UDC 004
DOI: 10.21122/2309-4923-2023-2-4-12

PRIHOZHY A.A., KARASIK O.N.

INFLUENCE OF SHORTEST PATH ALGORITHMS ON ENERGY CONSUMPTION OF MULTI-CORE PROCESSORS

Belarusian National Technical University
Minsk, Republic of Belarus

Modern multi-core processors, operating systems and applied software are being designed towards energy efficiency, which significantly reduces energy consumption. Energy efficiency of software depends on algorithms it implements, and, on the way, it exploits hardware resources. In the paper, we consider sequential and parallel implementations of four algorithms of shortest paths search in dense weighted graphs, measure and analyze their runtime, energy consumption, performance states and operating frequency of the Intel Core i7-10700 8-core processor. Our goal is to find out how each of the algorithms influences the processor energy consumption, how the processor and operating system analyze the workload and take actions to increase or reduce operating frequency and to disable cores, and which algorithms are preferable for exploiting in sequential and parallel modes. The graph extension-based algorithm (GEA) appeared to be the most energy efficient among algorithms implemented sequentially. The classical Floyd-Warshall algorithm (FW) consumed up to twice as much energy, and the blocked homogeneous (BFW) and heterogeneous (HBFW) algorithms consumed up to 52.2 % and 21.2 % more energy than GEA. Parallel implementations of BFW and HBFW are faster by up to 4.41 times and more energy efficient by up to 3.23 times than the parallel implementation of FW and consume less energy by up to 2.22 times than their sequential counterparts. The sequential GEA algorithm consumes less energy than the parallel FW, although it loses FW in runtime. The multi-core processor runs FW with an average frequency of 4235 MHz and runs BFW and HBFW with lower frequency of 4059 MHz and 4035 MHz respectively.

Keywords: *multi-core processor, shortest paths algorithm, single-thread application, multi-threaded application, runtime, energy consumption, OpenMP*

Introduction

Multi-core CPUs are at the heart of modern computing platforms whose share of the total energy consumption is rapidly increasing. The energy consumption of computing systems and devices accounts for 20% of the global electricity demand [1, 2], and the prediction is up to 50% of global electricity in 2030. A model for estimating with high accuracy the power consumption of multi-core processors is presented in [3].

Power management is one of the most critical issues in the design of today's microprocessors [4, 5]. Its goal is to maximize performance within a given power budget. Power management techniques must balance between the demanding needs for higher performance and the impact of aggressive power consumption and negative thermal effects. The most adopted power saving technique for current multi-core processors is the ability of dynamic frequency tuning which is based on Dynamic Voltage and Frequency Scaling (DVFS). Many studies use DVFS to adjust the frequency of processor cores, and to save power. They are classified into two groups: profiling and performance monitors. The profiling techniques measure the behaviors of applications and analyze the obtained results to tune the frequency of processors. The hardware performance monitors collect information about CPU usages in run-time and then tune the frequency of multi-core processor to save power without significant overhead.

Energy consumption can also be decreased by optimizing machine code and creating green software. The contribution of this paper is a methodology of developing and selecting applied algorithms (on example of shortest paths algorithms) which significantly reduce the energy consumption and increase performances.

All pairs shortest path algorithms

Let $G = (V, E)$ be a simple directed dense graph with real edge-weights consisting of a set V, $|V| = N$, of vertices numbered 1 through N and a set E of edges. Let W be a cost adjacency matrix for G. So, $w(i, i) = 0$, $1 \le I \le N$; $w(i, j)$ is the cost (weight) of edge (i, j) if $(i, j) \in E$ and $w(i, j) = \infty$ if $i \ne j$ and $(i, j) \notin E$. Let consider the problem and algorithms of shortest paths search in graph G.

Floyd-Warshall algorithm (FW). Let D be a matrix of distances and element $D(i, j)$ be a length of a shortest path from i to j. Let $SP(i, j, k)$ be a function that returns the length of the shortest path from i to j passing through vertices from set $\{1, 2 \dots k\}$. The goal of *FW* [6–8] is to find $SP(i, j, N)$, $i, j = 1 \dots N$. If we have $SP(i, j, k–1)$, then $SP(i, j, k)$ can be defined recursively:

$$SP(i, j, k) = \min(SP(i, j, k–1), SP(i, k, k–1) + SP(k, j, k–1)), \quad (1)$$

with the base case $SP(i, j, 0) = w(i, j)$. The *FW* algorithm is derived from definition (1):

```
D ← W
for k ∈ {1…N}} {
    for i, j ∈ {1…N}} {
        D(i, j) ← min (D(i, j), D(i, j) + D(i, j))
}}
```

The *FW* algorithm has the same computational complexity of $\Theta(|V|^3)$ no matter how many edges the graph contains. An advantage of the algorithm is its simplicity in the organization of computations. Its drawback is the recalculation of all elements of matrix D in every iteration of the loop along k. *FW* can be parallelised by OpenMP. An alternative to *FW* is proposed in [9].

Graph extension-based algorithm (GEA). The algorithm was proposed in [10, 11]. In *GEA*, the process of calculating the shortest paths is associated with stepwise adding of vertices to graph G. Therefore, the shortest path distances are represented by a sequence of matrices $D[1\times1]$ … $D[(k–1)\times(k–1)]$, $D[k\times k]$ … $D[N\times N]$. *GEA* uses two operations: 1) adding row k and column k to matrix $D[(k–1)\times(k–1)]$ with obtaining matrix $D[k\times k]$; 2) updating matrix $D[(k–1)\times(k–1)]$ over row k and column k. These operations are described as:

```
D ← W
for k ∈ {1...N} {
    for i, j ∈ {1…N} {
        D(i, k) ← min (D(i, k), D(i, j) + D(j, k))
        D(k, j) ← min (D(k, j), D(k, i) + D(i, j))
    }
    for i, j ∈ {1…N} {
        D(i, j) ← min (D(i, j), D(i, k) + D(k, j))
}}
```

