Proceedings of the Federated Conference on Computer Science and Information Systems pp. 865–872
DOI: 10.15439/2016F255
ACSIS, Vol. 8. ISSN 2300-5963

Efficient parallel execution of genetic algorithms on Epiphany manycore processor

Łukasz Faber, Krzysztof Boryczko
AGH University of Science and Technology
al. Mickiewicza 30, 30-059 Kraków, Poland
E-mail: {faber,boryczko}@agh.edu.pl

***Abstract*—Recent years have seen a growing trend towards the introduction of more advanced manycore processors. On the other hand, there is also a growing popularity for cheap, credit-card-sized, devices offering more and more advanced features and computational power.**

In this paper we evaluate Parallella – a small board with the Epiphany manycore coprocessor consisting of sixteen MIMD cores connected by a mesh network-on-a-chip. Our tests are based on classical genetic algorithms. We discuss some possible optimizations and issues that arise from the architecture of the board. Although we achieve significant speed improvements, there are issues, such us the limited local memory size and slow memory access, that make the implementation of efficient code for Parallella difficult.

I. Introduction

FOLLOWING the Manycore Revolution [1] and the popularity of small integrated, power consumption- and cost-oriented computing boards (for example, Raspberry Pi), it was expected that these two "directions" would merge at some point. One of the results of this "merge" is the Parallella board[1] [2], created by Adapteva. It is a small (credit card-sized) board, comprising of a 16-core Epiphany coprocessor and the main dual-core ARM processor.

Such boards are interesting for researchers due to their costs, simplicity, and the low level requirements for beginning work with them. Parallella has the benefits of being a standalone, plug-and-play coprocessor similarly to Intel Phi [3]. In this case, "standalone" means that it is a completely separate computer unit that can run independently of any other nodes. On the other hand, it is plug-and-play, because it only requires an Ethernet cable to connect to it.

In this paper we want to review the Parallella board as a simple and effective tool for implementing strictly computational systems. We do not expect the platform to provide higher performance than well established manycore architectures like GPGPU. However, it seems that the programming model and achievable efficiency are *good enough* for creating quick, low-cost hardware-accelerated parallel platforms for simulations and computations. We want to show that it is feasible to implement various execution strategies on the platform and demonstrate its possible weaknesses.

The work reported in this paper concentrates on the realization of genetic and evolutionary algorithms on the Parallella board. It is related to and extends our previous publications regarding the implementation of effective tools for running population-based computational intelligence systems [4], especially using the agent paradigm [5], [6] in both parallel and distributed [7], as well as heterogeneous environments [8].

In the following sections, we briefly introduce the Parallella platform and Epiphany manycore (Section II) and its memory and programming models. Then, in Section III, we discuss our benchmarks and introduced optimizations. In Section IV we present the results. Finally, these results are discussed in Section V, alongside introducing the next steps we are taking with Parallella.

II. Parallella

Parallella is a hardware platform built on top of the manycore Epiphany [2] coprocessor, created by Adapteva in 2011. It was funded by a Kickstarter campaign[2] The board was first presented in June 2014.

The Epiphany processor consists of a 2D array of nodes (known as "eNodes") connected by a mesh network-on-a-chip. Each node consists of a single RISC core (called "eCore"), a DMA engine, 32 kB of memory and a network interface. Each core includes a 32-bit floating point RISC CPU, local memory, a DMA engine, an event monitor and a network interface. The mesh connections on a 16-core processor are shown in Figure 1.

Each "eCore" contains a floating-point unit (FPU), an arithmetic logic unit (ALU) and a 64-word register file, as shown in Figure 2.

The address space in Epiphany is flat and consists of 2^{32} bytes. Each node has 32 kB of its own local range of memory aliased in addresses `0x0000–0x7FFF`. However, memory of each node can be accessed by prefixing the address with a globally addressable ID consisting of 6 bits for a row ID and 6 bits for a column ID (counted from 0) – thus giving a theoretical maximum of $64 \times 64 - 1 = 4095$ cores. For example, if a core wanted to access the memory of the core located in the second row and the third column (1,2) it would access addresses `0x04200000–0x04207FFF`.

