OPTIMIZATION OF PERFORMANCE AND ENERGY CONSUMPTION BY BALANCED DATA LOCALITY FOR THE EXECUTION OF PARALLEL APPLICATIONS ON NUMA SYSTEMS

PhD thesis - Abstract

for obtaining the scientific title of doctor at
Politehnica University of Timisoara
in the field of Computer Science and Information Technology

author ing. Știrb Iulia
Scientific leader Prof. PhD. Horia CIOCÂRLIE

Contents:

1. Introduction
2. The current state of research in the fields addressed in the thesis
3. Description of NUMA-BTLP and NUMA-BTDM algorithms
4. Experimental results
5. Conclusions and personal contributions

Summary content:

1. Introduction

The motivation of this thesis was to create an algorithm called *NUMA-BTLP* [1], which assigns at compile-time one type to each thread in the input code, the classification of the threads being based on static criteria that we defined in the thesis, and another algorithm called NUMA-BTDM [2] that maps threads (mapping establishes the cores that threads will run on) at compile-time according to their type, aiming to improve the balanced data locality on NUMA systems. *NUMA-BTDM* [2] takes into account the static behavior of the code when performing the mapping and eliminates the disadvantages of dynamic mapping (execution time and extra energy consumption during running) and some important disadvantages of static mapping: unpredictability at compile-time of the number of threads and unpredictability of the latencies of memory operations.

In identifying the type of the thread I have considered the following:

1. The threads that do not have data dependencies with no thread created on the same level in the hierarchy of generation of threads, are considered to be of the autonomous type. The threads that are created by the main thread usually execute independent processing, in which case they are considered to be of the autonomous type;
2. The threads between which there are data dependencies, such as, for example, those created in a loop, are considered to be of the side-by-side type;
3. Threads that do not require immediate execution are considered to be of the postponed type, these threads being usually found among the last ones created in a function. They have the property that they have data dependencies only with the generator thread.

The purpose of the research is to optimize both the execution time and the energy consumption for parallel multi-threading C / C ++ applications that use task-level parallelism

obtained using the PThreads library [3] and can also use loop-level parallelism, through optimization of balanced data locality when running on Non-Uniform Memory Access (NUMA) systems (the relationship between threads and tasks is one-to-one in the case of the POSIX parallel calculation model used by ***PThreads*** [3]). The optimization of parallel applications is achieved by applying at compile-time the two algorithms presented in the paper that have been implemented in the ***Low-Level Virtual Machine*** (***LLVM***) compiler [4], on the input code in intermediate representation.

Although a mapping at compile-time generally shows some inaccuracies due to changes in dynamic behavior at execution, these inaccuracies are largely eliminated in the case of the two algorithms by the novelty element brought by them, namely insertion at compile-time, after each ***pthread_create*** call, that creates a thread, a ***pthread_setaffinity_np*** call, setting the cores that the thread will run on. Thus, regardless of whether or not the thread will be created during the execution (aspect related to the dynamic behavior of the application), its efficient mapping at compilte-time is guaranteed, because this mapping only takes into account the processing carried out by the threads in the function attached to each one and determined by static analysis. An efficient mapping of execution threads optimizes the balanced data locality when running parallel applications on NUMA systems, which leads to increased execution performance and lower energy consumption.

The analysis that led to the decision to implement the *NUMA-BTDM* algorithm [2] in ***LLVM*** [4] had as a starting point the practical utility of this algorithm. If the algorithm were implemented in the PThreads parallel computing library then users of this library could have called the *NUMA-BTDM* [2] mapping algorithm in the code they are developing. But the call of this algorithm requires the automatic assignment of one type to each thread based on the static characteristics of the code, namely: (1) the category of autonomous threads, (2) the category of side-by-side threads from the point of view of NUMA distance (i.e the data are obtained from two sources - memories - near or the same) to the generating thread or to other threads and (3) the category of postponed threads from the point of view of NUMA time (i.e. data access time may be higher) compared to the generating thread. Each category has several criteria which are based on the static characteristics of the code. Non-automated analysis (i.e. done by the user) of a criterion requires advanced knowledge of internal memory management in NUMA systems but also hardware and operating systems in general, which is why the *NUMA-BTDM* algorithm [2] was not implemented in PThreads [3], but in ***LLVM*** [4], where the *NUMA-BTLP* algorithm classification of threads could also be implemented [1].

