Parallel Function Optimisation Using Evolutionary Algorithms and Deterministic Neighbourhood Search

Ioannis Zgeras, Jürgen Brehm, Andreas Reisch

Institut für System- und Rechnerarchitektur
Leibniz Universität Hannover
Appelstrae 4
30167 Hannover
zgeras@sra.uni-hannover.de
brehm@sra.uni-hannover.de

Abstract: Modern computer hardware provides massive computational power by parallelism. However, many of the existing algorithms and frameworks are optimised for sequential execution and are not capable to be parallelised or do not scale well on complex parallel architectures. In our paper, we present a metaheuristic consisting of a parallel Evolutionary Algorithm and a parallel Neighbourhood Search. For the implementation massively parallel GPUs are used. This framework is evaluated on the application of function optimisation.

1 Introduction and State of the Art

Traditional programs do not run faster on a multi-core processor than on a single core CPU without exploiting parallelism. The tradi- tional programming paradigms do not address the new hardware architectures and thus have to be modified or extended. Another upcoming trend is to use Graphics Processing Units (GPUs) as highly parallel functional units to speed up complex operators not only from the area of graphical computing. To use these powerful functional units efficiently, the corresponding programs have to be tailored to the hardware.

In this paper, we present a parallel hybrid function solver using Evolutionary Algorithms (EA)[ES08] as exploration unit and a deterministic Neighbourhood [HMM10] Search for exploitation issues. By using Evolutionary Algorithms we ensure robustness with respect to varying functions. Both parts of the function solver are running on GPUs. Our approach simplifies the exploiting of current parallel hardware by offering the programmer a black box like behaviour. The only task a developer using this framework has is defining the function he wants to solve and intervals for the solutions if applicable.

There are several implementations of biologically inspired algorithms using parallelisation methods. General parallelisation strategies of EAs are discussed in [AC02]. In [PADc], the authors parallelise parts of an EA and distribute it over a set of connected computers to achieve an acceleration. In [PMBA06], the authors use a Neighbourhood Search algorithm

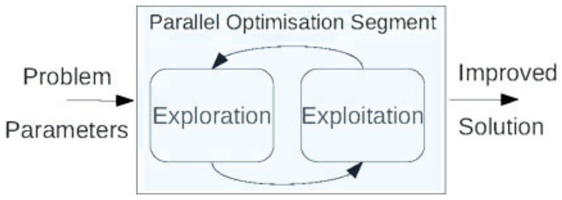

Abbildung 1: Architecture overview

to improve an initial solution for optimisation tasks. In [MBS07], PSO is parallelised by splitting the ant swarm into subswarms. These subswarms search the solution space in different regions. By splitting the swarm into subswarms, communication overhead is reduced. This approach is suitable for grid-computing environments whereas our work focuses on optimising the algorithms for multi-core architectures. A parallel implementation of Variable Neighbourhood Search is introduced by Crainic et al. in [CGHM04]. To the best of our knowledge, there is only one similar approach using a hybrid implementation of Evolutionary Algorithms and Neighbourhood Search algorithms on a GPU in [LMT10]. Here, Neighbourhood Search is only used to replace the crossover and mutation operations of EA.

2 Framework

This Section describes the architecture of our framework in a generic way (Sec. 2). Furthermore, we describe the use of the framework within the problem of function optimisation in Sec. 2.

The architecture, as shown in Figure 1, uses a black box like approach. The so called Parallel Optimisation Segment (POS) consists of two main parts, an exploration and an exploitation part. Both components have complementary strengths and weaknesses. Whereas the exploration part is responsible for searching the entire search space, the exploitation part improves the encountered solutions by the exploration part.

In every iteration within the POS initial solutions are found by the exploration part, improving the solutions from past iterations. These solutions are passed to the exploitation part, where fine granular optimisation is performed.