Some specifics of the Epiphany architecture related to eMesh include:

[1]https://www.parallella.org/

[2]https://www.kickstarter.com/projects/adapteva/parallella-a-supercomputer-for-everyone

978-83-60810-90-3/$25.00©2016, IEEE

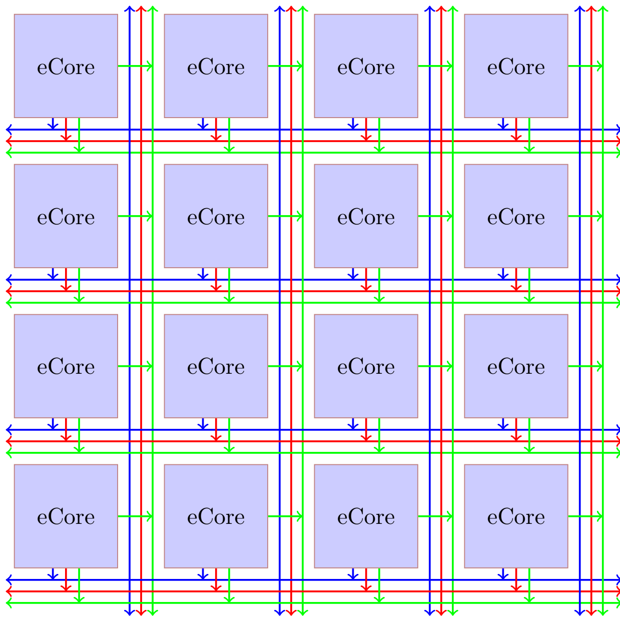

Fig. 1. eMesh Network-on-a-Chip. The blue lines indicate cMesh (used for on-chip writes), green — xMesh (off-chip writes), red — rMesh (read requests).

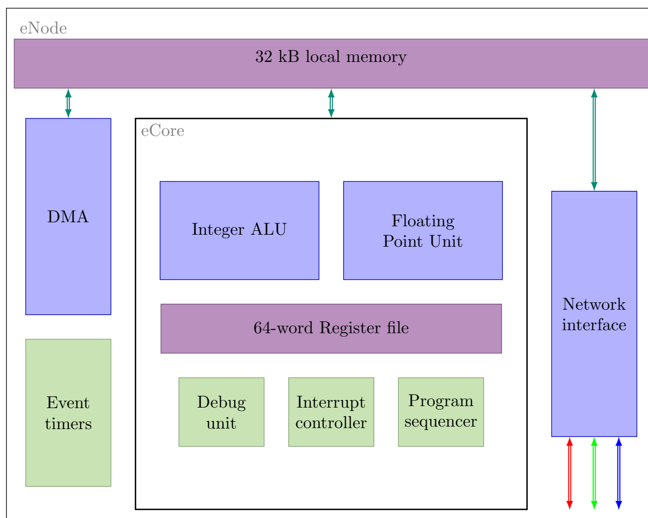

Fig. 2. eNode components. Each eNode has an eCore and 32 kB of local memory, a network router, a DMA engine and two event timers. Each eCore has a 64-word register file, FPU, ALU, interrupt controller, sequencer and debug unit.

- writes are preferred over reads – for a single read there need to be two transactions: one for a read request and one for an answer,
- non-local memory accesses are weakly ordered.

The main goals of the Epiphany architecture are: power efficiency (a single 16-core Epiphany processor consumes a maximum of 2 W, the whole Parallella board requires around 5 W), scalability, an easy programming model, and high performance (2 GFLOPS per single core).

Currently, there are 16- and 64-cores Epiphany processors available. However, 64-core versions have only been produced in limited numbers and are only available directly from Adapteva.

The Parallella board uses a 16-core Epiphany processor (E16G301), a Xillinx Zynq (models 7010 or 7020 with two ARM cores) and 1 GB of RAM. Additional components include: 1 Gbit Ethernet interface, USB and HDMI ports, and a MicroSD slot. The standard operating system (in this case – Linux) boots from the MicroSD card onto an ARM processor and can communicate with the Epiphany using an e-Link interface. 32 MB of memory is shared between ARM processors (host) and Epiphany. It is mapped in eNodes to `0x8E000000–0x8FFFFFFF` address space. eCores can use it in the same way as internal memory, however the performance will be lower as shown in Section IV-A1.

The memory size usable by the programmer is dependent on many factors. In the most common linker configuration the internal memory (32 kB) is used, for example, to store:

- program code,
- global variables,
- stack.

And a fragment of the external memory (32 MB) is used for the C standard library code, data and stack [9].

Programming on the Epiphany side is done in the usual way. The SDK supports the standard C library with mathematics functions. Additionally, it provides some specific utilities for managing hardware resources: registers operations, interrupts handling, timers, mutexes, barriers and DMA functions. The "workgroup" concept is supported and each created workgroup can have a different code loaded. The important thing to remember is that Epiphany is an MIMD processor and each core can execute a completely different code. There is no synchronization between cores (besides library functions and experimental `SYNC` instruction).