Then the obtained algorithm is formally transformed to a more efficient one using the inference technique proposed in [10, 11]. The transformation rules of the resynchronization of computations, reordering of instructions and merging of loops are used to do it. The following algorithm, *GEA* is a result of the transformation:

```
D ← W
for k ∈ {2…N} {
    r ← k–1
    for i, j ∈ {1…r} {
        D(i, j) ← min (D(i, j), D(i, r) + D(r, j))
        D(i, k) ← min (D(i, k), D(i, j) + D(j, k))
        D(k, j) ← min (D(k, j), D(k, i) + D(i, j))
}}
for i, j ∈ {1…N–1} {
    D(i, j) ← min (D(i, j), D(i, N) + D(N, j))
}
```

GEA has smaller number of iterations of loops along variables i and j, has fewer accesses to memory and has the improved spatial and temporal data references locality. Therefore, it can reduce the cache pressure in the multi-core processor and can speed up the computations.

Blocked FW algorithm (BFW). It was proposed in [12–18] as a further development of *FW*. *BFW* divides set V of graph vertices into subsets $V_1 \dots V_M$ of size S and splits matrix D into blocks of size $S\times$S each, creating a block-matrix $B[M\times M]$, where equality $M\cdot S = N$ holds. The blocks are recalculated in a loop along block count $m = 1\dots M$. Three phases are used for the recalculation: 1) calculation of a diagonal $D0$ block $B_{m,m}$; 2) calculation of $2(M–1)$ cross blocks $B_{v,m}$ and $B_{m,v}$ of types $C1$ and $C2$; calculation of $(M–1)^2$ peripheral blocks of type $P3$. *BFW* is described by the following pseudocode:

```
B ← W
for m ∈ {1…M} {
    B_{m,m} ← BCA (B_{m,m}, B_{m,m}, B_{m,m})        // D0
    for v ∈ {1…M} and v ≠ m {
        B_{v,m} ← BCA (B_{v,m}, B_{v,m}, B_{m,m})    // C1
        B_{m,v} ← BCA (B_{m,v}, B_{m,m}, B_{m,v})    // C2
    }
    for v, u ∈ {1…M} and v ≠ m and u ≠ m {
        B_{v,u} ← BCA (B_{v,u}, B_{v,m}, B_{m,u})    // P3
}}
```

Single function *BCA* calculates all types of blocks:

```
BCA (B^1, B^2, B^3) {
    for k, i, j ∈ {1…S} {
        B^1(i, j) ← min (B^1(i, j), B^2(i, k) + B^3(k, j))
    }}
```

Advantages of *BFW* are: 1) the localization of data accesses within blocks and increasing the efficiency of hierarchical memory operation; 2) the capability of parallel computation of blocks on multi-core processors. *BFW* can be parallelised by OpenMP in fork-join style. Cooperative threaded algorithms [19–21] are based on *BFW*.

Heterogeneous blocked FW algorithm (HBFW). The algorithm was proposed in [11, 22]. It inherits *BFW* and distinguishes four types of blocks: diagonal $D0$, vertical $C1$ of cross, horizontal $C2$ of cross, and peripheral $P3$. For each block type it provides a separate block calculation algorithm of higher performance. The separate algorithms account the features of block types. They allow the reduction of the number of iterations in nested loops, the exploit of sequential references locality of data in CPU caches, and the speedup of computations. All the separate algorithms improve the spatial and temporal reference locality in data processing. After replacing in *BFW* four calls of function *BCA* with calls of four separate functions $D0$, $C1$, $C2$ and $P3$ using 1, 2, 2 and 3 unique arguments, we obtain a heterogeneous *HBFW*:

```
B ← W
for m ∈ {1…M} {
    B_{m,m} ← D0 (B_{m,m})
    for v ∈ {1…M} and v ≠ m {
        B_{v,m} ← C1 (B_{v,m}, B_{m,m})
```

$B_{m,v} \leftarrow C2\,(B_{m,v}, B_{m,m})\,\}$
for $v, u \in \{1\ldots M\}$ and $v \neq m$ and $u \neq m$ {
$B_{v,u} \leftarrow P3\,(B_{v,u}, B_{v,m}, B_{m,u})\,\}$
}

Function $D0$ implements the *GEA* algorithm applied to block B^1. Function $C1$ is inferred by applying the stepwise graph extension concept to block B^1 calculated over block B^3 [22]:

$C1\,(B^1, B^3)$ {
for $k \leftarrow 2\ldots S$ {$r \leftarrow k$–1
for $i \leftarrow 1\ldots S$ {
for $j \leftarrow 1\ldots r$ {
$B^1(i,j) \leftarrow \min\,(B^1(i,\text{j}), B^1(i,r) + B^3(r,j))$
$B^1(i,k) \leftarrow \min\,(B^1(i,\text{k}), B^1(i,j) + B^3(j,k))$
}}}
for $i, j \leftarrow 1\ldots$S–1 {
$B^1(i,j) \leftarrow \min\,(B^1(i,j), B^1(i,S) + B^3(S,j))\,\}$
}}

Function $C2$ is inferred in a similar way by applying the stepwise graph extension concept to block B^1 calculated over block B^2 [22]:

$C2\,(B^1, B^2)$ {
for $k \leftarrow 2\ldots S$ {$r \leftarrow k$–1
for $i \leftarrow 1\ldots r$ {
for $j \leftarrow 1\ldots S$ {
$B^1(i,j) \leftarrow \min\,(B^1(i,j), B^2(i,r) + B^1(r,j))$
$B^1(k,j) \leftarrow \min\,(B^1(k,j), B^2(k,i) + B^1(i,j))$
}}}
for $i, j \leftarrow 1\ldots$S–1 {
$B^1(i,j) \leftarrow \min\,(B^1(i,j), B^2(i,S) + B^1(S,j))$
}}

Function $P3$ is inferred from *BCA* by reordering loops. All four functions improve the spatial and temporal data references locality and make the hierarchical memory operation more efficient. Moreover, functions $D0$, $C1$ and $C2$ reduce the number of iterations in loops and the number of accesses to main memory. *HBFW* can be parallelised at task level by OpenMP using fork-join parallelization style.

