A parallel version of the in-close algorithm

KODAGODA, Nuwan, ANDREWS, Simon <http://orcid.org/0000-0003-2094-7456> and PULASINGHE, Koliya

Published version

KODAGODA, Nuwan, ANDREWS, Simon and PULASINGHE, Koliya (2017). A parallel version of the in-close algorithm. In: NCTM 2017 Proceedings of the 6th National Conference on Technology and Management (NCTM). IEEE, 1-5.

A Parallel version of the In-Close Algorithm

Nuwan Kodagoda
Department of IT
Faculty of Computing
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
nuwan.k@sliit.lk

Simon Andrews
Conceptual Structures Research Group
Communication and Computing Research Centre
Faculty of Arts, Computing, Engineering and Sciences
Sheffield Hallam University, Sheffield, UK
s.andrews@shu.ac.uk

Koliya Pulasinghe
Department of IT
Faculty of Computing
Sri Lanka Institute of Information Technology
Malabe, Sri Lanka
koliya.p@sliit.lk

Abstract—This research paper presents a new parallel algorithm for computing the formal concepts in a formal context. The proposed shared memory parallel algorithm Parallel-Task-In-Close3 parallelizes Andrews's In-Close3 serial algorithm. The paper presents the key parallelization strategy used and presents experimental results of the parallelization using the OpenMP framework.

Keywords— FCA, In-close, parallel, OpenMP, formal concept analysis

I. INTRODUCTION

Formal Concept Analysis (FCA) is a contemporary data mining and data analysis technique for object-attribute relational data. A formal context describes a binary relationship between a set of objects and a set of attributes of a domain. A precise definition of a formal context is given below.

A formal context is defined as $K := (X,Y, I)$

Where X is a set of objects, Y a set of attributes and I a binary incidence relationship between X and Y with $I \subseteq X \times Y$. Since formal contexts are a binary relationship they can be represented as cross tables. Here each object and attribute is represented as a row and a column respectively [1][2]. A mathematical definition of Formal Concepts is given below.

For a set of objects $A \subseteq X$ the set A' is defined as

$A' := \{ y \in Y \mid y\ I\ x \text{ for all } x \in A\}$

Similarly for a set of attributes $B \subseteq Y$ the set B' is defined as

$B' := \{ x \in X \mid y\ I\ x \text{ for all } y \in B\}$

(A,B) is a formal concept if $A' = B$ and $B' = A$.

There are many formal concepts in a formal context. All the possible formal concepts that are there in a formal context can be generated and be represented in a concept hierarchy.

FCA has been applied a wide range of disciplines. A comprehensive survey of the usage of FCA in the area of Knowledge Processing in a wide range of domains which includes software mining, web analytics, medicine, biology and chemistry data is presented in [3][4].

II. NEED FOR PARALELIZATION

All computing devices used today are parallel machines. The introduction of multicore processors commenced around the year 2004 to solve the so called power wall problem. Prior to this CPU manufacturers resorted to increase the clock speed of each new generation of CPU eventually reaching the critical power consumption of 130 Watts around 2004. Beyond this point it was not economically possible to dissipate the heat produced by the CPU's. Over the last decade CPU manufacturers have kept the clock speed and core size of a CPU as constants and have resorted instead to add extra cores to a single die in the CPU to get better performance [5]

Today's laptops, desktop machines have at least two to four cores in the CPU. High end Xeon Processors have up to 24 cores. The latest high end Xeon Phi processors have up to 72 cores, where each core has the power of a single Intel Atom processor[6].

Computer programs must be designed and implemented using a parallel approach to leverage on the multiple cores available in the CPU[7]. Traditional serial programs can only make use of one CPU core of the computer.

III. PARALELL ARCHITECTURES

Today's computers are essentially Shared Memory Multiple Instructions, Multiple Data (MIMD) machines. They typically also support vector operations which are Single Instruction, Multiple Data (SIMD). The shared memory model simplifies the transactions between the CPUs. However this also constitutes a bottleneck and limits the scalability of the system[8]. Intel's new highly parallel many core CPU the Xeon Phi processor family have up to 72 cores running up to 288 threads with 512 bit vector instructions [6].