Alternative approaches to using the basic SDK are MPI [10] and OpenMP [11].

Epiphany does not provide double precision float operations in hardware. As such, these are emulated by a compiler and thus carry performance loss when used.

III. Programs

Our benchmark application was a simple genetic algorithm [12] with the fitness-proportional selection, mutation

enabled and using the two-dimensional Beale's function as a fitness function:

$$f(x,y) = (1.5 - x + xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.625 - x + xy^3\right)^2$$

The initial population was generated within $[-5.0, 5.0]$ boundary.

With this basic skeleton we prepared several versions of the implementation for the Parallella using different solutions and optimizations.

As we wanted to implement the whole algorithm on the Epiphany processor, we needed a separate pseudo-random number generator. We used a "tiny" version of the Mersenne Twister [13], [14] that requires only 127 bits of memory, and could be easily used with Epiphany.

A. *Fitness computation offloading*

The first and the simplest way to use the Epiphany processor is to offload what is usually the heaviest computation in genetic algorithms – the evaluation of the fitness function.

The main loop of the computation performs the following operations:

1) Generate a new population (on the host side).
2) Compute the fitness (on the Epiphany processor).
3) Find the best organism (on the host side).

The simplified version of the function initializing and starting Epiphany cores is shown in Listing 1. We perform the following steps:

1) reset workgroup,
2) load the device code,
3) write population and its size,
4) start the workgroup,
5) wait for all cores to finish working,
6) read the computed fitness values.

```
void epi_fitness_fill(
    simulation_t * simulation,
    e_platform_t * platform) {

  // Reset device and load code
  e_reset_group(&dev);
  e_load_group("e_main.srec", &dev, 0, 0,
    platform->rows, platform->cols,
    E_FALSE);

  // Write required data - size and population
  e_write(&e_size_mem, 0, 0, 0x0,
    &(simulation->size), sizeof(float));
  e_write(&e_population_mem, 0, 0, 0x0,
    simulation->population,
    simulation->population_size);

  // Send start interrupt
  e_start_group(&dev);

  // Wait for all cores to finish work
  epi_wait_for_status(STATUS_EXITED);

  // Read computed fitness
  e_read(&e_fitness_mem, 0, 0, 0x0,
    simulation->fitness.values,
    simulation->fitness_size);
}
```

Listing 1. Host part of fitness computation

For transferring the population we use an array of pairs of floats (with a size twice that of the population). The output (fitness values) is stored in a separate array of floats. In the basic version, both are located in the external memory.

B. *Full population evaluation*

In the second version we offloaded both the fitness computation and finding the best individual to the Epiphany cores. In this case, each core, finds the best individual in its fragment of the population. After computing all the fitness values, each core sends (using a 16-element `size_t` array) the index of the best individual that was found. The host collects these 16 elements and chooses the best among them. It also generates a new population.

C. *Whole algorithm on Epiphany*

In the final stage, we also implemented the whole algorithm on the Epiphany processor. Most of the implementation was straightforward, as the code flow could be similar. The main issue was handling large data with the small amount of local memory.

In order to handle a large number of organisms efficiently, for each iteration, we split the population into chunks each consisting of 16×1024 organisms (except the last one). Each core copies 1024 organisms from the external memory into its local memory. Then, it computes the fitness and generates a new population for this fragment and puts the new organisms to the external memory. After all cores perform these operations on all chunks, they load new organisms from the external memory and execute the next iteration.

A loss in the exchange of organisms between cores can be observed and we basically operate on subpopulations. However, to resolve (at least partially) this problem, we shuffle the population when copying organisms from the external memory. We do this efficiently using the DMA engine (see Section III-D3).

There is no explicit communication between cores and the host, although the host can "preview" the data at any time. In addition, cores do not exchange data in any other way than using the external memory, as previously mentioned.

The role of the host is only limited to initializing the population, copying data to the external memory and, after all iterations, copying the final population and fitness values back.

D. *Implementation solutions used in test programs*

In order to improve the efficiency of the implementation, we have implemented and tested technical optimizations such as removing the need for reloading the Epiphany code and using better communication facilities.

1) Removing need for reset–load before each computation: By default Epiphany does not provide a simple way to "restart" the same computation as before. The programmer needs to reset a device group and load the device code again when needed. Although such an approach generally works, it could have a noticeable impact on performance. We have measured the reload time and presented the results in Section IV.