2. The current state of research in the fields addressed in the thesis

This chapter begins with a description of the characteristics of NUMA systems that determine the development of thread mapping algorithms. These are:

- The speed of data fetch due to the structuring of the memory on several levels of private cache and common cache as well as the existence of several memory controllers [5], which helps to minimize the latency of the data fetch operations when the data is optimally placed in memory [6] (i.e optimum means that the data is managed by the appropriate controller)
- Memory access times are uneven [5], local access (i.e. data access from the same NUMA node), being faster than remote access (i.e. data access from memory associated with another NUMA node) [7]

With the above advantages, NUMA systems offer the possibility to optimize the accesses to the memory with the help of the mapping algorithms that aim to improve the data localily.

Due to the fact that the mapping algorithm proposed in this paper is static, I have have

analyzed and presented the aspects that make the compile-time prediction of the mapping algorithms difficult on the dynamic behavior of parallel applications. These aspects are [8]:

- Not all the characteristics of the execution are known (dynamic behavior)
- The access time of the volatile and non-volatile data is stochastic and non-deterministic
- The existence of other programs running in parallel on other cores or on the same cores as the parallel application, using the common memory resources and emphasizing the non-deterministic character of the parallel application execution

Also, the mapping patterns based on the allocation policy were presented in which each thread remains allocated on the initial core until it completes its execution and the migration policy in which the threads are migrated from one core to another in the execution time according to the communication between them at the time before the migration, both types of mapping being applied to dynamic mapping.

All of these mapping or migration operations are resource-consuming, which is why the thread mapping should not be performed at execution because mapping at this stage would induce considerably more energy consumption due to the reassignment of the threads [9].

In order to determine the optimal mapping / migration of the execution threads (either static or dynamic) two information are required in advance:

- The way in which the threads access the common data, information that is most often represented by a communication matrix in which the element (i, j) represents the amount of communication (data) between the threads whose identifiers are i and j
- Information related to the hardware architecture running the parallel program, such as number of processors, cores and cache levels (this information can be provided by the ***hwloc*** utility [10])

In this chapter there have been presented software optimizations that streamline the execution of parallel applications on NUMA systems, as well as optimizations of the ***LLVM*** compiler [4] through which this is achieved. Some of the optimizations that improve automatic parallelization by reducing parallelization overhead are the following [11]: ***scalar expansion*** (i.e. the most efficient technique in this regard, involves converting scalar data into single or two-dimensional arrays), ***substitution reduction*** (the second most effective in this regard, involving substitution and then reduction), ***recurrent substitution***, ***loop invariant code motion*** (i.e, code transformation so that the body of the loop does not depend on its induction variable), ***successive substitution*** (i.e. substitution of values in the same way which solves a system of linear equations) and ***loop interchange***. ***Loop fusion*** contributes to the optimization of loop-level parallelism by decreasing the total number of iterations of the loops, iterations executed in parallel. I detailed in the thesis this optimization, which we implemented in the LLVM compiler [4] prior to the doctoral program, because it can be used together with the algorithms proposed in the thesis for the optimization of parallel programs that contain both task-level parallelism (optimized by proposed algorithms), as well as loop-level parallelism (optimized by the loop fusion algorithm), such as real-time client-server applications.

We have described and classified several mapping algorithms such as ***Limited Best Assignment*** (***LBA***) [8], ***Opportunistic Load Balancing*** (***OLB***) [8] and other Greedy algorithms described in [8], ***SCOTCH*** [12], ***METIS*** [13,14], the mapping algorithm part of the ***Zoltan*** utility set [14], algorithms that identify patterns in the task communication graph by the method described in [15], ***Treematch*** [16,17], ***EagerMap*** [18] and ***NUMA-BTDM*** [2] according to the criteria: the moment of mapping, the method of mapping used, the consideration or not of the harware architecture on which the application runs.