Measuring energy consumption of multi-core processor

We used Intel VTune Profiler 2023.0 and built in Intel SoC Watch utility to measure energy consumption. Intel SoC Watch is a command line tool for monitoring metrics related to power consumption on Intel architecture platforms. It can report power states for the system/CPU/GPU devices, processor frequencies and throttling reasons, total energy consumption over a period, power consumption rate, and other metrics that provide insight into the system's energy efficiency. Intel SoC Watch collects data from both hardware and operating system with low overhead. Our experiments aimed at the measurement of energy consumption in Joules (*J*). To do it, we measured the runtime of each algorithm represented by single-thread and multi-threaded implementations and measured the average rate of energy consumption in Watts (*W*) of the CPU package for full duration of each algorithm execution. The CPU package energy consumption is related to all cores, each-core-separate L1 and L2 private caches, shared L3 cache and other hardware components included into the CPU package.

All runs of the program implementations of four shortest path algorithms *FW*, *GEA*, *BFW* and *HBFW* were carried out on a desktop computer equipped with Intel Core i7-10700 CPU processor which contains 8 cores (16 hardware threads) with the support of “Intel Turbo Boost 2.0”, “Intel Turbo Boost Max 3.0” and “Enhanced Intel SpeedStep” technologies. Every core is equipped with private L1 (512KB) and L2 (2MB) caches and shared L3 (16MB) cache. Its base frequency is 2.90 GHz and can increase up to 4.80 GHz. The algorithms were implemented in the C++ language using GNU GCC compiler v12.2.0.

Experiments were done on multiple randomly generated complete directed weighted graphs of 1200, 2400, 3600 and 4800 vertices. Every experiment was repeated several times and the results were verified against the results of original Floyd-Warshall algorithm. Two of the four examined algorithms are blocked and the other two are not. The following block sizes were considered: 30×30, 48×48, 50×50, 75×75, 100×100, 120×120, 150×150, 200×200, 240×240, 300×300, 600×600, 1200×1200 and 2400×2400. All the sizes divide the matrix into equal blocks without remainders.

Influence of single-thread implementations of algorithms on processor energy consumption

The sequential versions of algorithms *FW*, *GEA*, *BFW* and *HBFW* are implemented as single-thread applications written in C++. The single thread executes on one core and one logical processor at any time. Other cores are in idle state; therefore, the energy consumption is related to a part of the processor components: the core, its L1 and L2 caches, and shared cache L3. The experiments show mainly how efficiently the algorithms exploit the processor’s hierarchical memory.

The first series of experiments demonstrates how the block size in *BFW* and *HBFW* influences the processor energy consumption. Figures 1–3 show that on graph of 4800 vertices HBFW consumes less energy against BFW for all block sizes. The first reason is the runtime of *HBFW* is less than the runtime of BFW (Figure 2). The second reason is the consumption rate of *HBFW* is less against *BFW* for most block sizes (Figure 3). The figures also show that *GEA* has the lowest energy consumption and runtime of all the algorithms at any size of block; *FW* appears to be the worst with respect to both runtime and

energy consumption. At the same time *FW* and *GEA* have the same energy consumption rate (Figure 3).

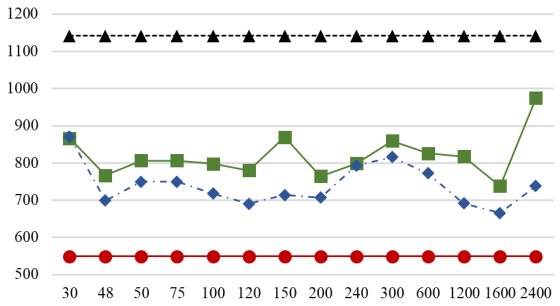

Figure 1. Energy consumption (*J*) of *FW* (triangle), *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms vs. block size on graph of 4800 vertices

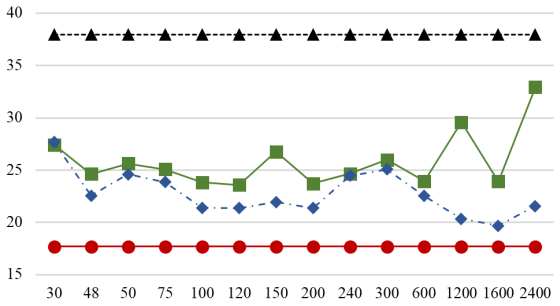

Figure 2. Runtime (*s*) of FW (triangle), *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms vs. block size on graph of 4800 vertices

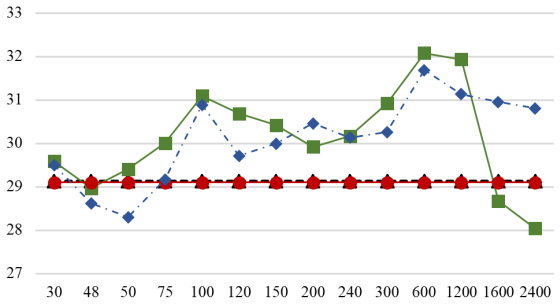

Figure 3. Average rate (*W*) of *FW* (triangle), *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms vs. block size on graph of 4800 vertices

Figure 4 compares the energy consumption that is caused by algorithms *GEA*, *BFW* and *HBFW* in comparison with those caused by *FW* on graphs of 1200 to 4800 vertices. On graph 1200, *GEA* has the lowest energy consumption. *FW* gains against *BFW* and *HBFW* but loses *GEA*. On larger graphs, *FW* loses all other algorithms.

