SPECULATIVE EXECUTION MULTI-CORE AND FPGA NELDER-MEAD IMPLEMENTATIONS FOR HIGH PERFORMANCE UNCONSTRAINED OPTIMIZATION

Artur Mariano

Institute for Scientific Computing
Technische Universität Darmstadt
Darmstadt, Germany
artur.mariano@sc.tu-darmstadt.de

Paulo Garcia, Tiago Gomes

Centro Algoritmi
Universidade do Minho
Guimarães, Portugal
paulo.garcia@algoritmi.uminho.pt,
tiago.a.gomes@algoritmi.uminho.pt

ABSTRACT

This paper proposes a new parallel version of the Nelder-Mead algorithm - based on speculatively executing the operations applied to the *simplex* - implemented in two different ways of controlling the fork-and-join mechanism. It also compares these x86 parallel and sequential versions of the algorithm with handwritten and automatic C-to-RTL FPGA designs. As the execution flow of the software is replicated in the FPGA designs, it is also compared the efficiency of the synchronization of different execution flows, when implemented by software and hardware.

Performance trials of these versions where performed using (i) a last-generation FPGA and a last generation multi-core CPU-chip to run the software versions and (ii) relatively simple objective functions in $\mathbb{R}^2$. Results show that performance of the handwritten hardware design is relatively equivalent to the sequential software version of the algorithm, although it is considerably more energy efficient, since it runs at a much lower clock frequency (average of 1.9Mhz vs 3.4GHz). They also suggest that the synchronization methods employed to control the speculative execution are too expensive when managed by software, but efficient if managed by hardware.

1. INTRODUCTION

One of the most fundamental principles in our world is the search for an optimal state [1]. In applied mathematics and numerical analysis, this is often called optimization, i.e., the process of trying to find the best possible elements $\mathbf{x}^\star$ in $\mathbb{X}$ in such a way that an objective function F($\mathbf{x}^\star$) is either maximized or minimized, depending on the target goal. Factors such as discontinuity and multiplicity of both local maxima and minima increase the complexity of the problem.

One particular class of optimization is unconstrained optimization. The goal is to locate a minimizer $\mathbf{x}^\star$ of a given (nonlinear) function $f : \mathbb{R}^n \to \mathbb{R}$. If f is nonsmooth or even discontinuous at some points in $\mathbb{R}^n$, the optimization method should only use the function values of f, since the derivatives of f may not exist for a particular point. Methods within this category are usually called Direct Search Methods (DSMs). Some of these derivative-free methods have been proposed in the past decades [2], including Spendley-Hext-Himsworth's method [3], Powell's method [4] and the Nelder-Mead algorithm [5], the focus of this paper.

The Nelder-Mead algorithm [5] is one of the best known algorithms for multidimensional unconstrained optimization without derivatives [6]. Beyond solving the classical unconstrained optimization problem, it is also used to solve parameter estimation and similar statistical problems [7, 8, 9]. Libraries implementing the Nelder-Mead mainly differ on the used stopping criterion [10].

As the Nelder-Mead algorithm can become computationally expensive, especially for high *simplex* dimensions and discontinuous objective functions, effective utilization of hardware resources is mandatory if high performance is to be achieved. This motivates the implementation and assessment of parallel versions of the Nelder-Mead algorithm for multi-core CPU-chips and FPGAs.

The goals of this paper are: (i) to implement and assess novel software parallel versions of the Nelder-Mead algorithm for shared-memory multi-core chips, (ii) to evaluate the suitability of re-configurable logic for this algorithm and (iii) to compare the two approaches, when considering relatively simple (although hard-to-optimize) functions in $\mathbb{R}^2$. FPGAs are particularly suited for comparison since the execution flow of the software versions can be replicated in an FPGA design, therefore enabling the comparison of the efficiency of the fork-and-join synchronization by software and hardware. According to the best knowledge of the authors, there are neither disclosed parallel versions of the Nelder-Mead algorithm for shared-memory CPU-chips, nor implementations for FPGAs.

The main contributions of this paper are the following:

- The specification and assessment of a new parallel Nelder-Mead algorithm's version, which speculatively calculates the *simplex* basic operations in parallel;

- The implementation of the specified version, using two different methods for the creation and management of the fork-and-join mechanism that enables speculative execution;

- The assessment and comparison of the new parallel software Nelder-Mead's version with a sequential software version, both running on a last-generation CPU-chip, and two hardware designs, based on manual and automatic C-to-RTL synthesis.

The remainder of the paper is organized as follows. Section 2 describes the Nelder-Mead algorithm and analyzes its execution flow. Section 3 presents the implementation details of the devised software versions whereas the hardware implementation is covered in Section 4. Section 5 presents the results of the performed trials. Section 6 describes some related work and Section 7 concludes the paper.