Distributed memory Multiple Instructions, Multiple Data (MIMD) machines are the other type of parallel machines that are available. These machines are made up of processors that communicate by exchanging messages. The communication cost is high, but since memory is not shared, such machines can scale well. Clusters and Supercomputers are examples of such machines [8].

The parallel algorithm presented in this research paper is a shared memory parallel algorithm.

IV. Sequential Algorithm – In-Close3

The In-Close3 algorithm was originally described by Andrews [9]. Kodagoda and Pulasinghe[10] in their comparison of eight different variations of Kuznetsov's CbO[11] family of algorithm confirmed that the algorithm CbO-PC-ICF-BF (In-Close3) is the fastest serial FCA algorithm. The pseudo code presented here has an additional parameter level, which is used to keep track of the recursion level.

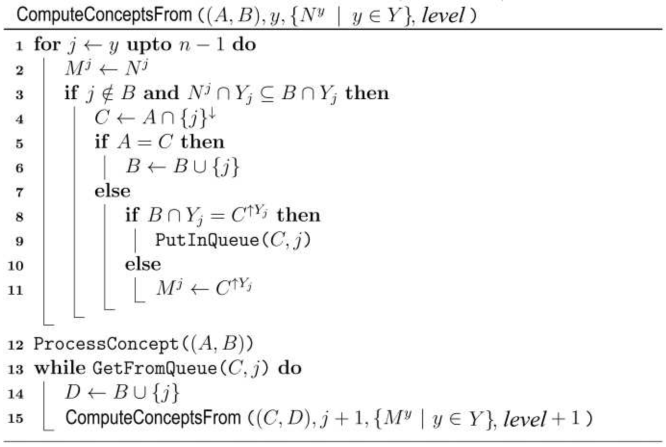
Here (A,B) is the concept generated where A is the extent and B is the intent. y is the attribute that is considered. In line 4 the next extent C is computed by intersecting the existing extent A with each column of the formal context to find every column that contains the extent. $j \notin B$ in Line 3, enables skipping attributes in the current intent [12]. Here if the currently considered attribute j is already a member of the currently considered intent then the extent of this has already been computed. This is due to the following observation [10]

$$\{0,1,2,,,k,,n\}^{\downarrow} = \{0,1,2,,,k,,n\}^{\downarrow} \cap \{k\}^{\downarrow}$$

The key feature of the In-Close family of algorithms is the use of partial closures [9]. Line 11 and 14 computes $C^{\uparrow Yj}$, the partial closure of C where the context I is examined upto the current attribute j.

A full closure operator ↑ is equivalent to ↑Y where Y is the set of all attributes in context I. The partial closure operator ↑Y is defined as follows [9].

$$A^{\uparrow Z} := \{y \in Z \mid \forall x \in A : x\, I\, y\}$$



```
ComputeConceptsFrom ($(A, B), y, \{N^y \mid y \in Y\}, level$)
1  for $j \leftarrow y$ upto $n-1$ do
2      $M^j \leftarrow N^j$
3      if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
4          $C \leftarrow A \cap \{j\}^{\downarrow}$
5          if $A = C$ then
6              $B \leftarrow B \cup \{j\}$
7          else
8              if $B \cap Y_j = C^{\uparrow Y_j}$ then
9                  PutInQueue($C, j$)
10             else
11                 $M^j \leftarrow C^{\uparrow Y_j}$
12 ProcessConcept($(A, B)$)
13 while GetFromQueue($C, j$) do
14     $D \leftarrow B \cup \{j\}$
15     ComputeConceptsFrom ($(C, D), j+1, \{M^y \mid y \in Y\}, level+1$)
```

Fig 1, In-Close3 algorithm pseudo code

The failed canonicity test is defined by the condition given in line 3.

$$N^j \cap Y_j \subseteq B \cap Y_j$$

In essence the cannocity test prevents the recomputation of previously generated concepts. This is due to the lexical order of computing concepts in the CbO family of algorithms. Here Y_j is a set containing all the attributes upto attribute j. M^j and N^j are used to capture the intent of failed cannocity tests in the algorithm. Initially M^j is set to the intent of the previously failed cannocity test N^j in line 2. If there are failed cannocity tests at attribute level j, the value of $C^{\uparrow Yj}$ is captured in M^j in line 11. This is passed to the algorithm during the recursion call as parameter N^j. Thus it implies that the concept has been computed before and can be skipped. The extents are computed in line number 4. The intents are incrementally computed in line 6 and 14. A queue is used in line 9 and 13 to use a combined depth first and breadth first strategy in computing the concepts.