Function Optimisation: The task of function optimisation is to find a global maximum or minimum of a given function. The described framework is applied on the problem of function optimisation. Only the individuals used by the EA and the operations of EA and Neighbourhood Search have to be adapted and the input has to be determined. The framework input consists of three parts. The function to be solved and the intervals, if known, within the single values of the solution may vary. At last, the dimensions of the function have to be known by the framework.

Evolutionary Algorithm adaptations: A small number of adaptations of the EA have to be applied to fit the problem space of function optimisation. First of all, a fitness function has to be determined. In case of function optimisation, the fitness function is the same as the function that is optimised. Afterwards, the representation of the individuals has to be declared. An individual consists of an array of values (solution array), which describe the solution found by this individual, and a fitness value which is the computed value applying the solution array on the function which is to be optimised.

Crossover and mutation also have to be adapted. In case of crossover, positions of the solution array are simply swapped. Mutation is performed by changing the corresponding position of the solution array to a random value within a predefined range.

Neighbourhood Search adaptations: To apply Neighbourhood Search on function optimisation, the method of improving the passed solutions by the EA has to be adapted. As already mentioned, there is no random component used in the exploitation part, all operations are deterministic and depend on the id of the thread and the corresponding block. Every iteration of the Neighbourhood Search consists of the following steps:

1. Make a local copy of the working individual, so that each thread of the block has the same copy.
2. For every position of the individual: Search the neighbourhood for better solutions. This is accomplished by the following formula:

   $$s_{new} = s_{old} \pm \frac{threadId + 1}{\#threads} * \frac{\alpha * (blockId + 1)}{\#blocks} \tag{1}$$

   Where s_{new} is the new solution, s_{old} the old solution and α a constant to adjust the search space. The smaller α is, the smaller is the search space for Neighbourhood Search.
3. The last iteration step is dedicated to synchronise all found solutions. For this purpose, reduction kernels are used in CUDA. Reduction kernels are usually used to iteratively determine the optimum of a set of values saved in an array structure. In this case, two reductions have to be done. Firstly, the best solution of each block has to be determined. Secondly, a synchronisation between the block populations has to be performed.

3 Evaluation

This Section presents our first proof of concept evaluation results. We have used the following sum function described in [YL96, Ves04] as a benchmark function: $f(\vec{x}) = \sum_{i=0}^{n-1} x_i^2$ with $-5.12 \leq x_i \leq 5.12$. The task is to find a valid value assignment to find the optimum (minimum value: 0.0).

We have compared two versions using a metaheuristic with an EA only version. The impact of constant memory, both in terms of solution quality and speed was evaluated. The first implementation is the metaheuristic consisting of an EA and a Neighbourhood Search running both on the GPU. Thereby, the Neighbourhood Search is only triggered if the EA does not provide better solutions than the previous iterations. The second implementation is similar to the first one with the only difference of using the constant memory of the GPU. The third implementation is a simple EA running on the GPU as described in Section 2.

As parameter for function dimensions, different values are selected (30,100 like in [YL96]) to compare the different approaches. Due space constraints, we only show the results of the evaluation with 30 dimensions. Note that the y-axis in the figures is logarithmic for better overview.

