motivated by our success with the single variable case, we are presently investigating the application of this method to multi-variable functions. Our experimentation suggests that functions that are highly multimodal and hence require a large number of bisections are the functions for which our method will be best suited. We believe that the AB algorithm will out-perform the AZM algorithm in obtaining the global minimum of all multi-dimensional functions as well.

6 Conclusion

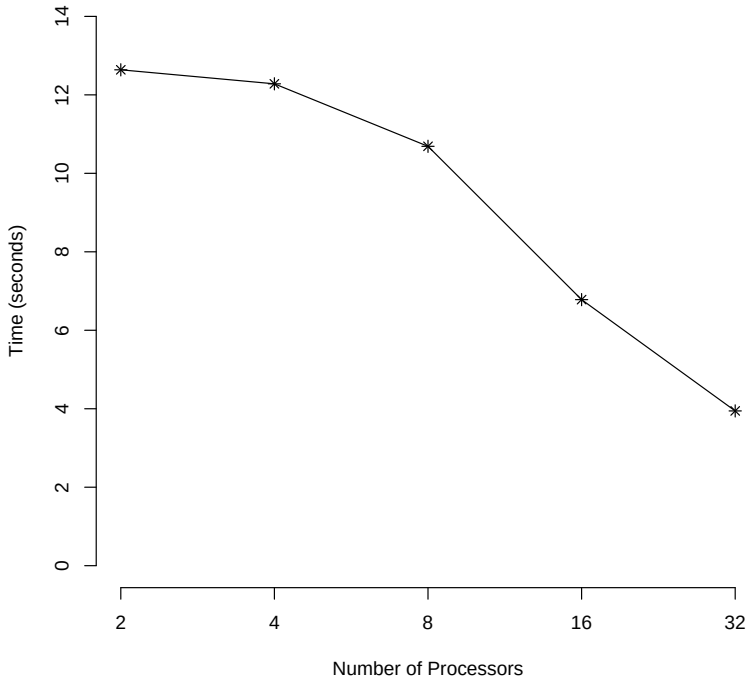
We have developed, implemented, and tested sequential and parallel versions of a depth-first search algorithm for finding the global minimum of a function of one variable. We

were able to achieve a substantial reduction in run time while still reliably obtaining the global minimum for a variety of functions previously examined in the literature. For functions with a large number of local extrema on the initial search domain, we are able to see even more reduction in run time as the number of processors used in the parallel AB algorithm are increased. We have demonstrated that the AB algorithm is reliable in all tested cases, and is more efficient than the AZM algorithm, while its parallelization also exhibits the expected speedup characteristics as the number of processors is increased. We have also demonstrated that the use of load-balancing techniques results in further reduction in computational effort for highly multimodal functions.

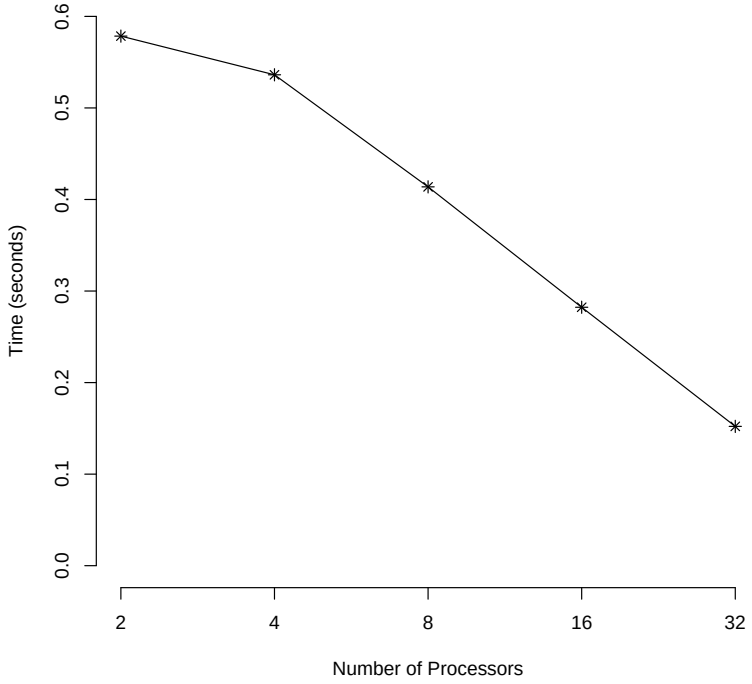
Appendix A: Test Functions[5][1][2][6]