We have also described the factors that directly influence the efficiency of the thread mapping on NUMA systems. These are:

- Factors related to hardware architecture: the bandwidth, which is indicated to be as high as possible [19] and, less important than the bandwidth, the number of hops crossed in

the case of remote accesses [19], as well as the number of processors [20], which, the larger, the more important the problem of mapping and data locality;

- The factor related to the execution of parallel applications: balanced data locality on NUMA systems [20].

We have shown in this chapter that the allocation of memory in the context of data migration or threads migration, also contributes to the improvement of data locality. Thread schedulers such as the ***Completely Fair Scheduler*** (***CFS***) [21], currently used by ***Linux*** have (along with mapping algorithms) an important role because they focus on balancing the load and using resources properly [22, 23]. Thread schedulers do not have information about the dynamic behavior of the applications, which does not influence the scheduling. Scheduling refers mainly to the order of execution, as opposed to mapping, which decides the place of execution of the threads.

At the end of the chapter I explained some factors that increase the energy consumption of parallel processing compared to the sequential one, such as saving and restoring the execution context, scheduling the jobs or degrading the data locality [24]. Mapping algorithms aim to improve these energy-consuming aspects, as well as the performance by which, indirectly, energy consumption can be optimized.

3. Description of NUMA-BTLP and NUMA-BTDM algorithms

The research aims to optimize the parallel C / C ++ applications that use the ***PThreads*** Library [3] for the management of threads, through two algorithms, one for static classification of the threads and the other for their static mapping. Algorithms eliminate some of the disadvantages of not knowing the dynamic behaviorat compile-time, such as not knowing the number of threads. The algorithms optimize the execution time and power consumption of the applications by improving the balanced data locality when running these applications on NUMA systems.

Through a static analysis that classifies the execution threads into three types: autonomous, side-by-side and postponed (*NUMA-BTLP* algorithm [1]), static mapping according to the type of thread (*NUMA-BTDM* algorithm [2]) ensures the placement in execution of threads that use common data, on the same optimally identified cores and independent ones, on different cores.

The decrease of the power consumption is due to the decrease of the dynamic power, the decrease of the dynamic power is due to the optimization of the use of the cache memory, and the optimization of the use of the cache memory is realized by improving the balanced data locality on NUMA systems, obtained by the two algorithms.

Increased performance is achieved by optimizing the balanced data locality on NUMA systems, optimization due to the fact that the communicating threads are mapped on the same cores and the other threads are evenly distributed on the cores.

The two algorithms, *NUMA-BTLP* [25] and *NUMA-BTDM* [2], are implemented in the ***LLVM*** compiler [4]. In their implementation, the algorithms use two tree structures: the generation tree of the threads and the communication tree of the threads [25].

The thread generation tree is constructed according to the following rules [25]:

1. The main thread (executes the main function) is the root of the tree, representing the first level of the tree;
2. The threads created directly from the main function, through ***pthread_create*** calls, are the son nodes of the root, forming the second level in the tree;
3. The execution threads created directly from the functions attached to the threads created from the main function, represent the third level in the tree and so on.

After obtaining the generation tree of the threads, it is traversed in preorder and each

thread in the tree is assigned one type, this type being the output of a static analysis for the respective thread [25].

A thread is autonomous if there is no data dependency between it and each other thread. I considered that the autonomous thread does not have data dependencies when:

1. Another thread does not write any date that the autonomous thread reads;
2. The autonomous thread does not write any date read by any other thread;
3. The autonomous thread can read the data read only by all the threads.

The thread *i* is side-by-side relative to the thread *j* if there is at least one data dependency between them. The property is not usually transitive, that is, if the thread *i* is side-by-side relative to the thread *j* and the thread *j* is side-by-side realtive to the thread *k*, *i* is not side-by-side relative to *k*, unless the two use in common at least a data. Any two threads can be side-by-side, regardless of the position in the thread generation hierarchy. If one thread is side-by-side relative to at least one other thread, then it can no longer be autonomous.