The solution to this issue is to make the computation execute in an infinite loop and signal new data available for processing by using the same interrupt that is used for starting the computation (`E_SYNC`). At the beginning of the computation we register and enable an empty handler for this interrupt. Then, when waiting, we execute `idle` opcode in a loop effectively putting the core to sleep. When it receives the interrupt, it wakes up and retests the loop condition which can be, for example, checking for new data. The waiting loop is shown in the Listing 2.

```
while (1) {
  // Perform computation
  while (/* Test status/flag */){
    // Go into idle mode
    __asm__ __volatile__ ("idle");
  }
}
```

Listing 2. Waiting loop

On the host side we simply execute `e_start_group` on the whole workgroup each time we want to signal new data. Internally, it sends the correct interrupt.

2) Communication: In most cases there is no need for communication between cores, as data can be cleanly split among them. On the other hand, communication with the host is performed at least two times for every step of the computation (for versions executing on Epiphany only partially). Epiphany does not provide any way to synchronize its cores with the external host.

We solved this issue by using a simple status variable that eCores use to share their statuses with the host and the host signals that there is new work to do. We store statuses in a 16-element array of `uint8_t` (8-bit unsigned integer) elements. This gives us enough flexibility to use multiple different statuses and is still small enough not to waste the memory.

To wait for a specific status on the host side we make a tight loop for reading the whole array and check all values after every read.

On the Epiphany side if the core needs to wait for a particular status, it is done by a simple `while` loop checking a dedicated array index.

3) DMA: We have also decided to test the difference the DMA engine makes to memory operations. Epiphany offers two DMA channels per core. They can be used in a linear (copying continuous blocks of memory) or non-linear (copying regularly spread fragments of memory) fashion. In most cases we used the former method, but for the porting of the whole algorithm, we used the latter. This allowed us to efficiently shuffle populations without involving cores.

TABLE I
MEASURED MEMORY BANDWIDTH ON PARALLELLA.

Initiator	Target	Type	Bandwidth (MB/s)
ARM Host	eCore (0,0)	write	45.82
ARM Host	eCore (0,0)	read	5.20
ARM Host	DRAM	write	88.25
ARM Host	DRAM	read	131.96
ARM Host	DRAM	memcpy	353.01
eCore (0,0)	eCore (1,0)	write (DMA)	1242.38
eCore (0,0)	eCore (1,0)	read (DMA)	401.46
eCore (0,0)	DRAM	write (DMA)	233.94
eCore (0,0)	DRAM	read (DMA)	87.45
eCore (0,0)	eCore (1,0)	write	534.37
eCore (0,0)	eCore (1,0)	read	115.70
eCore (0,0)	DRAM	write	71.61
eCore (0,0)	DRAM	read	4.29

IV. RESULTS

Our tests focused mainly on execution time and memory performance. We performed two micro-benchmarks in order to determine:

- memory bandwidths,
- Epiphany initialization penalty.

Then, we measured times related to the execution of the test case presented in the previous section.

A. Microbenchmarks

1) Memory bandwidth: The memory bandwidth results shown in Table I were obtained using a micro-benchmark provided with the Parallella SDK. It is easily observable that writes involving the Epiphany processor perform significantly (several times) better than corresponding reads. This is related to the way the reads are executed (see Section II) – they consist of two transactions. The DMA engine also provides a large speed boost.

Moreover, these operations are significantly slower than the theoretical limits which would be 1.6 GB/s [15] for the bidirectional off-chip traffic and 8 GB/s for on-chip DMA [16]. However, for the former the cMesh implementation limits it to around 4.8 GB/s. There are also two errata items reporting issues with DMA engine limiting its bandwidth to around 25% of this value.

These speeds and significant differences to the theoretical values are similar to other results [17].

2) Loading the code on Epiphany: To load and start the code on the Epiphany processor we need to call the following functions:

1) `e_reset_group`
2) `e_load_group`
3) `e_start_group` (this can be implicitly called by previous function).

There is no means provided to "restart" the same code on Epiphany and it is necessary to handle it manually using interrupts (see Section III). Thus, we decided to perform a microbenchmark measuring the time it takes to execute a full reload operation on all 16 cores.

The results are as follows: for the code and data filling the whole 32 kB of core memory: 314.613 ms. For the

minimal example (writing a single integer to the well-known memory buffer): 117.620 ms. These values are not high, but considering possible multiple iterations of the algorithm they can accumulate to significant delays.