2. THE NELDER-MEAD ALGORITHM

2.1. Algorithm's description

Nelder-Mead is an iterative algorithm that defines a *simplex* at each iteration. A *simplex S* in $\mathbb{R}^n$ is defined as the convex hull of *n*+1 vertices x_0,x_1,...,$x_n \in \mathbb{R}^n$. For instance, a *simplex* in $\mathbb{R}^2$ is a triangle (see Figure 1), and a *simplex* in $\mathbb{R}^3$ is a tetrahedron. At the beginning of each iteration k, the notation $S_k := < X_0, X_1, ..., X_n >$ represents the *simplex* with vertices ordered in an increasing manner with regard to their correspondent function values, i.e., $f(X_0) \leq f(X_1) \leq ... \leq f(X_n)$, where X_0 is the best vertex, X_1 is the second vertex and X_n is the worst vertex. Vertices represented by x_i, in the *simplex*, are not ordered in relation to X_i vertices.

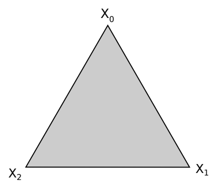


Fig. 1. An initial *simplex* in $\mathbb{R}^2$.

The Nelder-Mead algorithm is based on auxiliary points - candidates to be a vertex of the *simplex* defined at each iteration - accepted or rejected on the basis of a comparison between their correspondent function values and $f(X_0)$, $f(X_{n-1})$ and $f(X_n)$. To compute the auxiliary points, the version used in this work uses four operations, referred to as basic operations: reflection, contraction, expansion and shrinkage, associated with scalar parameters α, γ and β that satisfy $0 < \alpha \leq 1$, $\gamma >1$ and $0 < \beta < 1$.

At each iteration, the *centroid* $\bar{x}$ - the medium point of the hyperplane defined by $X_0, X_1, ..., X_{n-1}$ - is calculated as:

$$\bar{x} = \frac{1}{n} \sum_{i=0}^{n-1} X_i. \quad (1)$$

Once the *centroid* $\bar{x}$ is calculated, the algorithm calculates the reflection vertex x_r (see Figure 2) as:

$$x_r = (1 + \alpha) \times \bar{x} - \alpha \times X_n. \quad (2)$$

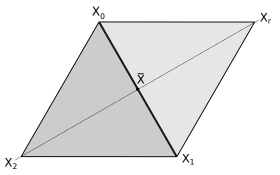


Fig. 2. Reflection vertex, in $\mathbb{R}^2$.

The quality of x_r is assessed with basis on the result of $f(x_r)$ and the function values of some other *vertices*, a test referred to as the reflection vertex test (RVT). In particular, x_r can be considered *very good*, *good*, *weak* or *very weak*. When x_r is considered *good*, that is, if $f(X_0) \leq f(x_r) < f(X_{n-1})$, no additional calculations are performed, the new vertex x_r is accepted and the *simplex* S_{k+1}, for the next iteration, is $< X_0, X_1, ..., X_{n-1}, x_r >$. In the remaining cases, additional steps are computed:

1. If x_r is considered *very good*, i.e., $f(x_r) < f(X_0)$, the algorithm performs an expansion of the *simplex*. The expansion vertex x_e is calculated (see Figure 3) as:

$$x_e = \gamma \times x_r + (1 - \gamma) \times \bar{x}. \quad (3)$$

If x_e is *very good*, i.e., $f(x_e) < f(X_0)$, x_e is accepted and $S_{k+1} :=< X_0, X_1, ..., X_{n-1}, x_e >$. Otherwise, x_r is accepted instead and $S_{k+1} :=< X_0, X_1, ..., X_{n-1}, x_r >$.

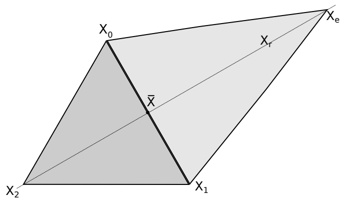


Fig. 3. Expansion vertex, in $\mathbb{R}^2$.