A thread *i* is postponed if the thread has data dependencies only with the thread *j*, where *j* is the generating thread of the thread *i*. The thread generating the postponed thread executes with priority the other son threads, with which the postponed thread does not have data dependencies. The postponed thread does not write data read by the threads on the same or lower levels in the thread generation tree.

A second tree structure used by the two algorithms is the communication tree of the threads [25]. It is built on the basis of the following rules [25]:

1. If the thread is autonomous or postponed, add the thread as son of the generating thread;
2. If the thread is side-by-side, add the thread as son of all the threads in relation to which it is side-by-side, already added to the tree.

Static analysis for a thread implies the use of an algorithm that receives as its parameter the parent and its parent in the generation tree [1]. The analysis identifies the data dependencies between the thread and all the threads in the subtree that has the root the parent of the thread, if these dependencies exist [1]. With the help of specific classes in the ***LLVM*** compiler [4], the implementation of static analysis also takes into account the alias dependencies. The term alias refers to the same memory area indicated by two different variables.

Mapping threads (*NUMA-BTDM* algorithm [2]) implies mapping all autonomous threads, side-by-side threads and postponed threads respectively, in this order.

Mapping autonomous threads is done by traversing in preordered the communication tree and adding autonomous threads to a list. Then, the algorithm maps the nodes in the list evenly, on cores, as follows: divide the number of threads *i* into the number of cores *j* resulting the real number *k* and obtains the core for each thread by successively summing *k* with itself starting from to 0 and applying, the functions modulo to *j* (number of cores) and *floor*, to each partial result.

The mapping of the threads is done as follows:

1. Traverse the communication tree in inorder and set the CPU affinity for each node of the side-by-side type: make the reunion between the cores on which the node (i.e. thread) is executed and the cores on which the parent node is executed and set the CPU affinity of the side-by-side thread with the result.
2. The root and side-by-side threads on the first level are mapped, starting with core 0, except those side-by-side relative to the root (Point 1 applies).

The mapping of the postponed threads is done in the following way: the communication tree is traversed in postorder and each posponed thread identified is mapped in turn on the least loaded core at that time (from the application perspective, not the whole system). The core load is statically determined and represents the number of threads mapped to it at a given time. A postponed thread that has no other brother thread is set with priority as side-by-side.

In conclusion, I have detailed the sequence of calls of the two algorithms:

1. Identifying the number of logical cores and the number of logical cores per CPU at

compile-time (*NUMA-BTLP* [25]);
2. Creating the thread generation tree from the intermediate LLVM IR representation of the input code (*NUMA-BTLP* [25]);
3. Determining the type of each thread based on a static analysis and adding it to the communication tree (*NUMA-BTLP* [25]);
4. Mapping the threads based on the communication tree obtained in the previous step (*NUMA-BTDM* [2]);
5. Identifying in the intermediate LLVM IR representation of the input code, a ***pthread_create*** call, which creates the thread (*NUMA-BTLP* [25]);
6. Addying after each ***pthread_create***, a ***pthread_setaffinity_np*** call, which sets the CPU affinity of the thread created according to the mapping obtained in the previous step (*NUMA-BTLP* [1]).

4. Experimental results

Conclusions of the experimental results for the CPU-X real benchmark

Following the application of the *NUMA-BTDM* algorithm [2], the experimental results indicate an average optimization of 0.27 W / s of the power consumption of the NUMA system running the CPU-X real benchmark [26] and with 0.32 W / s of the UMA system, for a number of side-by-side threads less than 12. Optimization is obtained by subtracting from the average power consumption when the *NUMA-BTDM* algorithm [2] is not applied, the average power consumption when the algorithm is applied. The optimization variance is obtained in the same way, that is, subtracting from the average variance when the *NUMA-BTDM* algorithm [2] is not applied, the average variance when the algorithm is applied and has the value ~ 0.06 for the NUMA system and 0.13 for the UMA system. Due to the fact that the variance is smaller than the optimization, it can be concluded that there is an optimization. This is due to the increase in the cache hit rate, more precisely the rate of obtaining the data from the first and second levels of cache used in common by the side-by-side threads, mapped by the *NUMA-BTDM* algorithm [2] on the same core. Increasing the cache hit rate leads to an increase in the number of local accesses at the expense of remote ones (i.e. those accesses to another NUMA node), resulting the optimization.