Figure 5 depicts the speedups the *GEA*, *BFW* and *HBFW* have in comparison with *FW*. *FW* is the slowest algorithm; therefore, all the speedups exceed 1. *GEA* has the lowest runtime; as a result, it has the lowest energy consumption and yields the highest speedup. *HBFW* has a lower runtime and therefore a lower energy consumption than *BFW* has. It is interesting that there is a graph size (local optimum at 3600 vertices), for which the speedup of all three algorithms is the highest.

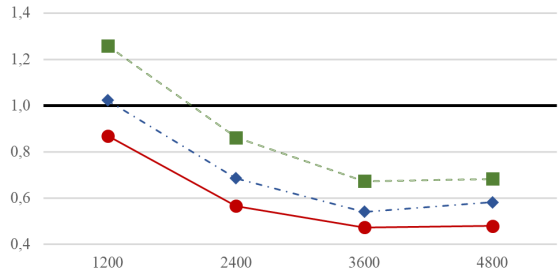

Figure 4. Relative energy consumption given by *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms against *FW* vs. graph size

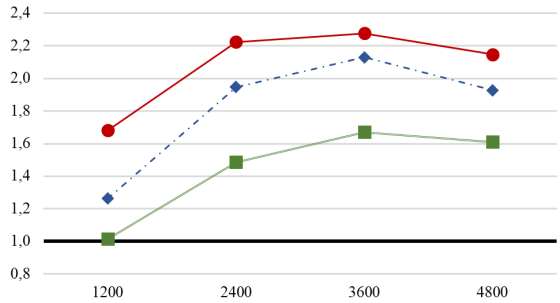

Figure 5. Speedup of *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms against *FW* vs. graph size

Figure 6 shows that algorithms *FW*, *GEA*, *BFW* and *HBFW* can gain and lose each other regarding the energy consumption rate.

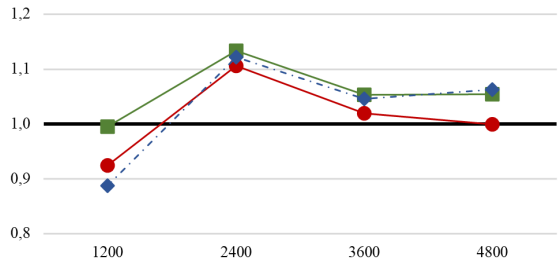

Figure 6. Energy consumption average rate of *GEA* (circle), *BFW* (square) and *HBFW* (diamond) algorithms against *FW* vs. graph size

Influence of parallel implementations of algorithms on processor energy consumption

The parallel multi-threaded implementations [23, 24] of algorithms *FW-OMP*, *BFW-OMP* and *HBFW-OMP* were generated by the OpenMP compiler. We have not succeeded to generate an acceptable parallel implementation for *GEA* using OpenMP. In the implementations, the energy consumption is related to all cores, caches, and other components of the CPU package.

Figures 7, 8 and 9 show on graph of 4800 vertices how the block size in multi-threaded *BFW-OMP* and *HBFW-OMP* influences the processor energy consumption. These algorithms consume less energy than the single-thread *GEA* and multi-threaded *FW-OMP* for block-sizes 48–300 (Figure 7). For larger block-sizes, *GEA* and even *FW-OMP* can gain *BFW-OMP* and *HBFW-OMP*. It is interesting that *GEA's* energy consumption is about twice lower than one of *FW-OMP's*.

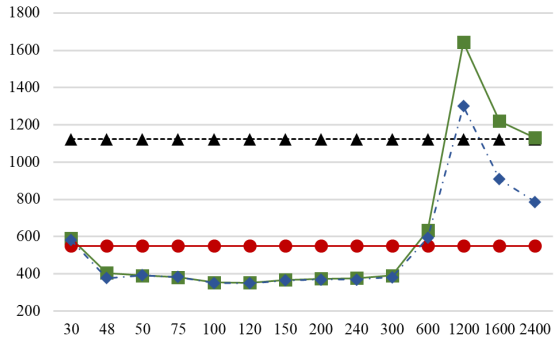

Figure 7. Energy consumption (*J*) of *FW-OMP* (triangle), *GEA* (circle), *BFW-OMP* (square) and *HBFW-OMP* (diamond) algorithms vs. block size on graph of 4800 vertices

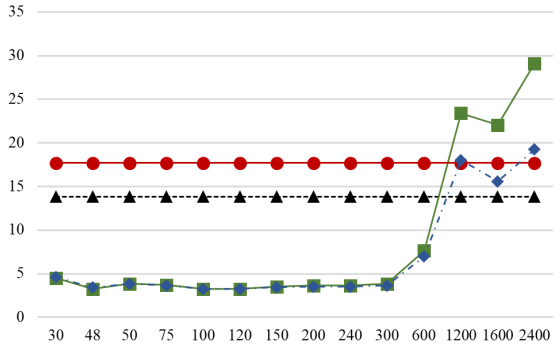

Figure 8. Runtime (*s*) of *FW-OMP* (triangle), *GEA* (circle), *BFW-OMP* (square) and *HBFW-OMP* (diamond) algorithms vs. block size on graph of 4800 vertices

The patterns depicted in Figure 8 for the algorithms' runtimes explain the patterns of energy consumption from Figure 7. The runtimes of *BFW-OMP* and *HBFW-OMP* are the lowest for most block-sizes. The runtimes of *FW-OMP* and *GEA* are close enough. Figure 9 shows that single-thread *GEA's* energy consumption rate is significantly lower than those of parallelized multiple-threaded *HBFW-OMP* and *BFW-OMP*, which in their turn have lower energy rate than those of *FW-OMP*.

It can be observed from Figure 10 that *FW-OMP* loses other algorithms on energy efficiency for any graph-size. The 1-thread *GEA* gains other multi-threaded algorithms on graph 1200. On larger graphs, *BFW-OMP* and *HBFW-OMP* consume less energy than *GEA* and *FW-OMP*.