2. If x_r is considered *weak*, i.e., $f(X_{n-1}) \leq f(x_r) < f(X_n)$, the algorithm performs a contraction to the exterior. The outside contraction vertex $\hat{x}_c$ (see Figure 4) is calculated as:

$$\hat{x}_c = \beta \times x_r + (1 - \beta) \times \bar{x}. \tag{4}$$

If $\hat{x}_c$ is *good* ($f(\hat{x}_c) < f(X_{n-1})$), $\hat{x}_c$ is accepted and $S_{k+1} :=< X_0, X_1, ..., X_{n-1}, \hat{x}_c >$. Otherwise the *simplex* is shrunk. Shrinking the *simplex* consists of replacing each vertex X_i, for i=1,...,n, by the mean point x_i of the segment that connects X_i to X_0:

$$x_i = \frac{X_i + X_0}{2}. \tag{5}$$

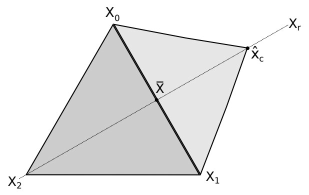


Fig. 4. Outside contraction vertex, in $\mathbb{R}^2$.

3. If x_r is considered *very weak*, i.e., $f(x_r) \geq f(X_n)$, the algorithm performs a contraction to the interior. The inside contraction vertex x_c is calculated as:

$$x_c = \beta \times X_n + (1 - \beta) \times \bar{x}. \tag{6}$$

If x_c is *good* ($f(x_c) < f(X_{n-1})$), x_c is taken and $S_{k+1} :=< X_0, X_1, ..., X_{n-1}, x_c >$. Otherwise, the *simplex* is shrunk and $S_{k+1} :=< X_0, x_1, ..., x_{n-1}, x_n >$.

In the proposed implementations, the parameters are set as follows: $\alpha = 1, \gamma = 2$ and $\beta = \frac{1}{2}$. The stopping criterion, tested at each iteration, verifies if the *simplex* size is smaller than or equal to a given amount $\epsilon > 0$ (≈ 0), i.e.:

$$\frac{1}{\Delta} \, max_{1 \leq i \leq n} \, ||X_i - X_0||_2 \leq \epsilon \tag{7}$$

where $\Delta = max(1, ||X_0||_2)$ and $\epsilon = 10^{-16}$.

If the stopping criterion is verified, the algorithm stops and the first vertex of the ordered *simplex*, X_0, is considered the final result, since it provides the best approximation to the minimizer of the objective function.

2.2. Execution-flow analysis

The Nelder-Mead algorithm tries to improve the *simplex* (and consequently its solution) at each iteration, based on the calculation of the centroid. The computational requirements of the algorithm become particularly high when a large number of iterations are required for convergence. No iterations can be processed in parallel, nor can an iteration be parallelized itself, unless it is based on applying a shrinkage to the *simplex*. However, this operation is too cheap to be worth computing it in parallel.

After initializing the process, the algorithm calculates the centroid, the reflection vertex and the RVT, which assesses the reflection vertex and determines which operations must be applied to the *simplex*. When the reflection vertex is considered *good*, only the *simplex* is updated, which involves low overhead. When the reflection vertex is considered *weak*, an outside contraction is performed. For the *very weak* case, the algorithm performs an inside contraction. When the reflection vertex is considered *very good*, an expansion is done instead. Except when the reflection vertex is considered *good*, which requires no further work, every case consists of a basic operation and an associated test, as described in Section 2.1.

As the complexity of the contraction (either to the exterior or to the interior) and the expansion basic operations require the same number and type of operations to be performed, their associated tests determine how computationally expensive each case is. In that regard, classifications are ordered as *good* < *very good* ≤ *weak* ≤ *very weak*, from the least to the most expensive. As an exception of this classification, the *weak* case might be more expensive than the *very weak* case, if a shrinkage is required in the *weak* case but not required in the *very weak* one. The exact cost of each iteration depends primarily on the objective-function and, at a lesser extent, on the stage of the *simplex*.

3. SOFTWARE IMPLEMENTATIONS

A sequential version of the algorithm was implemented in line with the algorithm description in Section 2.1. Two parallel versions were developed to calculate the basic operations in a thread-level speculative execution fashion (see e.g., [11]). All code is written in C and parallel versions are implemented with pThreads. Although OpenMP provides a more convenient programming interface, it does not allow task killing, a functionality one of the proposed versions depends on.

Similarly to branch prediction in computer architecture, but for a broader number of cases, all basic operations are computed before the result of the RVT is known. In each iteration, the RVT calculation is overlapped with the parallel speculative computation of the basic operations, referred to as "decision paths", i.e., *good, very good, weak* and *very weak*. This is expected to boost performance when several iterations are required to convergence, as long as each computation is executed on a different core. A process p runs the algorithm and manages thread creation, destruction and synchronization. As soon as the result of the RVT is known, one of the four computations is committed (process p might need to wait for its completion), whereas the others are discarded.