According to the same type of reasoning as the one in the previous paragraph, the CPU consumption when running the CPU-X real benchmark is not optimized (an optimization of 0.17 W with a variance of the optimization of 1.92). This is due to the performance degradation caused by the large number of active-idle transitions and vice versa, the threads being all mapped on the same core. However, the optimization of power consumption due to the efficiency of the operations with memory is greater than the degradation due to the large number of active-idle transitions and vice versa, thus resulting the optimization of the power consumption of the whole system in the previous paragraph.

The experimental results also indicate that, as the number of side-by-side threads with high L1 and L2 cache hit rate increases, the higher the power consumption optimization, reaching a pick of 15% for less than 12 side-by-side threads.

Conclusions of the experimental results for the CPU real benchmark

The execution time on the NUMA system for the CPU real benchmark [27], composed of the CPU applications [27], Flops [27] and Iops [27], is optimized with a very small value (0.03 s relative to the total execution time of 600 s of Flops [27] and Iops [27] and 0.02 s compared to the total execution time of 0.59 s of the CPU application [27]), so it can be

considered that the execution time is kept with the optimization of the applications through the two algorithms.

In the case of Flops [27] and Iops [27], the optimization produced by the NUMA-BTDM algorithm [2] following the experiments, is not very large (0.03 s), since the mapping of a large number of threads (2400) produces a runtime degradation (due to ***pthread_setaffinity_np*** calls and parallelization overhead). However, the degradation is covered by the execution time gain resulting from the application of the *NUMA-BTDM* algorithm [2], finally obtaining a performance gain.

At CPU [27], the same runtime is retained (as in the case of the CPU-X real benchmark [26]), but it reduces the power consumption of the NUMA system by 0.9 W / s and of the UMA system with 7.56 W / s. The variance of the optimization is also calculated as the difference between the average variance of the power consumption measurements of the system running the non-optimized application and the average variance of the power consumption measurements of the system on which the optimized application runs. The optimization variance is much greater than the optimization itself, so the CPU application [27] is not considered optimized. Optimization of the execution of the Flops application after applying the *NUMA-BTDM* algorithm [2] on the NUMA system is 0.6 W / s and 0.3 W / s on the UMA system. This is considered to be an optimization given that the optimization variance is smaller than the optimization itself (the optimization variance has the value 0.14). The Iops application [27] is not optimized by the *NUMA-BTDM* algorithm [2], resulting in higher power consumption on average when the algorithm is applied than when it is not applied.

Conclusions of the experimental results for the Context Switch real benchmark

The execution time of the Context Switch real benchmark [28] on the NUMA system is not improved when applying the *NUMA-BTDM* algorithm [2]. Due to the small number of threads that the application uses, namely two autonomous execution threads, mapped to different cores by the algorithm, the time with which the total execution time of the application is reduced, following the application of the NUMA-BTDM algorithm [2], is smaller than the execution time consumed with the actual mapping of the execution threads on cores through ***pthread_setaffinity_np*** calls, resulting an insignificant performance degradation (0.44 s, compared to 41.9 s, the average running time of the application).

The experimental results indicate an average optimization with 0.32 W / s of energy consumption for the Context Switch real benchmark [28] running on the NUMA system using two autonomous threads, i.e. an optimization of up to 5%, while maintaining the same execution time. The reduction of energy consumption is caused by the reduction due to the mapping of the number of active-idle transitions and vice versa, of the threads.