B. Genetic algorithm

We implemented several different optimization scenarios and measured execution times for each of them. These scenarios were:

- naive — no Epiphany-specific optimizations: we put the data in the external (DRAM) memory and restarted Epiphany in each iteration, cores use data directly from DRAM,
- no reload — as above but without code reloading and Epiphany restart (see Section III-D1),
- local — as "naive" but each core copied fragments of data to local memory before processing,
- no reload, local — as "no reload" but each core copies fragments of data to the local memory before processing (using standard `memcpy`),
- no reload, local, dma — as "no reload, local" but cores used DMA engine to perform copying of data,
- no reload, push — as "no reload" but the host (ARM) pushed data directly to the local memory of each core; it is the host's responsibility to split the data into sizes fitting the local Epiphany memory.

Additionally, we executed the basic one-thread computation on the Parallella's ARM CPU as a reference for other measurements.

For every scenario we executed 100 (one hundred) iterations over various selected population sizes: 16, 32, 128, 256, 1024, 2048, 8192, 10240, 51200. These numbers needed to be divisible by 16, as each core should have the same population to work on. The results for 51200 are presented in Table II.

Our time measurements included three, increasing in size, portions of the code:

- code execution on the Epiphany coprocessor in a single iteration and the same code on the host (in the CPU version),
- a single iteration — both ARM host and Epiphany code including code upload in some of the above scenarios but without the generation of a new population,
- a whole algorithm — from the Epiphany initialization to closing the device.

For measuring the execution time we used the `clock_gettime()` function with the `CLOCK_MONOTONIC` clock.

We tested the three scenarios described in Section III: fitness offloading, population evaluation and a whole algorithm. For the whole algorithm we measured only the full computation time, as the measurement of a single iteration would not be efficient or useful.

It is worth noting that in the largest measured population (51200 organisms) each Epiphany core has only 3200 organisms to evaluate.

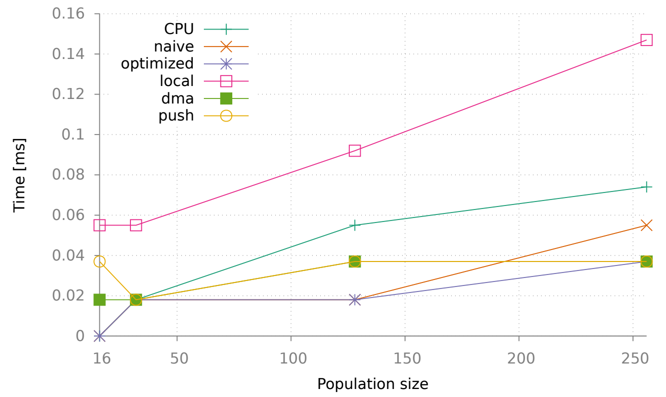

Fig. 3. Time of a single iteration execution of the "fitness offloading" version on the Epiphany processor for population sizes 16, 32, 128, 256.

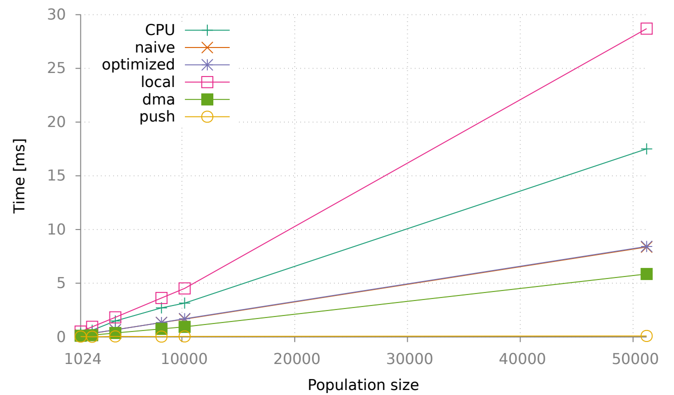

Fig. 4. Time of a single iteration execution of the "fitness offloading" version on the Epiphany processor for population sizes 1024, 2048, 8192, 10240, 51200.

Figures 3 and 4 show times for the Epiphany-offloaded code and the equivalent code on the CPU. Firstly, we can observe, that the "push" version performs the best. This is a direct result of the fact that there are no external memory operations on the Epiphany side in this version. All memory handling is done on the host. Secondly, the "local" version that uses `memcpy` performs more than two times worse than the CPU version. This is caused by very slow (around 4.3 MB/s, as shown in Table I) read rates for non-DMA copies between cores and DRAM. After changing the copy method to DMA the measured time was significantly reduced, performing even better than no-copy versions ("naive" and "no reload"). Finally, there are no significant differences between versions with and without code reloading so more complicated iteration logic on the Epiphany side has no penalties (as expected).