V. Parallel Task – In-Close3 Algorithm

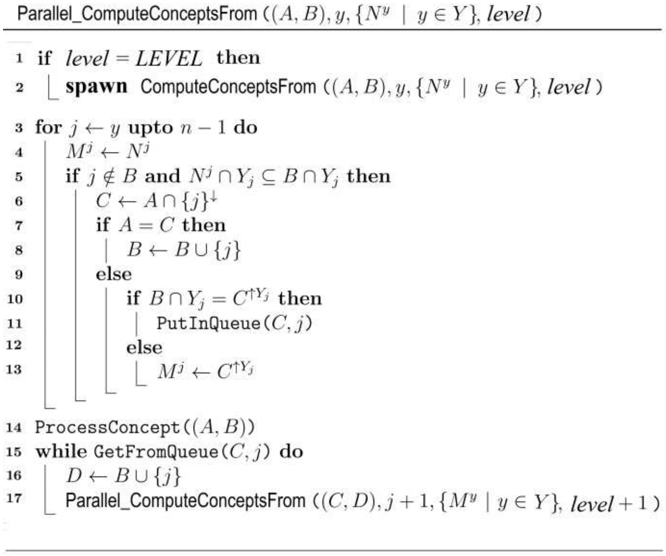
Huaiguo Fu had created a parallel implementation of the NextClosure algorithm but it was limited to 50 attributes [13]but this was subsequently greatly extended [14]. Krajca [15] presented a parallel algorithm called PFCbO which parallelizes the FCbO algorithm. This is also a variation of the CbO algorithm[11]. Andrews's best of breeds In-Close3 is an improvement over the serial FCbO algorithm, where the key difference is the use of partial closures instead of full closures[9]. Krajca had used a queue specific to each thread to capture parameters of recursive sub call trees of a specific level of recursion[15]. Once all the sub call trees are captured, instances of threads are spawned in a round robin fashion to compute the remaining concepts in parallel.

In-Close3 is a naturally recursive algorithm. In the serial implementation each of the recursive call will be executed by the same processor. The recursion will occur in a combined depth first and breadth first approach as showed in Fig 3. The numbers indicate the order of execution. The breadth first traversal is due to the use of a queue in line 9 and 13 of the original In-Close3 algorithm.

A simple naïve parallelization strategy would be to spawn each recursive call as a separate thread running on a separate core. One of the challenges of parallel programming is to allocate each thread sufficient work and solutions typically scale well if the work assigned to each thread is uniform. The naïve parallelization approach outlined above should in theory support uniform workload distribution but has the two following drawbacks. One being that the number of threads spawned would be significant even for modest datasets. Secondly the workload provided to the threads would be very small resulting in the threads swapping the workload provided frequently. An experimental evaluation of this naïve parallelization approach yielded poor performance.

The proposed solution uses an approach similar to that of Krajca's PCbO[15] where an entire recursion subtree is assigned to each thread. The parallel algorithm consists of two functions Parallel_ComputeConceptsFrom() and ComputeConceptsFrom().