5. Conclusions and personal contributions

NUMA-BTDM [2] is a mapping algorithm, applied at compile-time to parallel applications, which decides the CPU affinity of each thread based on its type. The thread type is returned based on the static characteristics of the code by the *NUMA-BTLP* algorithm [1] which classifies the threads into autonomous, side-by-side and postponed following a static analysis of the data dependencies. *NUMA-BTLP* [1] and *NUMA-BTDM* [2] contribute to achieving balanced data locality, optimizing the mapping of threads on NUMA systems. After applying the two compile-time algorithms on the parallel applications that use the PThreads Library [3] in order to obtain the task-level parallelism, the task-level parallelization will be combined with the balanced data locality at runtime, which optimizes the energy consumption.

NUMA-BTDM [2] uses the ***PThreads*** Library [3] to set the CPU affinity of each thread, allowing the threads to run as close in terms of NUMA time and distance to the data they use.

The novelty of the two algorithms are:

1. The ability to allow parallel C / C ++ applications that use PThreads [3] to customize and control thread mapping based on the static characteristics of the code, rather than allowing the operating system to perform this mapping randomly.
2. Eliminating the disadvantage of not knowing the dynamic appearance of the number of threads, in the compilation phase, by inserting at compilation a ***pthread_setaffinity_np*** call immediately after each ***pthread_create*** call, the call mapping the thread(s) created by the ***pthread_create*** call, according to the *NUMA-BTLP* algorithm [1], regardless of their number and eliminating the disadvantage of the unpredictability of the amount of latencies of the remote accesses during the compilation phase, by favoring the local accesses to the detriment of the remote ones as a result of mapping the side-by-side threads using the same data, on the same cores
3. Defining some original static criteria for classifying the execution threads into 3 categories and defining these categories
4. Mapping threads according to their type
5. Integration of the algorithms for classifying the threads and mapping them into a modern compiler
6. Mapping threads using 2 trees: one describing data dependencies and another describing the generation hierarchy [25]
7. Ensure the portability of the algorithms on any NUMA architecture running the Linux operating system by inserting in the source code, when compiling, some system calls that execute commands through which the number of cores and the number of logical cores per CPU of the hardware architecture are identified at the execution of the parallel application, required to run the *NUMA-BTDM* mapping algorithm [2]

NUMA-BTDM [2] is one of the few compile-time optimizations dedicated to a particular type of system, in this case dedicated to NUMA systems. In addition to this, these algorithms obtain the balanced data locality in a new way introduced by this work, namely: the balanced distribution on cores of the autonomous threads, the proximity in time and distance NUMA of the side-by-side threads to the data and the existence of postponed threads that do not "steal" the cache from critical threads that require immediate execution, being mapped to the least loaded core.

Although the *NUMA-BTLP* [1] algorithm inserts, at compile-time, additional function calls that set the CPU affinity of each thread, the *NUMA-BTLP* [1] algorithm does not degrade either the runtime or the power consumption of NUMA or UMA systems, for tested applications, but improves the runtime and power consumption for small number of autonomous threads and only power consumption for large number of autonomous threads and a small number of side-by-side threads.

Bibliografie

[1] Iulia Știrb. „NUMA-BTLP: A static algorithm for thread classification”. In 2017 5th International Conference on Control, Decision and Information Technologies (CoDIT)”, p. 882–887. IEEE, 2017.

[2] Iulia Știrb. „NUMA-BTDM: A thread mapping algorithm for balanced data locality on NUMA systems”. In 2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT), p. 317–320. IEEE, 2016.