For smaller population sizes some of these observations differ (DMA and "push" versions are slower), but this is due to the initialization of these memory access paths [16].

Figures 5 and 6 show times for the whole iteration – host and device sides – without generation of the new population. They do not include lines for the "naive" version for readability.

TABLE II
MEASURED EXECUTION TIMES FOR 51200 ORGANISMS.

Version	Version	Epiphany [ms]	Iteration [ms]	Full [s]
CPU	—	13.769	16.533	545.53
fitness offloading	none	8.368	223.764	546.83
	no reload	8.423	22.303	546.24
	no reload, local	28.699	42.559	546.03
	no reload, local, dma	5.806	19.593	545.89
	local, dma	5.861	232.556	546.93
	no reload, push	0.276	51.702	546.11
full	none	8.645	215.802	546.68
	no reload	8.737	18.248	545.90
	no reload, local, dma	5.714	15.280	545.87
whole	local memory	—	—	2.42
	external memory	—	—	38.73

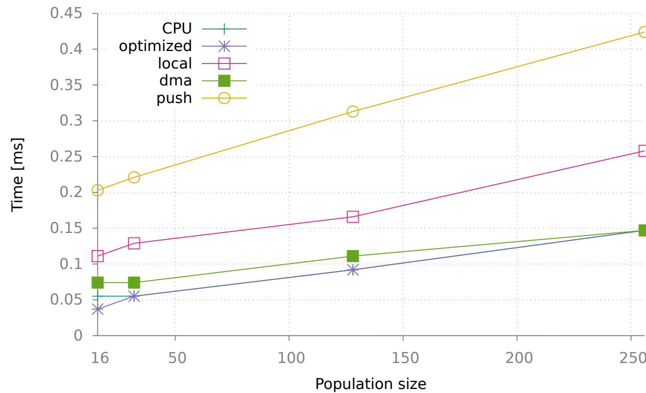

Fig. 5. Time of a single iteration execution of the "fitness offloading" version (Epiphany + host) for population sizes 16, 32, 128, 256. The "naive" version is not included due to the scale.

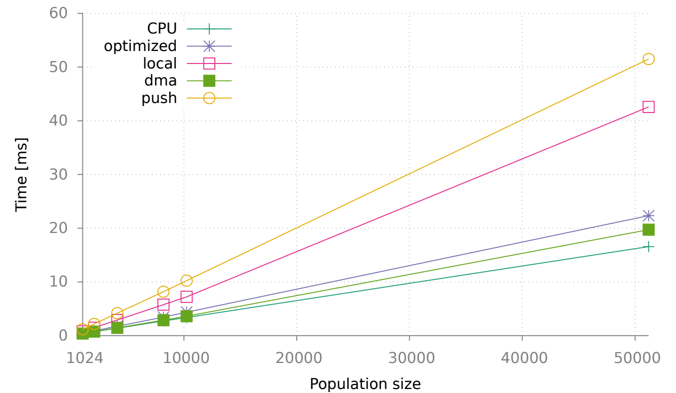

Fig. 6. Time of a single iteration execution of the "fitness offloading" version (Epiphany + host) for population sizes 1024, 2048, 8192, 10240, 51200. The "naive" version is not included due to the scale.

The first thing to observe is the very poor performance (as much as ten times worse) of the versions that reset the Epiphany during the iteration. As we noted in Section IV-A2, such an operation takes at least 110 ms (for the smallest possible binary).

In these results, we see that the "push" version performs poorly. This is due to the fact, that all of the memory operations are executed on the host side and, firstly, host-to-core transfers are slower (as seen in Table I), secondly, the host needs to perform sixteen separate copies instead of one to the external memory. The CPU version has the best performance, but it is important to note, that there are no additional memory copies in this version. In all Epiphany implementations we need to perform at least one additional copy of the population per iteration.

We also measured the execution time of the whole program (for a smaller number of iterations), however, as the most time-consuming task for large populations is the generation of the new population, the results are similar in each implementation.

The "full evaluation" version, which also included finding the best organism on Epiphany, did not make any significant difference to the previous one. We can observe, that iteration times are several milliseconds shorter than the corresponding "offload" version. This follows from the fact, that we split the lookup on all cores.

The final version we tested — the whole algorithm ported to Epiphany — was implemented in two versions: one which copied the data to the local memory of cores (using DMA), and another which operated fully on the external memory.