1. $f(x) = \sin(x) - 2\cos(x^2 - 1)$
2. $f(x) = \cos\left(\pi\left(8x^3 - 1\right)\right) + \sin\left(\pi\left(8x^2 - 1\right)\right)$
3. $f(x) = e^{\sin(x)} + \cos\left(x^2\right)$
4. $f(x) = \cos\left(\sin\left(x^2 - 1\right) - 1\right)$
5. $f(x) = \sin\left(\cos\left(e^x\right)\right)$
6. $f(x) = \sin\left(x^4 + x^3 + x^2 + x + 1\right)$
7. $f(x) = \cos\left(x^2 - x^3\right)$
8. $f(x) = x(1 - x)$
9. $f(x) = -x^5 + x^4 + x^3 + x^2 + x + 1$
10. $f(x) = \dfrac{x^2}{4000} - \cos(x) + 1$
11. $f(x_1, \cdots, x_5) = \sum_{i=1}^{5} \dfrac{x_i^2}{400} - \prod_{i=1}^{5} \cos\left(\dfrac{x_i}{\sqrt{i}}\right) + 1$
12. $f(x) = (x + \sin(x)) \cdot e^{-x^2}$
13. $f(x) = -\sum_{k=1}^{5} k \cdot \sin((k+1)x + k)$
14. $f(x) = \sin(x)$
15. $f(x) = e^{-3x} - \sin^3(x)$
16. $f(x) = \sin\left(\dfrac{1}{x}\right)$
17. $f(x) = x^4 - 10x^3 + 35x^2 - 50x + 24$
18. $f(x) = 24x^4 - 142x^3 + 303x^2 - 276x + 93$
19. $f(x) = \sin(x) + \sin\left(\dfrac{10x}{3}\right) + ln(x) - 0.84x$
20. $f(x) = 2x^2 - \dfrac{3}{100} e^{-(200(x - 0.0675))^2}$
21. $f(x) = -\sum_{i=1}^{10} \dfrac{1}{(k_i(x - a_i))^2 + c_i} \quad a_i, k_i, c_i \in \mathbb{R}^+$
22. $f(x) = -\sum_{i=1}^{10} \dfrac{1}{(k_i(x - a_i))^2 + c_i} \quad a_i, k_i, c_i \in \mathbb{R}^+$
23. $f(x) = \dfrac{x^2}{20} - \cos(x) + 2$
24. $f(x) = -\dfrac{1}{(x-2)^2 + 3}$
25. $f(x) = x^2 - \cos(18x)$
26. $f(x) = (x-1)^2\left(1 + 10\sin^2(x+1)\right) + 1$
27. $f(x) = e^{x^2}$
28. $f(x) = x^4 - 12x^3 + 47x^2 - 60x - 20e^{-x}$
29. $f(x) = x^6 - 15x^4 + 27x^2 + 250$
30. $f(x) = \sin^2\left(1 + \dfrac{x-1}{4}\right) + \left(\dfrac{x-1}{4}\right)^2$
31. $f(x) = \left(x - x^2\right)^2 + (x-1)^2$