[3] POSIX Threads. 2017.
[4] The LLVM Compiler Infrastructure Project. https://llvm.org/, 2018. Accessed: 9.10.2018.
[5] Nakul Manchanda și Karan Anand. Non-uniform memory access (NUMA). New York University, 4, 2010.
[6] Manu Awasthi, David Nellans, Kshitij Sudan, Rajeev Balasubramonian, și Al Davis. „Handling the problems and opportunities posed by multiple on-chip memory controllers". In 2010 18th International Conference on Parallel Architectures and Compilation Techniques (PACT), p. 318–330. IEEE, 2010.
[7] Matthias Diener, Eduardo HM Cruz, Marco AZ Alves, Philippe OA Navaux, și Israel Koren. „Affinity-based thread and data mapping in shared memory systems". ACM Computing Surveys (CSUR), 49(4):64, 2017.
[8] Robert Armstrong, Debra Hensgen, și Taylor Kidd. „The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions". In Proceedings Seventh Heterogeneous Computing Workshop (HCW 98), p. 79–87. IEEE, 1898.
[9] Haris Ribic și Yu David Liu. "Energy-efficient work-stealing language runtimes". In ACM SIGARCH Computer Architecture News, vol. 42, p. 513-527. ACM, 2014.
[10] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, și Raymond Namyst. „hwloc: A generic framework for managing hardware affinities in hpc applications". In 2010 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 170–176. IEEE, 2010.
[11] Imen Fassi și Philippe Clauss. "Xfor: filling the gap between automatic loop optimization and peak performance". In 2015 14th International Symposium on Parallel and Distributed Computing (ISPDC), p. 100–109. IEEE, 2015.
[12] François Pellegrini și Jean Roman. „Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs". In International Conference on High-Performance Computing and Networking, p. 493–498. Springer, 1896.
[13] George Karypis și Vipin Kumar. „A fast and high quality multilevel scheme for partitioning irregular graphs". SIAM Journal on Scientific Computing, 20(1):359–392, 1898.
[14] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, și Umit V Catalyurek. „Parallel hypergraph partitioning for scientific computing". In Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, p. 10–pp. IEEE, 2006.
[15] Eduardo HM Cruz, Matthias Diener, și Philippe OA Navaux. „Using the translation lookaside buffer to map threads in parallel applications based on shared memory". In 2012 IEEE 26th International Parallel and Distributed Processing Symposium, p. 532–543. IEEE, 2012.
[16] Hao Zhou și Jingling Xue. „A compiler approach for exploiting partial SIMD parallelism". ACM Transactions on Architecture and Code Optimization (TACO), 13(1):11, 2016.
[17] Emmanuel Jeannot și Guillaume Mercier. „Near-optimal placement of MPI processes on hierarchical NUMA architectures". In Euro-Par 2010-Parallel Processing, p. 189–200, 2010.
[18] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, și Philippe OA Navaux. „An efficient algorithm for communication-based task mapping". In 2015 22rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p. 197–204. IEEE, 2015.
[19] Baptiste Lepers, Vivien Quéma, și Alexandra Fedorova. „Thread and memory placement on NUMA systems: Asymmetry matters". In USENIX Annual Technical Conference, p. 267–279, 2015.
[20] Timothy Brecht. „On the importance of parallel application placement in NUMA multiprocessors". In Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV), p. 1–17, 1893.

[21] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, și Fun Wey. „Towards achieving fairness in the Linux scheduler”. ACM SIGOPS Operating Systems Review, 42(5):34–43, 2008.
[22] David Tam, Reza Azimi, și Michael Stumm. „Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors”. In ACM SIGOPS Operating Systems Review, vol. 41, p. 47–58. ACM, 2007.
[23] Tong Li, Dan Baumberger, și Scott Hahn. „Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin”. In ACM Sigplan Notices, vol. 44, p. 65–74. ACM, 2009.
[24] Candy Pang, Abram Hindle, Bram Adams, și Ahmed E Hassan. „What do programmers know about software energy consumption?”. IEEE Software, 33(3):83–89, 2016.
[25] Iulia Știrb. „Extending NUMA-BTLP algorithm with thread mapping based on a communication tree”. Computers, 7(4):66, 2017.
[26] Cpu-benchmarking. https://github.com/pdpriyanka/CPU-Benchmarking, 2016. Accessed: 16.08.2018.
[27] Cpu-x. https://github.com/X0rg/CPU-X, 2018. Accessed: 16.08.2018.
[28] contextswitch. https://github.com/tsuna/contextswitch, 2016. Accessed: 16.08.2018.