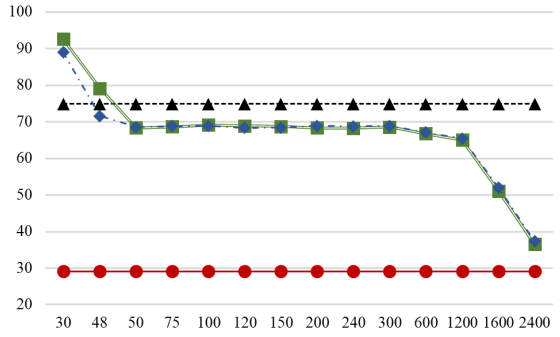

Figure 9. Average rate (*W*) of *FW-OMP* (triangle), *GEA* (circle), *BFW-OMP* (square) and *HBFW-OMP* (diamond) algorithms vs. block size on graph of 4800 vertices

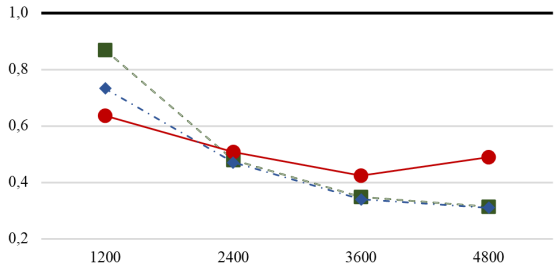

Figure 10. Relative energy consumption given by *GEA* (circle), *BFW-OMP* (square) and *HBFW-OMP* (diamond) algorithms against *FW-OMP* vs. graph size

Figures 11–12 compare multi-threaded omp-implementations against 1-thread implementations of the *FW*, *BFW* and *HBFW* algorithms depending on the graph size. The energy consumption of *FW* is higher for multi-threaded than for 1-thread implementations on almost all graph-sizes (Figures 11). Contrary, the multi-threaded *BFW* and *HBFW* have smaller energy consumption than their single-thread conterparts.

The speedup by *FW* is higher up to 3.26 times for multi-threaded implementations than for single-thread one (Figures 12). The speedup by parallel *BFW* and *HBFW* reaches 7.89 and 6.30 times. With the increase of graph size up to 3600 the speedup is being increased and then decreased.

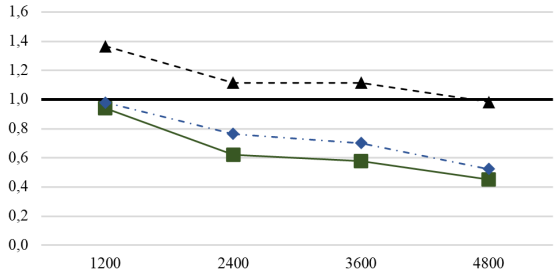

Figure 11. Relative energy consumption of OpenMP-implementations against single -thread ones of *FW* (triangle), *BFW* (square) and *HBFW* (diamond) algorithms vs. graph size

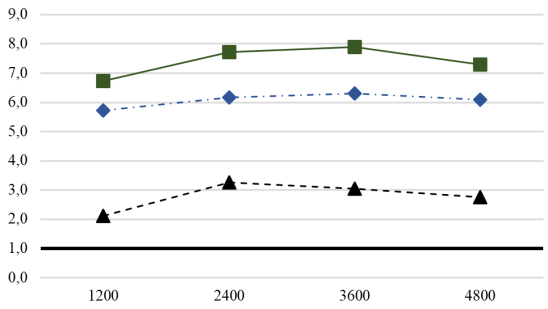

Figure 12. Speedup of OpenMP-implementations against single-thread ones of *FW* (triangle), *BFW* (square) and *HBFW* (diamond) algorithms vs. graph size

Figure 13 shows that the multi-threaded *FW-OMP* algorithm gains up to 46 % the single-thread *GEA* algorithm with respect to runtime, but the latter algorithm gains up to 57 % against the former one with respect to power consumption.

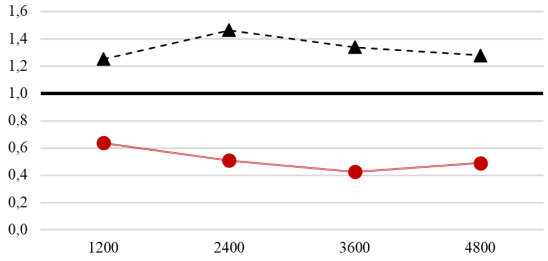

Figure 13. Relative energy consumption (solid line) and runtime (dashed line) given by *GEA* against *FW-OMP* vs. graph size

Influence of CPU performance state and operating frequency on energy consumption

Intel Core i7-10700 CPU supports 22 performance states (also known as *PX* states), where *P*0 corresponds to top performance in which processor uses its maximum capabilities and therefore may consume maximum power. The *P*1–*P*21 states correspond to active states in which processor's performance capabilities are truncated to reduce power consumption. The current *PX*-state and transitions between the states are determined by hardware. The operating system can request a change of state based on workload requirements and awareness of processor capabilities. However, in addition to the operating system request, the final decision is made accounting for different system constraints such as workload demand and thermal limits. During all conducted experiments all CPU cores were residing in the top performance *P*0 state. However, depending on the type of workload (different parallel algorithms) the active CPU frequency and percentage of residency in that frequency changed significantly.

Figures 14, 15 and 16 depict a percentage of residency in different CPU frequency intervals alongside an average frequency of each logical processor during execution of *FW-OMP*, *BFW-OMP* and *HBFW-OMP* algorithms respectively on graph of 4800 vertices. Figure 14 shows that algorithm *FW-OMP* operates over 60 % of its active time in 4600–4501 MHz frequency interval, which is a maximum non-Turbo Boost frequency of the target CPU, and the rest of its active time (around 20 %) in 4100–3901 MHz interval. This gives an average operating frequency of 4400 MHz.