The latter performed nearly 15 times better than the single CPU version, which is very good considering the memory bandwidths shown in the Table I. However, the former, using the local memory, presented the best computation time: below 3 s.

C. Memory limits of Parallella

As mentioned in Section II, the Epiphany cores in Parallella board can address only 32 kB of the internal and 32 MB of the external memory. These limits have significant impact on the size of program and data size.

This 32 kB of internal memory is divided in four banks, of which only two are usable for the user data without limitations, because the first one contains the code and the last one – the stack (starting at the end of the block). It is safe to assume that, for user data, 16 kB is fully available on each core, plus the remaining memory from the fourth bank. We have limited the local memory used in test scenarios to these sizes. As we use 24 bytes for a single organism, we can store a maximum

of 1024 organisms (if we ignore the stack). However, in cases where we do not need to remember the population between iterations we can safely reuse the same memory for delivering results to the host, giving us 1536 organisms for a single core.

As previously mentioned, one has to consider the stack, so the final numbers have to be smaller and the real code size must be taken into account.

D. Observed issues

During development we observed several issues with the board and SDK.

First, the board has a tendency to overheat. When the temperature measured by sensors reached 65°C we observed unstable behavior: missing writes (the host did not see updates from Epiphany) and hangups. To counteract these issues we kept the board cooled to around 50°C.

Second, the default linker configuration places standard C library and math functions in the external memory (DRAM). For example, we tested the code with the two-dimensional Ackley test function:

$$f(x,y) = -20\exp\left(-0.2\sqrt{0.5\left(x^2+y^2\right)}\right) \\ -\exp\left(0.5\left(\cos\left(2\pi x\right)+\cos\left(2\pi y\right)\right)\right)+e+20$$

In this case, the Epiphany version was several times slower than the plain CPU implementation. This is due to the execution of the math functions from the external memory. One of the solutions for this issue is to change linker configuration to place them in the local memory of the Epiphany.

V. Conclusions and future work

Our main concern in this paper was to review the Parallella board in order to follow later with more advanced plans touching on simulations and multi-agent systems. We have reviewed the board internals and its programming model. We then performed some microbenchmarks and tested a genetic algorithm in various versions from a naive implementation to more advanced optimizations.

Our benchmark results show that the Parallella can be fast and offers quite large benefits, but these are destroyed by slow memory transfers and the large number of manual optimizations required to get to them. Comparing only the results for a single-core CPU version and Epiphany "push" versions (as described in Section IV) we can notice that actual computation is nearly 50 times faster on Parallella. However, the work required for memory copying between the device and the host causes the Epiphany version to perform worse in a single iteration than the single-core ARM one. To gain really better results, we need to port the whole algorithm to the Epiphany in order to avoid Epiphany–ARM memory copying. However, porting the code, especially with a very small local memory, requires significant work.

Nevertheless, in the case of offloading only the most costly part of algorithms (for example, fitness computation), the local copying by Epiphany cores using the DMA engine has the best performance among all implementations and would be recommended for simple scenarios.

It is clear from our and others' results that Parallella is not mature enough to replace more advanced and well-established manycore platforms. However, it is an interesting board that can be used to accelerate parallel workloads in a very simple and cheap way. There is nearly no effort required to port programs written in C to the Epiphany compiler in a naive way. Optimization of such programs requires more work, but it is still simple considering the relatively poor feature set of the Software Development Kit and the processor.

Our next steps regarding Parallella will focus on the usage of multiple boards in a single cluster and Java–Epiphany interaction. We see possibilities for deploying island and agent-based models of evolutionary algorithms [4] on such clusters, as they usually can be organized in a way that requires little communication between islands. Separate populations could be computed on different boards with rare synchronization events between them.

The last model, namely Evolutionary Multi-Agent Systems (EMAS) [18], [19], which uses the agent paradigm for decentralizing the process of evolution, is of special interest to us, since it allows achieving a fine-grained parallelism with its implementation of agents [7]. This opens the way for another possible approach, even without using multiple boards – the implementation of multi-agent systems with lightweight agents (or groups of agents) executing on separate cores [20]. The MIMD nature of the Epiphany processor should be a matching architecture for these systems.

Acknowledgment

The research reported in the paper was supported by the grant "Hybrid model of the early detection of internal diseases based on the paradigm of interacting particles and multi-agent system" (No. DEC-2013/09/N/ST6/01011) from the Polish National Science Centre.

References

[1] J. Shalf, J. Bashor, D. Patterson, K. Asanovic, K. Yelick, K. Keutzer, and T. Mattson, "The MANYCORE revolution: will HPC lead or follow," *SciDAC Review*, vol. 14, pp. 40–49, 2009.