The implemented parallel versions differ on how they discard unnecessary computation. In both versions, four threads are created by process p - one per decision path - each maintaining a local, auxiliary, *simplex*. When created, threads immediately block on a private condition. The *simplex* used by the algorithm, i.e., by process p, is referred to as global *simplex*, also because it is visible to all threads.

At the beginning of each iteration, conditions are signaled and threads released, so they can copy the global *simplex* to their local *simplices*, with which they perform their respective operations. When threads are released, local *simplices* are coherent copies of the global *simplex*. At each iteration and depending on the result of the RVT, one thread is marked as *valid* whereas the remaining are marked as *invalid*. Also at each iteration, process p copies the local *simplex* of the *valid* thread to the global *simplex*, which might require waiting for the completion of the thread.

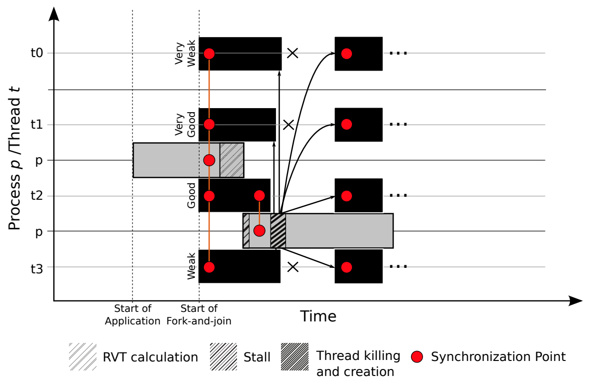


Fig. 5. Execution flow of the first iteration of a sample run in the first parallel version of Nelder-Mead, where the reflection vertex is considered *good*. Qualitative plot, not drawn to scale.

In the first version, referred to as "thread killing" version, process p waits for the *valid* thread until the result of the RVT is known, copies its *simplex* and kills the remaining threads. As each thread performs a decision path and dies, process p waits for them with the `pthread_join` primitive. In order to kill *invalid* threads, a `SIGSEGV` signal is send (using the `pthread_kill` primitive), forcing threads to finish. As soon as the local *simplex* is copied, four new threads are created (with `pthread_create`) and the process continues. As stated earlier, threads are released in the beginning of the subsequent iteration, after testing the stopping criterion.

The second version, referred to as "persistent threads" version, was motivated by the high cost of thread destruction and creation, present at each iteration. Unlike the first version, the second version does not kill and create new threads to maintain the fork-and-join mechanism, relying on four threads persistent across the whole application's lifetime and synchronization methods instead.

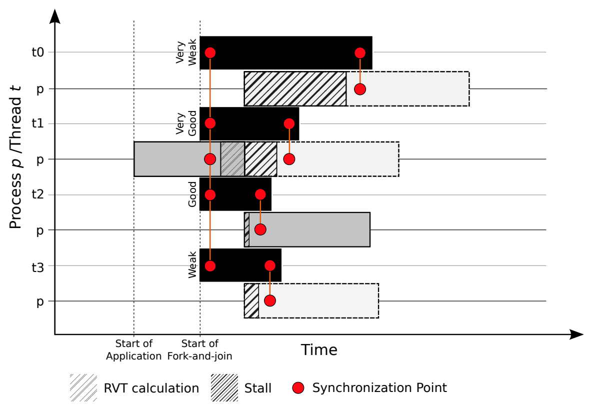


Fig. 6. Execution flow of the first iteration of a sample run in the second parallel version of Nelder-Mead, where the reflection vertex is considered *good*. Behavior of *process p* shown in light gray for remaining cases. Qualitative plot, not drawn to scale.

In the second version, process p has a set of four pThread conditions in which it blocks as a means of waiting for the completion of the thread. Process p uses these conditions to wait for the valid thread, by blocking on the correspondent one until the valid thread awakes it by releasing the condition. When this happens, the valid thread blocks itself on its condition until the next iteration starts (similarly to the "thread killing" version). As the remaining threads might still be computing their associated basic operations, process p resets them, by changing a variable read by every thread. Changing this variable is a lock-free operation. As a result, invalid threads might execute (a few) additional operations, very likely overlapped with the end of the iteration, which includes the stopping criterion calculation. Threads execute their respective operations on two nested loops where the outermost loop is executed indefinitely. When the "stopping" variable changes, threads break the inner-loop, which is executed again, since the outermost loop is infinite.