The Parallel_ComputeConceptsFrom() function (See Fig 2) is identical to the ComputeCoonceptsFrom() function (See Fig 1) with the exception of the first two lines. The Parallel_ComputeConceptsFrom() function is invoked with $(A,B)=(X,X^{\uparrow})$. Where X represents a complete set of extents. Initial attribute y = 0 and a set of empty $N_s, \{N^y = \emptyset \mid y \in Y\}$ and level = 0. These values are the same as that is used for the serial algorithm presented in Fig 1. The parameter level is used to keep track of the level of recursion. The constant LEVEL is an optimization parameter that is used to determine the recursion level at which separate processes are spawned with the task of computing all the concepts in a given recursive sub tree (See line 2 of Fig 2). For instance if the constant LEVEL is set to two for the recursive call tree given in Fig 3, the tasks that would be assigned to the parallel threads would be {5,{6,{8,9,10}},7,{11,{14}},12,{13,{15,16}},17,{18,{19,20}. The first available thread would be assigned the task of computing concepts of the recursive sub tree 5. However since there are no children in this sub tree the thread would complete the task as soon as the concepts of 5 are computed. The next available thread in the meantime would have been assigned the task of computing the concepts of the recursive sub tree 6. During the computation the same thread is used to compute the concepts of 8,9 and 10 which are discovered and computed at runtime. It is clear in this example that the workloads given to each thread would be different. This is one of the disadvantages of this proposed solution. Another is the fact that concepts upto LEVEL two are computed serially. In the above example the concepts for 1,2,3 and 4 are computed serially. The same disadvantages are there in Krajca's parallel solution as well.



```
Parallel_ComputeConceptsFrom ($(A, B), y, \{N^y \mid y \in Y\}, level$)

1  if $level = LEVEL$ then
2      spawn ComputeConceptsFrom ($(A, B), y, \{N^y \mid y \in Y\}, level$)
3  for $j \leftarrow y$ upto $n - 1$ do
4      $M^j \leftarrow N^j$
5      if $j \notin B$ and $N^j \cap Y_j \subseteq B \cap Y_j$ then
6          $C \leftarrow A \cap \{j\}^{\downarrow}$
7          if $A = C$ then
8              $B \leftarrow B \cup \{j\}$
9          else
10             if $B \cap Y_j = C^{\uparrow Y_j}$ then
11                 PutInQueue($C, j$)
12             else
13                 $M^j \leftarrow C^{\uparrow Y_j}$
14 ProcessConcept($(A, B)$)
15 while GetFromQueue($C, j$) do
16     $D \leftarrow B \cup \{j\}$
17     Parallel_ComputeConceptsFrom ($(C, D), j + 1, \{M^y \mid y \in Y\}, level + 1$)
```

Fig 2, Parallel Task - In-Close3 algorithm
Parallel_Compute_Concepts_From() pseudo code

Krajca used separate queues to store each of the recursive call subtree workloads that were later distributed to separate threads in a round robin fashion[15]. The storing was done serially and the spawning of threads was carried out only after the computation of all the recursive call subtrees. The parallel algorithm proposed in this research paper spawns new threads as soon as they are discovered.

VI. Implementation Details

The OpenMP command task was used to spawn new threads. High level shared memory thread programming frameworks such as OpenMP, Cilk+ have built in schedulers that are used to spawn threads. Developers only implicitly specify the intent of parallelization using appropriate commands in the code[5]. A sophisticated runtime scheduler in the background handles the creation, assigning work and deletion of threads.

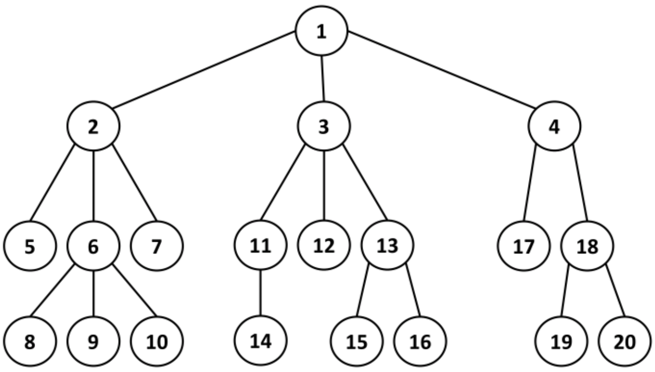


Fig 3, Combined Depth and Breadth First Recursive Call Tree of In-Close3

The OpenMP task command has an inherent queue which keeps track of requests made to spawn new threads. It retrieves requests from its internal queue and distributes the workload to threads which are free. This ensures that the implementations scale well for different shared memory machines with varied number of cores.

Table 1, No of threads called in parallel for different values of LEVEL

Level	Mushroom	Adult	Ad	Example
0	1	1	1	1
1	35	91	371	3
2	398	1,505	2,041	8
3	2,307	8,722	4,051	8
4	8,261	22,259	4,138	0
5	20,358	26,167	3,003	0

Table 1, shows the number of threads that are executed in parallel for different values of LEVEL. A LEVEL with a higher value seem to be ideal to parallelize. However it implies that the nodes above that LEVEL are computed serially.