[2] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting High-performance Energy-efficient Manycore Architectures with Epiphany," Nov. 2-5, 2014. doi: 10.1109/ACSSC.2014.7094761

[3] G. Chrysos, "Intel® Xeon Phi™ Coprocessor-the Architecture," *Intel Whitepaper*, 2014.

[4] M. Kisiel-Dorohinicki, G. Dobrowolski, and E. Nawarecki, "Agent populations as computational intelligence," in *Neural Networks and Soft Computing: Proceedings of the Sixth International Conference on Neural Networks and Soft Computing, Zakopane, Poland, June 11–15, 2002*, L. Rutkowski and J. Kacprzyk, Eds. Heidelberg: Physica-Verlag HD, 2003, pp. 608–613. ISBN 978-3-7908-1902-1

[5] M. Kisiel-Dorohinicki, "Agent-based models and platforms for parallel evolutionary algorithms," in *Computational Science - ICCS 2004*, M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2004, pp. 646–653.

[6] Ł. Faber, K. Piętak, A. Byrski, and M. Kisiel-Dorohinicki, *Agent-Based Simulation in AgE Framework*. Springer Berlin Heidelberg, 2012, pp. 55–83. ISBN 978-3-642-28888-3

[7] D. Krzywicki, W. Turek, A. Byrski, and M. Kisiel-Dorohinicki, "Massively concurrent agent-based evolutionary computing," *Journal of Computational Science*, vol. 11, pp. 153–162, nov 2015. doi: 10.1016/j.jocs.2015.07.003

[8] M. Pietroń, A. Byrski, and M. Kisiel-Dorohinicki, “GPGPU for difficult black-box problems,” *Procedia Computer Science*, vol. 51, pp. 1023–1032, 2015. doi: 10.1016/j.procs.2015.05.249
[9] *Epihany SDK Reference, rev. 5.13.09.10*. [Online]. Available: http://adapteva.com/docs/epiphany_sdk_ref.pdf
[10] J. A. Ross, D. A. Richie, S. J. Park, and D. R. Shires, “Parallel Programming Model for the Epiphany Many-Core Coprocessor Using Threaded MPI,” in *Proceedings of the 3rd International Workshop on Many-core Embedded Systems*. ACM, 2015. doi: 10.1145/2768177.2768183 pp. 41–47.
[11] A. Papadogiannakis, S. N. Agathos, and V. V. Dimakopoulos, *OpenMP 4.0 Device Support in the OMPi Compiler*. Cham: Springer International Publishing, 2015, ch. OpenMP 4.0 Device Support in the OMPi Compiler, pp. 202–216. ISBN 978-3-319-24595-9
[12] T. Back, D. B. Fogel, and Z. Michalewicz, Eds., *Handbook of Evolutionary Computation*, 1st ed. Bristol, UK, UK: IOP Publishing Ltd., 1997. ISBN 0750303921
[13] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. doi: 10.1145/272991.272995
[14] (2011) Tiny Mersenne Twister (TinyMT): A small-sized variant of Mersenne Twister. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/
[15] Andreas Olofsson - Public forum communication. [Online]. Available: https://parallella.org/forums/viewtopic.php?f=9&t=2391&p=13653
[16] *Epihany Architecture Reference, rev. 14.03.11*. [Online]. Available: http://adapteva.com/docs/epiphany_sdk_ref.pdf
[17] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, “Programming the Adapteva Epiphany 64-core network-on-chip coprocessor,” *International Journal of High Performance Computing Applications*, 2015. doi: 10.1109/IPDPSW.2014.112
[18] A. Byrski and M. Kisiel-Dorohinicki, *Man-Machine Interactions 3*. Cham: Springer International Publishing, 2014, ch. Agent-Based Approach to Continuous Optimisation, pp. 487–494. ISBN 978-3-319-02309-0
[19] A. Byrski, R. Dreżewski, L. Siwik, and M. Kisiel-Dorohinicki, “Evolutionary Multi-Agent Systems,” *The Knowledge Engineering Review*, vol. 30, no. 02, pp. 171–186, mar 2015. doi: 10.1017/s0269888914000289
[20] D. Krzywicki, Ł. Faber, A. Byrski, and M. Kisiel-Dorohinicki, “Computing agents for decision support systems,” *Future Generation Computer Systems*, vol. 37, pp. 390–400, jul 2014. doi: 10.1016/j.future.2014.02.002