4. HARDWARE IMPLEMENTATION

4.1. RTL Nelder-Mead Implementation

The Nelder-Mead algorithm was implemented on a Xilinx Virtex-7 FPGA. Every variable (e.g., vertices and intermediate values used for calculations) is implemented in both the 32 and 64-bit floating point representation defined by the IEEE 754-2008 standard, and all operations are performed using the embedded FPGA DSP blocks. The system implements input ports to set initial *simplex* values and output ports to display the result, as well as input/output control/status signals, e.g., start operation, calculation done. In order to maximize processing speed, the implementation attempted to parallelize computations as much as possible. Hence, the system is composed of four execution paths computing concurrently, in a similar fashion to the speculative execution in software versions, which determine the next iteration value for *good*, *very good*, *weak* and *very weak* vertices, respectively. Common operations to all execution paths at the beginning of each iteration, such as the vertex sorting, are executed by a single module which propagates results to the execution paths. The evaluation function multiplexes which computation path's output is written back to the *simplex* registers. A simplified block diagram of the system is depicted on Figure 7.

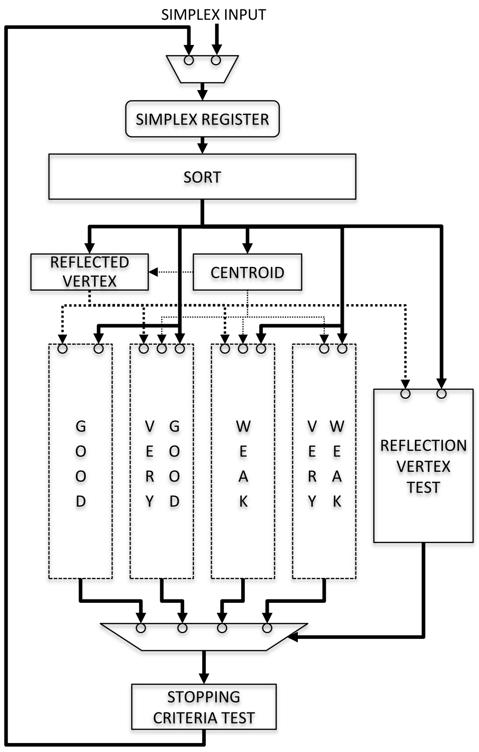


Fig. 7. Simplified block diagram of the FPGA Nelder-Mead design.

Each execution path is exclusively composed of combinational logic. At each clock cycle, the new *simplex* value is updated on the *simplex* register. Following this approach, each clock cycle corresponds exactly to one iteration of the algorithm. This is motivated by the fact that, albeit each computational path can be executed in parallel, there is no possible parallelism between sequential iterations, i.e., each iteration requires the final value of the previous one to begin computation. This approach is highly area-demanding, since functional units that are used by several paths (such as the module which implements the objective function) must be replicated in order to allow concurrent computation. If multi-cycle execution was performed, replication of functional units could be avoided, but the overall execution time would increase since execution paths would have to be stalled in case of concurrent access. As this work is focused on studying and developing high performance Nelder-Mead versions, performance-area trade-offs fall outside the scope of this research. As previously mentioned, the function to optimize is also implemented on-chip. This approach requires re-synthesis whenever a new function is desired, but offers the highest performance. If the function was implemented in a more flexible way (e.g. off-chip), communication delays would decrease performance [12].

Table 1 presents the resource utilization rates for every implemented function. The design does not use BRAMs or DFFs. DSP blocks, on the other hand, are considerably used, especially in the 64-bit version, due to the wide number of 64-bit floating point arithmetic operators in the algorithm.

The design was implemented using tools from the Xilinx's Vivado 2012 Suite, namely Xilinx ISE 14.3 for design and implementation, including mapping, placing and routing, and Xilinx ISIM 14.3 for simulation.

Table 1. FPGA synthesis results. f represents frequency.

Function	1	2	3	4	5	6
			32-bit			
f (MHz)	3.591	3.582	3.730	2.821	2.380	2.701
Registers	<1%	<1%	<1%	<1%	<1%	<1%
LUTs	4.3%	4.5%	3.3%	9.0%	9.8%	9.6%
IOBs	40.1%					
DSPs	28.5%	24.8%	18.1%	27.0%	20.3%	30.7%
			64-bit			
f (MHz)	2.147	2.148	2.123	1.880	1.398	1.713
Registers	<1%	<1%	<1%	<1%	<1%	<1%
LUTs	13.1%	15.0%	10.8%	16.9%	9.8%	18.4%
IOBs	80.2%					
DSPs	87.9%	59.0%	49.4%	87.9%	58.9%	97.5%

Although the maximum operating frequency may appear low for a hardware implementation (an average of 1.9MHz for the 64-bit version), the system performs one algorithm iteration per clock cycle, thus yielding very short execution times. As previously mentioned, there would be no actual performance gain from introducing intermediate registers for multi-cycle operation at higher frequencies. The current operating frequency is due to the large number of floating point arithmetic modules cascaded on the datapath. Using 64-bit precision also contributes to the low frequency; this design decision was motivated by wanting to achieve completely equivalent hardware and software implementations (i.e., hardware registers are equivalent to software *double* type variables). The 32-bit implementation results in a much more area-efficient implementation and provides moderate performance improvements. The smaller resolution showed equivalent results for five functions, failing only on one function, where for certain inputs, the difference in resolution yielded different results, albeit approximate. Experiments up to date indicate that 32 bit resolution is sufficient for most function optimizations using the Nelder-Mead algorithm, thus future work will encompass reducing resolution in order to further increase performance, including using alternative encodings for increased throughput [13]. Performance results were obtained through Xilinx's ISIM timing (post place and route) simulation.