VII. Experimental Results

The OpenMP implementation of the Parallel Task - In-Close3 algorithm was executed on a single node of the ARCHER Super Computer . An ARCHER node has two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) series Xeon processors. The two processors are connected by two QuickPath Interconnect (QPI) links. The memory is arranged in a non-uniform access (NUMA) form, where each 12-core processor is a single NUMA region with local memory of 32 GB. By nature a super computer provides dedicated access of the compute nodes required to run a program. The graph shown in Fig 4 shows the relative speedup of running the real world datasets Mushroom, Adult and Internet Ads[16].

The values were computed with the LEVEL set as two. Speedup is defined as the ratio between the time taken to run the dataset with one processor over the time taken to run the same dataset with P processors[17]. Table 2, shows the best time obtained for each of the different real world datasets.

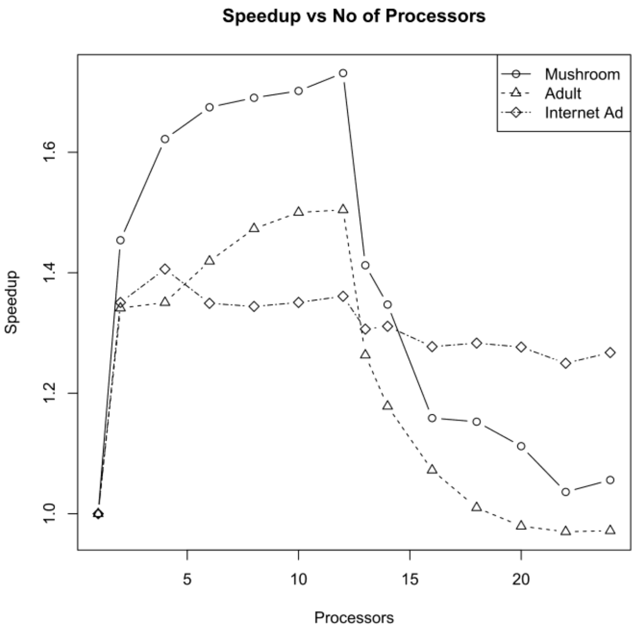


Fig 4, Speedup vs No of Processors for the Mushroom, Adult and Internet Ad real world datasets

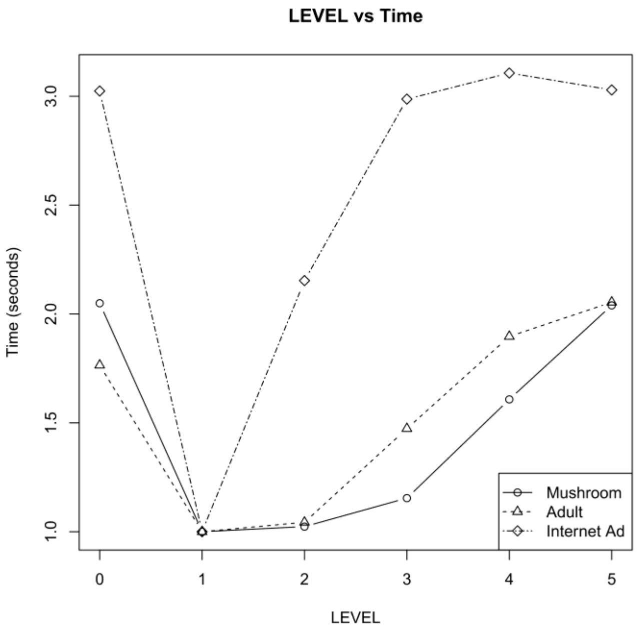


Fig 5, LEVEL vs Time for the Mushroom, Adult and Internet Ad real world datasets

There is a clear drop in performance when then core count reaches thirteen (See Fig 3). This can be easily explained when one looks at the CPU configuration of the machine used to run the experiments. Access to the local memory by cores within a NUMA region has a lower latency than accessing memory on the other NUMA region.

Table 2, Best Time obtained for different datasets

Data Set	Mushroom	Adult	Internet Ads
|X| x |Y|	8,124x125	32,561x99	3279x1565
Density	17.4%	11.29%	0.97%
# Concepts	226,920	80,332	16570
Time (seconds)	0.07667	0.02972	0.05771
Best Results (cores)	12	12	04

The current codebase has room for optimization by the removal of mutexes and avoiding false memory sharing. The parallel algorithm is memory bound which is also another reason why it doesn't scale well.