At the same time, both *BFW-OMP* and *HBFW-OMP* (Figures 15 and 16) spend most of the active time in frequency intervals of 4400-4200 and 3700-3600 MHz (around 30 % and 25 % of overall time respectively), which leads to an average operating frequency of 4000 MHz. Such significant differences in operating frequencies along with the levels of references locality in big data processing result in an up to 3 times smaller energy consumption of both *BFW-OMP* and *HBFW-OMP* algorithms over the *FW-OMP* algorithm (see Figure 10).

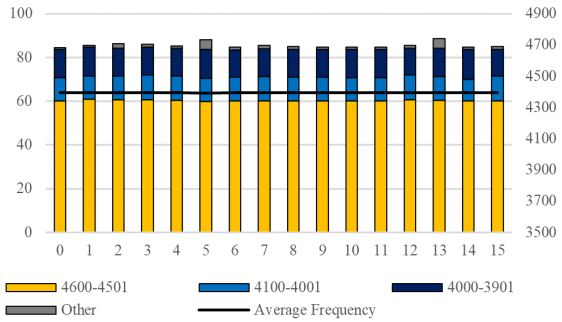

Figure 14. Logical processors residency % in CPU frequency intervals (MHz) of *FW-OMP* algorithm (stacked bars) and average CPU frequency (solid line, MHz) on graph of 4800 vertices

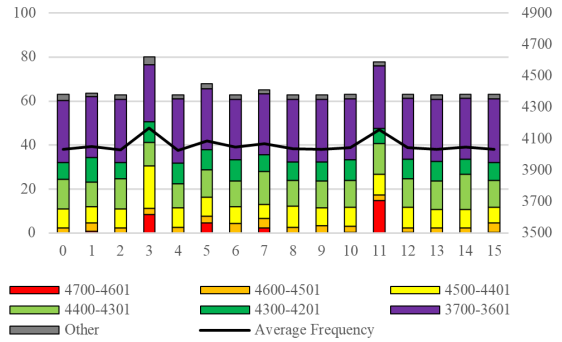

Figure 15. Logical processors residency % in CPU frequency intervals (MHz) of BFW-OMP algorithm (stacked bars) and average CPU frequency (solid line, MHz) on graph of 4800 vertices

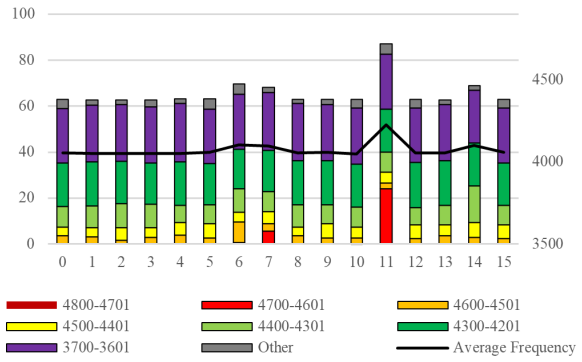

Figure 16. Logical processors residency % in CPU frequency intervals (MHz) of *HBFW-OMP* algorithm (stacked bars) and average CPU frequency (solid line, MHz) on graph of 4800 vertices

Conclusion

Modern multi-core processors are designed to exploit every possibility to reduce energy consumption. Development of algorithms and computer programs which force the processor's components to consume less energy is an additional external source of increasing the energy efficiency of hardware. On four algorithms of searching for shortest paths in large dense directed weighted graphs and on sequential and parallel implementations of the algorithms we have measured and analyzed how the processor energy consumption depends on the algorithm properties and how the processor accounts for the properties to tune its behavior with the objective of increasing its energy efficiency. The paper has found out the most energy efficient algorithms for searching for shortest paths in dense graphs.

REFERENCES

1. **Andrae A., Edler T.** On global electricity usage of communication technology: trends to 2030. Challenges 2015; 6(1):117-157. DOI: 10.3390/challe6010117

2. **Khokhriakov S., Manumachu R.R., Lastovetsky A.** Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance. Applied Energy, vol. 268, 2020, 114957, ISSN 0306-2619, DOI: 10.1016/j.apenergy.2020.114957

3. **Basmadjian R. and De Meer H.** Evaluating and modeling power consumption of multi-core processors. In Proc. of the 3rd Int'l Conf. on Future Energy Systems (e-Energy 2012). ACM, May 2012, pp. 1-10.

4. **Attia K.M., El-Hosseini M.A., Ali H.A.** Dynamic power management techniques in multi-core architectures: A survey study. Ain Shams Engineering Journal, 2017, vol. 8, no. 3, pp. 445-456.

5. **Chen K.Y., Chen F.G.** The Smart Energy Management of Multithreaded Java Applications on Multi-Core Processors. International Journal of Networked and Distributed Computing, 2013, vol. 1, no. 1, pp. 53-60.

6. **Floyd R.W.** Algorithm 97: Shortest path. Communications of the ACM, 1962, 5(6), p. 345.

7. **Madkour A., Aref W.G., Rehman F.U., Rahman M.A., Basalamah S.** A Survey of Shortest-Path Algorithms. ArXiv:1705.02044v1 [cs.DS] 4 May 2017, 26 p.

8. **Singh A., Mishra P.K.** Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm. International Journal of Computer Applications, vol. 107, no. 16, 2014, pp. 23-27.

9. **Pettie S.** A new approach to all-pairs shortest paths on real-weighted graphs. Theoretical Computer Science, 312 (1), 2004: 47-74.

10. **Prihozhy A., Karasik O.** Inference of shortest path algorithms with spatial and temporal locality for Big Data processing // Proceedings VIII International conference "Big data and advanced analytics", Minsk: Bestprint, 2022. pp. 56-66.