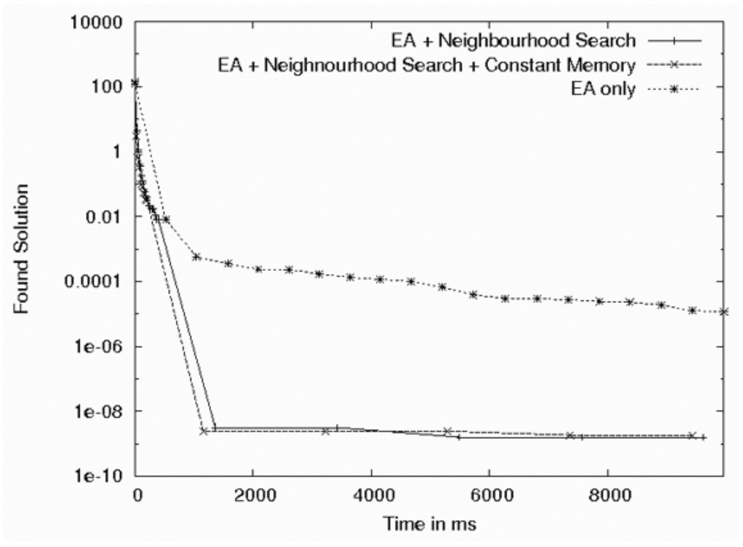

Abbildung 2: Results using 30 dimensions

In Figure 2, the results of the function initialised with 30 dimensions are shown. All three implementations have the same progression till $\approx$ 800ms. After then, there is a great gap between the optimisation of the function by using just EA and EA together with Neighbourhood Search. The implementation with the use of constant memory is slightly faster than without constant memory while they find the similar solutions. The metaheuristic implementation do not find the optimal solution (that is 0.0). The solution quality depends on the choice of the parameter α. By choosing a smaller value for α, we would achieve a better final solution, but there would be more optimisation steps necessary.

4 Conclusion and Future Work

In this paper, we presented a new approach for a highly parallel computational framework aiming at today's hardware and biologically inspired algorithms to speed up computation by exploiting inherent parallelism. We have presented a hybrid version of parallel Evolutionary Algorithm and parallel Neighbourhood Search running both on GPUs. Furthermore, we have presented an implementation using constant memory to speed up the Neighbourhood Search. Our first evaluation results show an advantage of the hybrid version towards the EA only version, in particular when using constant memory.

In our future work we plan to optimise both, the EA and the Neighbourhood Search, to speed up the function optimisation. The main bottleneck of Neighbourhood Search is the amount of function evaluations done by single threads. Furthermore, we plan to implement an adaptive version of the framework with automatic changeable parameters. By this, we hope to achieve better adaptivity of our framework to new functions and better solutions (e.g. by continuous decreasing of the parameter α and mutation rate). Moreover, we plan more detailed evaluations by using a larger set of functions and dimensions to approve the adaptivity of our framework in general. While our initial framework exploits the parallelism of GPUs, the parallelism of CPUs is not treated. We are working currently on a multi-CPU implementation.

Literatur

[AC02] Enrique Alba and Carlos Cotta. *Parallelism and evolutionary algorithms*, volume 6. 2002.

[CGHM04] Teodor Gabriel Crainic, Michel Gendreau, Pierre Hansen, and Nenad Mladenovi. Cooperative Parallel Variable Neighborhood Search for the p-Median. volume 10, pages 293–314. Springer Netherlands, 2004.

[ES08] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing (Natural Computing Series)*. Springer, October 2008.

[HMM10] Pierre Hansen, Nenad Mladenovic, and Jose MorenoPerez. Variable neighbourhood search: methods andapplications. *Annals of Operations Research*, 175:367–407, 2010.

[LMT10] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Parallel Hybrid Evolutionary Algorithms on GPU. 2010.

[MBS07] Sanaz Mostaghim, Jürgen Branke, and Hartmut Schmeck. Multi-Objective Particle Swarm Optimization on Computer Grids. pages 869–874, 2007.

[PADc] M. Parrilla, J. Ar, and S. Dormido-canto. Parallel Evolutionary Computation: Application of an EA to Controller Design.

[PMBA06] José Pérez, Nenad Mladenović, Belén Batista, and Ignacio Amo. *Variable Neighbourhood Search*. 2006.

[Ves04] Jakob S. Vesterstrøm. A Comparative Study of Differential Evolution, Particle Swarm Optimization, and Evolutionary Algorithms on Numerical Benchmark Problems. 2:1980–1987, 2004.

[YL96] Xin Yao and Yong Liu. *Fast Evolutionary Programming*. MIT Press, 1996.