4.2. Manual vs automatic hardware generation

The software version of the Nelder-Mead algorithm was translated to hardware through the Xilinx HLS C-To-RTL generation tool. This approach presented worse results than the *ad hoc* implementation, due to several reasons:

i) The tool unrolls only a few loops. The majority of computations are implemented as multi-cycle hardware paths, probably due to the wide number of function calls on several loops' iterations.

ii) Albeit the C-To-RTL version yields much higher operating frequency than the manual implementation, at the cost of multi-cycle algorithm iterations, this results in no performance increase. This is due to the fact that subsequent Nelder-Mead iterations cannot be parallelized, therefore there is no gain from a pipelined implementation. The higher number of intermediate registers increases the setup-hold times in the critical path, resulting in an overall slower execution.

Software version could be altered in order to achieve better results in automatic synthesis, using code re-factoring techniques encompassing hardware synthesis estimation [14], but no such experimentation was performed and is postponed to future work.

5. RESULTS

The developed software versions were tested on a last generation CPU whereas the hardware versions were simulated on a last generation FPGA. The characteristics of the tested platforms are summarized in Table 2. Software versions were compiled with `gcc -O3`. The execution time was measured with the OpenMP `omp_get_wtime()` flag.

Six hard-to-optimize functions in $\mathbb{R}^2$, presented in Table 3 and known for having multiple local optima, were chosen as case studies for benchmarking.

Table 3 also shows the initial *simplices* used for each function. In particular, F_3 is known as Rosenbrock's function, F_4 is a derivative of Griewank's function, F_5 and F_6 are respectively known as Schwefel and Rastringin functions. The algorithm was limited to one hundred thousand iterations for every function and every version, both in the CPU and in the FPGA, the number of iterations taken by every function due to the strict stooping criterion. The execution times presented for each trial (optimization process of one function) on the CPU is the mean of five runs, whereas the hardware simulations are 100% accurate and have not been, therefore, statistically treated.

Table 2. Test platform specifications. iC and dC stand for instruction and data cache, respectively.

Device	CPU	FPGA
Manufacturer	Intel	Xilinx
Brand	Core 5 Ivy Bridge	Virtex 7
Model	i5-3570K	XC7VX485T
Max clock	3.4 GHz	300 MHz
Cores	4	-
System mem	16 Gbytes	68 Mbytes
L1 Cache	32kB iC+dC/core	-
L2 Cache	256kB/core unified	-
L3 Cache	6MB shared unified	-
Year	2012	2012

5.1. CPU

The run time of the implemented software versions is shown in Figures 8(a), 8(b) and 8(c). Both parallel versions are slower than the sequential one, due to the fork-and-join mechanism's overhead: either by creating and destroying threads in the first version or by thread synchronization in the second version. Not surprisingly, thread synchronization is more efficient than kill and create threads at each iteration, but it is still slower than the sequential version.

While trials with functions in $\mathbb{R}^2$ show that the overhead of the speculative execution mechanism kills any speedup, more complex functions and functions in higher dimensions, will mitigate the impact of the fork-and-join mechanism, highly noticeable since calculating the used functions is a very quick process.

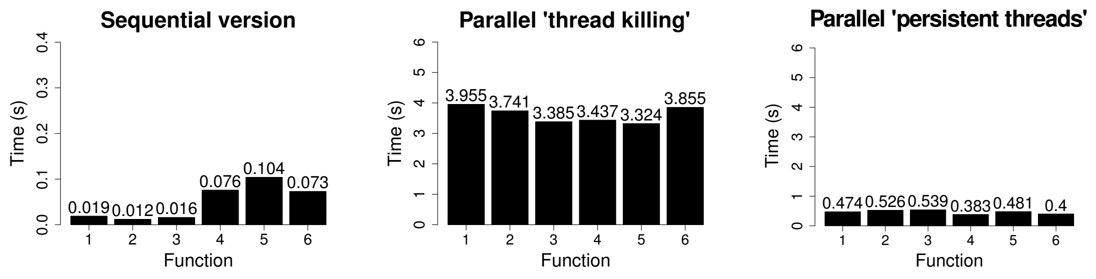
CPU versions are optimized, mostly due to `-O3` gcc flag, and have less than 1% of L1 cache misses, as reported by Cachegrind tool, in essence because *simplices* data structures fit on cache.