11. **Prihozhy A.A., Karasik O.N.** Heterogeneous blocked all-pairs shortest paths algorithm. System analysis and Applied Information Science, 2017, no. 3, pp. 68-75. DOI: 10.21122/2309-4923-2017- 3-68-75

12. **Venkataraman, G., Sahni, S., Mukhopadhyaya, S.** A Blocked All-Pairs Shortest Paths Algorithm. Journal of Experimental Algorithmics (JEA), 2003, vol. 8, pp. 857-874.

13. **Park, J.S., Penner, M., and Prasanna, V.K.** Optimizing graph algorithms for improved cache performance. IEEE Trans. on Parallel and Distributed Systems, 2004, 15(9), pp. 769-782.

14. **Albalawi, E., Thulasiraman, P., Thulasiram, R.** Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0. 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1-2, 2013, pp. 109-112.

15. **Tang, P.** Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers. IEEE SOUTHEASTCON, 2014, pp. 1-7.

16. **Karasik O.N., Prihozhy A.A.** Tuning block-parallel all-pairs shortest path algorithm for efficient multi-core implementation. System analysis and applied information science, 2022, no. 3, pp. 57–65. DOI: 10.21122/2309-4923-2022-3-57-65

17. **Prihozhy A.A.** Simulation of direct mapped, k-way and fully associative cache on all pairs shortest paths algorithms. System analysis and applied information science, 2019, no. 4, pp. 10-18.

18. **Prihozhy A.A.** Optimization of data allocation in hierarchical memory for blocked shortest paths algorithms. System analysis and applied information science, 2021, no. 3, pp. 40-50.

19. **Prihozhy A.A., Karasik O.N.** Cooperative model for optimization of execution of threads on multi-core system. System analysis and applied information science, 2014, no. 4, pp. 13-20.

20. **Prihozhy A.A., Karasik O.N.** Cooperative block-parallel algorithms for task execution on multi-core system. System analysis and applied information science, 2015, no. 2, pp. 10-18.

21. **Karasik O.N., Prihozhy A.A.** Threaded block-parallel algorithm for finding the shortest paths on graph. Doklady BGUIR, 2018, no. 2, pp. 77-84.

22. **Prihozhy A.A., Karasik O.N.** Advanced heterogeneous block-parallel all-pairs shortest path algorithm. Proceedings of BSTU, issue 3, Physics and Mathematics. Informatics, 2023, no. 1 (266), pp. 77–83. DOI: 10.52065/2520-6141-2023-266-1-13

23. **Prihozhy A.A., Karasik O.N.** Investigation of methods for implementing multithreaded applications on multicore systems. Informatization of education, 2014, no. 1, pp. 43-62.

24. **Prihozhy, A.A.** Analysis, transformation and optimization for high performance parallel computing. Minsk: BNTU, 2019, 229 p.

ЛИТЕРАТУРА

1. **Andrae A.** On global electricity usage of communication technology: trends to 2030 / A. Andrae, T. Edler // Challenges 2015. – No. 6(1):117-157. DOI: 10.3390/challe6010117

2. **Khokhriakov S.** Multicore processor computing is not energy proportional: An opportunity for bi-objective optimization for energy and performance / S. Khokhriakov, R.R. Manumachu, A. Lastovetsky // Applied Energy. – Vol. 268. – 2020. – Pp. 114957. ISSN 0306-2619. DOI: 10.1016/j.apenergy.2020.114957

3. **Basmadjian R. and De Meer H.** Evaluating and modeling power consumption of multi-core processors // In Proc. of the 3rd Int'l Conf. on Future Energy Systems (e-Energy 2012). ACM, May 2012. – Pp. 1-10.

4. **Attia K.M.** Dynamic power management techniques in multi-core architectures: A survey study / K.M. Attia, M.A. El-Hosseini, H.A. Ali // Ain Shams Engineering Journal. – 2017. – Vol. 8. – No. 3. – Pp. 445-456.

5. **Chen K.Y.** The Smart Energy Management of Multithreaded Java Applications on Multi-Core Processors / K.Y. Chen, F.G. Chen // International Journal of Networked and Distributed Computing. – 2013. – Vol. 1. – No. 1. – Pp. 53-60.

6. **Floyd, R.W.** Algorithm 97: Shortest path. Communications of the ACM, 1962, 5(6), p. 345.

7. **Singh, A.** Performance Analysis of Floyd Warshall Algorithm vs Rectangular Algorithm / A. Singh, P.K. Mishra // International Journal of Computer Applications. – 2014. –Vol. 107, No. 16. – Pp. 23-27.

8. **Madkour, A, Aref, W.G., Rehman, F.U., Rahman, M.A., Basalamah, S.** A Survey of Shortest-Path Algorithms // ArXiv:1705.02044v1 [cs.DS] 4 May 2017, 26 p.

9. **Pettie, S.** A new approach to all-pairs shortest paths on real-weighted graphs / S. Pettie // Theoretical Computer Science. 312 (1). – 2004. – Pp. 47-74.

10. **Prihozhy A., Karasik O.** Inference of shortest path algorithms with spatial and temporal locality for Big Data processing // Сборник материалов VIII Международной научно-практической конференции. – Минск: Беспринт, 2022. – Pp. 56-66.

11. **Прихожий, А.А.** Разнородный блочный алгоритм поиска кратчайших путей между всеми парами вершин графа / А.А. Прихожий, О.Н. Карасик // Системный анализ и прикладная информатика. – 2017. – № 3.– С. 68-75.

12. **Venkataraman, G., Sahni, S., Mukhopadhyaya, S.** A Blocked All-Pairs Shortest Paths Algorithm // Journal of Experimental Algorithmics (JEA). 2003. – Vol. 8. – Pp. 857-874.

13. **Park, J.S.** Optimizing graph algorithms for improved cache performance / J.S. Park, M. Penner, V.K. Prasanna // IEEE Trans. on Parallel and Distributed Systems. – 2004. – No. 15(9). – Pp. 769-782.