Fig 5, shows how the implementation behaved for different values of LEVEL. When LEVEL is zero all the concepts are computed in one single thread. We can see a significant drop in performance for this value. The best results are obtained when LEVEL is set to one. However results when LEVEL is set to two is similar for the mushroom and adult datasets. Krajca had reported best results when LEVEL had the value of two[15].

The original serial implementation of In-Close3 had a large scratchpad of memory allocated at the start of the program to capture the extents and intents produced. In shared memory programming care should be taken to avoid data races. A data race occurs when two or more threads in a single process access the same memory location concurrently, and at least one of the accesses is for writing[18]. For the parallel implementation the scratchpad were separated so that each processor would have separate access to its own scratchpad.

VIII. Conclusion

The results demonstrate that CbO based algorithms which are naturally recursive by nature, can be easily parallelized with only minor changes to the codebase. OpenMP tasks can be used for this purpose where an entire recursive call sub tree can be assigned to separate threads. Further research needs to be carried out to compare the parallel implementation of In-Close3 to other parallel implementations such as PFCbO.

Acknowledgements

This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts," in Ordered sets, Springer, 1982, pp. 445–470.

[2] B. Ganter, G. Stumme, and R. Wille, "Formal concept analysis: Methods and applications in computer science," TU Dresden, http//www. aifb. uni-karlsruhe. de/WBS/gst/FBA03. shtml, 2002.

[3] J. Poelmans and S. Kuznetsov, "Formal concept analysis in knowledge processing: a survey on models and techniques," Expert Syst. with …, vol. 40, no. 2003, pp. 1–40, 2013.

[4] J. Poelmans, D. Ignatov, and S. Kuznetsov, “Formal concept analysis in knowledge processing: A survey on applications,” Expert Syst. Appl., vol. 40, no. 16 SRC-GoogleScholar FG-0, pp. 6538–6560, 2013.

[5] S. Chappell and A. Stokes, Parallel Programming with Intel Parallel Studio XE. John Wiley & Sons, 2012.

[6] J. Jeffers, J. Reinders, and A. Sodani, Intel Xeon Phi Processor High Performance Programming, 2nd Edition. Morgan Kaufmann, 2016.

[7] H. Sutter, “The free lunch is over: A fundamental turn toward concurrency in software,” Dr. Dobb’s J., vol. 30, no. 3, pp. 202–210, 2005.

[8] G. Barlas, Multicore and GPU Programming: An integrated approach. Elsevier, 2014.

[9] S. Andrews, “A ‘Best-of-Breed’ approach for designing a fast algorithm for computing fixpoints of Galois Connections,” Inf. Sci. (Ny)., vol. 295, pp. 633–649, 2015.

[10] N. Kodagoda and K. Pulasinghe, “Comparision Between Features of CbO based Algorithms for Generating Formal Concepts”, Int. J. Concept. Struct. Smart Appl., vol. 4, no. 1, pp. 1–34, 2016.

[11] S. O. Kuznetsov and S. a. Obiedkov, “Comparing performance of algorithms for generating concept lattices,” J. Exp. Theor. Artif. Intell., vol. 14, no. 2–3, pp. 189–216, 2002.

[12] V. Vychodil, “A New Algorithm for Computing Formal Concepts,” Proceeding 19th EMSCSR, pp. 15–21, 2008.

[13] H. Fu and E. M. Nguifo, “A Parallel Algorithm to Generate Formal Concepts for Large Data,” in International Conference on Formal Concept Analysis (ICFCA), 2004, pp. 394–401.

[14] H. Fu and M. O. Foghlu, “A distributed algorithm of density-based subspace frequent closed itemset mining,” in High Performance Computing and Communications, 2008. HPCC’08. 10th IEEE International Conference on, 2008, pp. 750–755.

[15] P. Krajca, J. Outrata, and V. Vychodil, “Parallel recursive algorithm for FCA,” in CLA, 2008, vol. 2008, pp. 71–82.

[16] A. Frank and A. Asuncion, “UCI machine learning repository.” 2010.

[17] M. D. McCool, A. D. Robison, and J. Reinders, Structured parallel programming: patterns for efficient computation. Elsevier, 2012.

[18] A. Vladimirov and V. Karpusenko, “Parallel Programming and Optimization with Intel Xeon Phi Coprocessors,” ColeFax Int., no. May, p. 520, 2013.