5.2. FPGA

Figure 9 shows the results of the FPGA, both for the C-to-RTL version, in 9(a), and for the handwritten design in 9(b).

Table 3. Tested functions and their respective inputs.

Function	Input Simplex
$F_1(x,y) = -40000x - 60000y + 5x^2 + 10y^2 + 10xy$	$S = \ < (0.99, -0.34), (0.61, 1.39), (1.05, -1.895) >$
$F_2(x,y) = 20000 \times ((x\ +\ 70)^2 + (y+275)^2) + y^2 + (y+195)^2$	$S = \ < (234.55, 8.32), (23.343, 34.33), (0.992, 2.23) >$
$F_3(x,y) = 100(y - x^2)^2 + (1-x)^2$	$S = \ < (0.081, 0.912), (92.2, 0.21), (18.11, 0.01) >$
$F_4(x,y) = x \times y \times 0.7623 + \frac{1}{4000} \times \prod_{i=1}^{i=2} x_i^2 - \sum_{i=1}^{i=2} cos(x_i)$	$S = \ < (10.23, 0.16), (1.24, 0.7), (0.1, 0.1) >$
$F_5(x,y) = 418.9820 \times 2 - \sum_{i=1}^{i=2} x_i \times sin(\sqrt{\lvert x_i \rvert})$	$S = \ < (2.234, 0.832), (1.118, 304), (1.999, 0.354) >$
$F_6(x,y) = 20 + ((x^2 - 10cos(2\pi x)) + y^2 - 10cos(2\pi y))$	$S = \ < (20.667, 340.832), (1.2318, 200), (10.54, 0.7354) >$

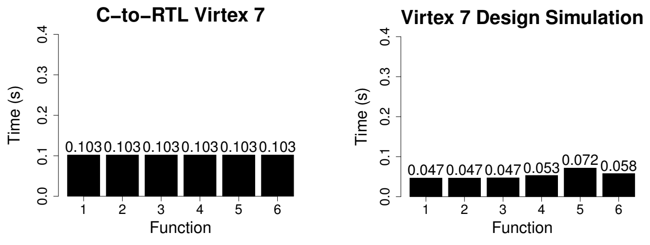


(a) Sequential version (b) Parallel v. w/ thread killing (c) Parallel v. w/ persistent threads

Fig. 8. Total runtime, in seconds, for the devised Nelder-Mead CPU versions, for the six presented objective functions.

The latter was implemented with Verilog, according to the description in Section 4.

As shown in Figures 8 and 9, the FPGA implementation offers higher and lower performance than the equivalent software implementation, depending on the tested function. However, the FPGA runs at a considerably lower clock frequency (average of 1.9MHz of FPGA vs. 3.4GHz of the CPU). As the design replicates the parallel software versions, one can also conclude that the FPGA performance is not affected by synchronization issues, which are guaranteed by the used frequency.



(a) C-to-RTL version (b) Verilog simulation

Fig. 9. Total runtime, in seconds, both for a C-to-RTL version and a 64-bit Verilog simulation of Nelder-Mead.

6. RELATED WORK

A parallel version of the Nelder-Mead algorithm was proposed earlier, using parallelization at the parameter level [15]. However, the parallel version has a different search path though the parameter space than the non-parallel algorithm, in contrast to this paper. Moreover, the approach relies on even finer grained parallelism than the presented approach, thus likely unsuited for multi-core CPU-chips.

Both Nelder-Mead and Powell's methods were globalized and parallelized on a distributed memory environment with six single-core Pentium 4 machines running at 2.8 GHz, following a server-client fashion [16]. This paper, on the other hand, proposes parallel shared-memory and hardware implementations.

A method for concurrent execution of the algorithm has also been proposed [17], with better results than the Nelder-Mead original algorithm on smooth, noisy, and functions with many local minima. This paper is rather focused on the original algorithm than on variants of it.

7. CONCLUSIONS

This paper introduced a novel parallelization of the Nelder-Mead algorithm, to work on shared-memory multi-core CPU-chips. It is based on speculatively executing the operations to apply to the *simplex*, overlapping them with the RVT calculation to therefore boost performance, by raising resource usage. This technique was employed with success in two different ways.