14. **Albalawi, E., Thulasiraman, P., Thulasiram, R.** Task Level Parallelization of All Pair Shortest Path Algorithm in OpenMP 3.0 // 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), 2013, Los Angeles, CA, July 1-2. – 2013. – Pp. 109-112.

15. **Tang, P.** Rapid Development of Parallel Blocked All-Pairs Shortest Paths Code for Multi-Core Computers // IEEE SOUTHEASTCON. – 2014. – Pp. 1-7.

16. **Карасик О.Н.** Настройка блочно-параллельного алгоритма поиска кратких путей на эффективную многоядерную реализацию / О.Н. Карасик, А.А. Прихожий // Системный анализ и прикладная информатика. – 2022. – № 3. – С. 57-65. DOI: 10.21122/2309-4923-2022-3-57-65

17. **Прихожий, А.А.** Моделирование кэш прямого отображения и ассоциативных кэш на алгоритмах поиска кратчайших путей на графе / А.А. Прихожий // Системный анализ и прикладная информатика. – 2019. – № 4. – С. 10-18.

18. **Прихожий, А.А.** Оптимизация размещения данных в иерархической памяти для блочных алгоритмов поиска кратчайших путей / А.А. Прихожий // Системный анализ и прикладная информатика. – 2021. – № 3. – С. 40-50. DOI: 10.21122/2309-4923-2021-3-40-50

19. **Прихожий, А.А.** Кооперативная модель оптимизации выполнения потоков на многоядерной системе / А.А. Прихожий, О.Н. Карасик // Системный анализ и прикладная информатика. – 2014. – № 4. – С. 13-20.

20. **Прыхожы, А.А.** Кааператыўныя блочна-паралельныя алгарытмы рашэння задач на шмат'ядравых сістэмах / А.А. Прыхожы, А.М. Карасік // Системный анализ и прикладная информатика. – 2015. – № 2. – С. 10-18.

21. **Карасик, О.Н.** Потоковый блочно-параллельный алгоритм поиска кратчайших путей на графе / О.Н. Карасик, А.А. Прихожий // Доклады БГУИР. – 2018. – № 2. – С. 77-84.

22. **Прихожий, А.А.** Усовершенствованный разнородный блочно-параллельный алгоритм поиска кратчайших путей на графе / А.А. Прихожий, О.Н. Карасик // Труды БГТУ. Сер. 3, Физико-математические науки и информатика. – 2023. – № 1 (266). – С. 77-83. **DOI:** 10.52065/2520-6141-20

23. **Прихожий, А.А.** Исследование методов реализации многопоточных приложений на многоядерных системах / А.А. Прихожий, О.Н. Карасик // Информатизация образования. – 2014. – № 1. – Pp. 43-62.

24. **Прихожий, А.А.** Анализ, преобразование и оптимизация для высокопроизводительных параллельных вычислений. – Минск: БНТУ, 2019. – 229 p.

ПРИХОЖИЙ А.А., КАРАСИК О.Н.

ВЛИЯНИЕ АЛГОРИТМОВ ПОИСКА КРАТЧАЙШИХ ПУТЕЙ НА ЭНЕРГОПОТРЕБЛЕНИЕ МНОГОЯДЕРНЫХ ПРОЦЕССОРОВ

Белорусский национальный технический университет
г. Минск, Республика Беларусь

Современные многоядерные процессоры, операционные системы и прикладное программное обеспечение разрабатываются с учетом требований энергоэффективности, что значительно снижает энергопотребление. Энергоэффективность программного обеспечения зависит от алгоритмов, которые оно реализует, и от того, как оно использует аппаратные ресурсы. В данной работе мы рассматриваем последовательную и параллельную реализации четырех алгоритмов поиска кратчайших путей на плотных взвешенных графах, измеряем и анализируем их время выполнения, энергопотребление, состояния производительности и рабочую частоту процессора. Наша цель – выяснить, как каждый из алгоритмов влияет на энергопотребление процессора, как процессор и операционная система анализируют рабочую нагрузку и предпринимают действия по увеличению или уменьшению рабочей частоты и отключению ядер, а также какие алгоритмы предпочтительнее использовать в последовательном и параллельном режимах. Алгоритм на основе расширения графа (GEA) оказался наиболее энергоэффективным среди алгоритмов, реализуемых последовательно. Классический алгоритм Флойда-Уоршалла (FW) потребил в два раза больше энергии, а блочные однородный (BFW) и неоднородный (HBFW) алгоритмы потребили на 52,2 % и 21,2 % больше энергии, чем GEA. Все эксперименты проводились на 8-ядерном процессоре Intel Core i7-10700. Параллельные реализации алгоритмов BFW и HBFW быстрее и энергоэффективнее параллельной реализации FW. Они потребили меньше энергии, чем их последовательные аналоги. Последовательный алгоритм GEA потребил меньше энергии, чем параллельный FW, хотя проиграл последнему по времени выполнения. Многоядерный процессор выполнял FW со средней частотой 4235 МГц, и выполнял BFW и HBFW с меньшей частотой 4059 МГц и 4035 МГц соответственно.

Ключевые слова: *многоядерный процессор, алгоритм кратчайших путей, однопоточное приложение, многопоточное приложение, время выполнения, энергопотребление, OpenMP*

Anatoly Prihozhy is full professor at Computer and system software department of Belarus national technical university, Doctor of Science (1999) and Full Professor (2001). His research interests include programming and hardware description languages, parallelizing compilers, and computer aided design techniques and tools for software and hardware at logic, high and system levels, and for incompletely specified logical systems. He has over 300 publications in Eastern and Western Europe, USA and Canada. Such worldwide publishers as IEEE, Springer, Kluwer Academic Publishers, World Scientific and others have published his works.

Karasik Oleg is a Technology Lead at ISsoft Solutions (part of Coherent Solutions) in Minsk, Belarus, and PhD in Technical Science. His research interests include parallel multithreaded applications and the parallelization for multicore and multiprocessor systems.