Appendix B: Speedup for Additional Selected Functions

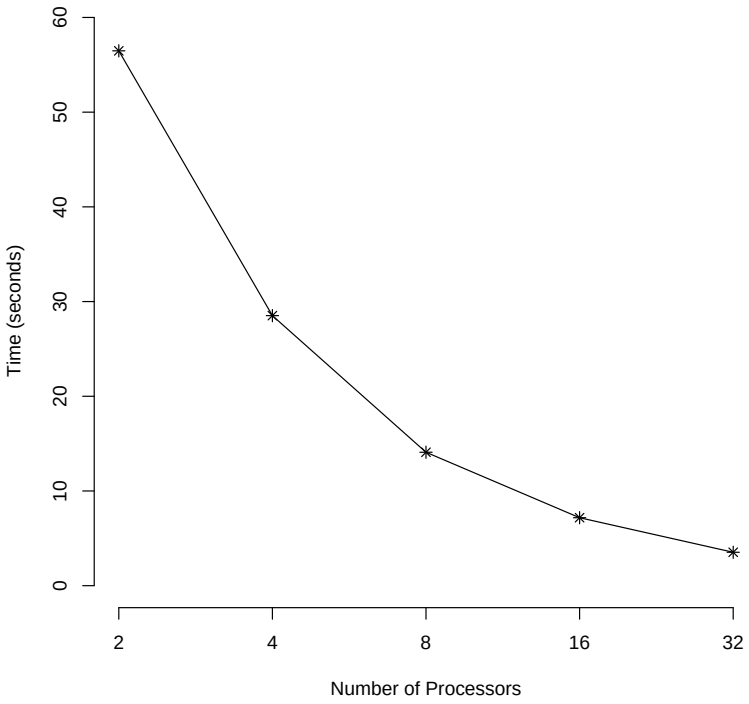


(a) Function #2

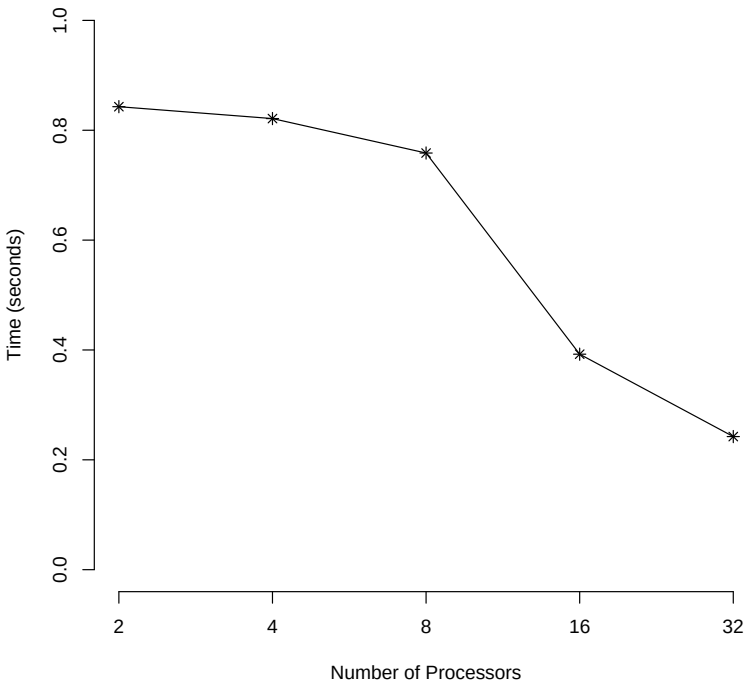


(b) Function #3

Appendix B: Continued

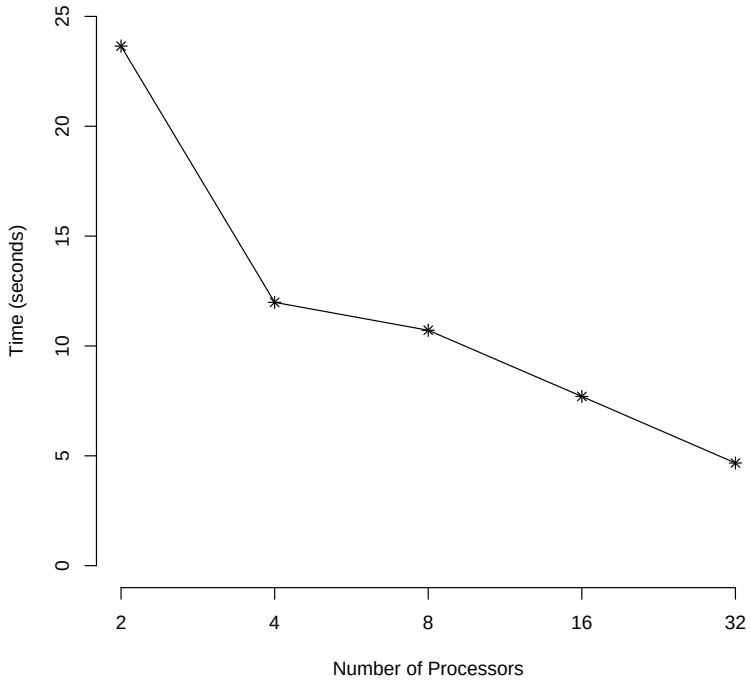


(a) Function #4



(b) Function #5

Appendix B: Continued



(a) Function #6

References

[1] ASAITHAMBI, N., ZUHE, S., AND MOORE, R. On computing the range of values. *Computing* (1981), 225–237.

[2] CHO, H., OLIVERA, F., AND GUIKEMA, S. D. A derivation of the number of minima of the griewank function. *Applied Mathematics and Computation* (2008), 694–701.

[3] MOORE, R. E. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[4] NATARAJ, P. V., AND KUBAL, N. On the asaithambi-zuhe-moore algorithm for computing the range of values. *Computing* (2007), 247–253.

[5] POHLHEIM, H. *Genetic and Evolutionary Algorithm Toolbox for Matlab Examples of Objective Functions*, geatbx version 3.8 ed., December 2006.

[6] RATZ, D. A nonsmooth global optimization technique using slopes: The one-dimensional case. *Journal of Global Optimization* (1999), 365–393.