Trials with a 2D implementation of these versions, running on a quad-core CPU-chip, showed that even though speculative execution is applicable, performance is degraded due to (i) thread creation, destruction and synchronization costs to manage the fork-and-join mechanism that maintains the speculative execution and (ii) small computation overlap between the RVT and decision paths. As a result, this approach is not profitable, unless the objective functions take more time to compute, thus reducing the relative communication overhead.

Re-configurable logic has shown to deliver similar performance to the CPU, but at a much lower clock frequency and, consequently, energy efficiency. The FPGA design also calculates all the four decision paths in parallel, but the execution time is as long as the longest path. This suggests that synchronization for fork-and-join mechanisms would be considerably more efficient if implemented by hardware. The results of a C-to-RTL version, converted with basis on the sequential software version, showed that automatic conversion is less efficient for this particular algorithm, especially due to the low resource utilization, a common handicap of automatic synthesis.

The performance of both devices is strictly related with the characteristics of the performed trials. FPGA designs benefit from the use of both simple objective functions and small dimensions. While complex objective functions would favor the CPU's parallel proposed variant, due to bigger computation overlaps between the RVT and basic operations, higher *simplex* dimensions would require more area in the FPGA, already at ≈90% utilization rates for DSPs, in some cases. At some point, area would be completely used and performance would be decreased. Benchmarks of both more complex objective functions and higher *simplex* dimensions are scheduled for future work, as well as the assessment of other methods to parallelize the algorithm by software.

8. ACKNOWLEDGMENTS

The authors would like to thank Edite Fernandes, Christian Bischof and Michael Burger for their constructive comments and suggestions aimed at improving the paper.

9. REFERENCES

[1] T. Weise, *Global Optimization Algorithms - Theory and Application*, 2nd ed. Self published, May, 2009.

[2] R. M. Lewis, V. Torczon, and M. W. Trosset, "Direct Search Methods: Then And Now," *Journal of Computational and Applied Mathematics*, vol. 124, pp. 191–207, 2000.

[3] W. Spendley, G. R. Hext, and F. R. Himsworth, "Sequential Application of Simplex Designs in Optimization and Evolutionary Operation," *Technometrics*, vol. 4, pp. 441–461, 1962.

[4] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, no. 2, pp. 155–162, January 1964.

[5] J. A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *The Computer Journal*, vol. 7, no. 4, pp. 308–313, January 1965.

[6] S. Singer and J. Nelder, "Nelder-Mead algorithm," *Scholarpedia*, vol. 4, no. 2, p. 2928, 2009.

[7] T. G. Kolda, R. M. Lewis, and V. Torczon, "Optimization by direct search: New perspectives on some classical and modern methods," *SIAM Review*, vol. 45, pp. 385–482, 2003.

[8] M. J. D. Powell, "Direct Search Algorithms for Optimization Calculations," *Acta Numerica*, vol. 7, pp. 287–336, 1998.

[9] M. H. Wright, "Direct Search Methods: Once Scorned, Now Respectable," in *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, ser. Pitman Research Notes in Mathematics, D. F. Griffiths and G. A. Watson, Eds., vol. 344. Boca Raton, Florida: CRC Press, 1996, pp. 191–208.

[10] S. Singer and S. Singer, "Efficient Implementation of the NelderMead Search Algorithm," *Applied Numerical Analysis & Computational Mathematics*, vol. 1, no. 2, pp. 524–534, 2004.

[11] H. K. Pyla, C. Ribbens, and S. Varadarajan, "Exploiting coarse-grain speculative parallelism," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 555–574.

[12] K. Kanazawa and T. Maruyama, "An fpga solver for sat-encoded formal verification problems," pp. 38–43, 2011.

[13] F. de Dinechin, M. Joldes, B. Pasca, and G. Revy, "Multiplicative square root algorithms for fpgas," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 574–577.

[14] A. Cilardo, P. Durante, C. Lofiego, and A. Mazzeo, "Early prediction of hardware complexity in hll-to-hdl translation," in *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, 2010, pp. 483–488.

[15] D. Lee and M. Wiswall, "A Parallel Implementation of the Simplex Function Minimization Routine," *Comput. Econ.*, vol. 30, no. 2, pp. 171–187, September 2007.

[16] A. Koscianski and M. A. Luersen, "Globalization and Parallelization of Nelder-Mead and Powell Optimization Methods," *Innovations and Advanced Techniques in Systems, Computing Sciences and Software Engineering*, pp. 93–98, 2008.

[17] A. Lewis, D. Abramson, and T. Peachey, "RSCS: a parallel simplex algorithm for the Nimrod/O optimization toolset," in *Parallel and Distributed Computing, 2004. Third International Symposium on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004.*, July 2004, pp. 71–78.