Towards a Catalog of Performance Smells for Parallel Computing

BHARATKUMAR SHARMA, SIEMENS TECHNOLOGY AND SERVICES PVT. LTD., INDIA
GIRISH SURYANARAYANA, SIEMENS TECHNOLOGY AND SERVICES PVT. LTD., INDIA

Parallel computing architectures have been proven to significantly boost performance and are, therefore, being increasingly adopted in industrial applications. However, our experience with applications built on parallel computing architectures reveals that often they are not designed properly to leverage the power of these architectures and consequently suffer from performance "smells". We believe that an awareness of such smells can help practitioners design better and faster applications. However, our study of the existing literature on performance smells reveals the lack of a generalized comprehensive smell catalog that focuses on parallel computing. To address this gap, we have aggregated and cataloged commonly-occurring performance smells related to parallel computing. In this paper, we describe these smells along with their associated mitigation techniques, and attempt to draw out the relationships between different smells.

General Terms: Performance Smells

Additional Key Words and Phrases: Multi-Core, Many-Core, GPU, Parallel Computing

ACM Reference Format:

Sharma, B. and Suryanarayana, G. 2015. Towards a Catalog of Performance Smells for Parallel Computing. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2015), 9 pages.

1. INTRODUCTION

Complex real-world applications such as those that control and manage power grids, industrial automation, and healthcare delivery are of a critical nature, and mandate real-time decision-making in response to rapidly arriving information. When such applications perform poorly, it not only leads to loss in user confidence and credibility but can also lead (in severe cases) to serious injuries and loss of human life and property. For example, in Healthcare systems like MRI (Wright 1997), CT (Kato 1989) etc, faster 3D reconstruction may mean the patient will be exposed to harmful radiation for a lesser time period. Performance is, thus, one of the most ubiquitous quality requirements in real-world applications, and has consequently inspired many approaches for performance tuning and engineering.

One such approach that is being widely and rapidly adopted is the use of parallel computing. In fact, in domains like healthcare, oil and gas, and finance it has become necessary to consider parallel architectures as part of the overall design to achieve the desired performance requirements. With the advent of the multi-core/many-core era, a multitude of complex parallel architectures have emerged rapidly. For instance, Nvidia releases a new parallel architecture every 2 years. This requires designers and architects to keep themselves abreast of the latest parallel architectures and design suitably taking into consideration the constraints introduced due to the new parallel architecture being used. Failing to do so can result in the occurrence of "performance smells"; a performance smell in the context of parallel computing is an indication of a possibly non-optimal or wrong design decision for an underlying parallel architecture that negatively impacts the performance of the application.

It is possible that a particular technique which improves the performance when applied to a certain parallel architecture may degrade the performance resulting in a smell when applied to a different parallel architecture. For example, in CUDA (Garland 2010) architecture for Nvidia GPU (Graphics Processing Unit), it is recommended to launch more threads than the number of available cores to hide the latency; however, the same optimization when applied on multi-core CPU (Central Processing Unit) architecture

This work is supported by Research and Technology Center, Siemens Technology and Services Pvt. Ltd., India
Author's: Sharma, B. email: bharatkumar.sharma@siemens.com; Suryanarayana, G. email: girish.suryanarayana@siemens.com

may result in performance degradation because more time is spent in context switching rather than executing.

Clearly, it is important for software engineers to understand the interplay of their design decisions with the underlying parallel architecture to prevent such performance smells. However, a study of the existing literature reveals the lack of a generalized comprehensive performance smell catalog that focuses on parallel computing. To address this gap, we have created a catalog of commonly-occurring performance smells related to parallel computing that software engineers can refer to while applying a performance related technique/decision to a particular parallel architecture. In this paper, we present our catalog and describe each smell using an example along with the associated mitigation techniques for that smell.

The rest of the paper is organized as follows. Section 2 describes related work and their shortcomings that have motivated our work. Section 3 introduces our performance smells catalog. For each smell, it includes the definition, the signs that indicate the smell, a relevant example, and the associated mitigation techniques to address that smell. During our exploration, we have found that often smells are related to each other, and we have attempted to document these relationships in Section 4. Section 5 concludes with a short summary of our work.

2. BACKGROUND AND RELATED WORK

Before we delve into the shortcomings of existing literature on performance smells, it is important to first discuss briefly what a "performance smell" is and the various connotations of the term. The term "smell" is a well-known term in software engineering and has been used to denote problems in code or design via terms such as "code smells" and "design smells" respectively (Fowler et al. 1999) (Ganesh et al. 2011). We extend this term to the topic of performance in the context of parallel computing, and define a performance smell as "an indication of a possibly non-optimal or wrong design decision for an underlying parallel architecture that negatively impacts the performance of the application".

A study of the literature reveals that considerable attention has been devoted to performance-related smells. Interestingly, much of the work has come in the context of performance anti-patterns which denote poor recurring solutions that cause performance problems. For instance, Connie Smith presents a set of anti-patterns related to performance (Connie et al. 2000) (Smith et al. 2003). However, her work focuses on general design decisions and does not concentrate on issues related to multi-core/many-core architectures. Hallal et al. discuss performance smells but their discussion is limited to Java (Hallal et al. 2004). Similarly, Bradbury and Jalbert present a catalog of programming anti-patterns for concurrent Java (Bradbury and Jalbert 2009). In a similar fashion, there exists a catalog of anti-patterns which is focused only on the .NET framework (Stephen 2010). Such catalogs are specific to a particular language or platform and will require an equivalent mapping to the concepts of other languages, frameworks, or architectures in order to be applicable to them. Further, a new era of heterogeneous computing is emerging where different parallel architectures are present on the same die/chip. Due to this, application designers are moving to higher level frameworks (Bell and Hoberock 2011) that provide a layer of abstraction to help target many parallel architectures using the same design. In such a case, there is a need for a performance smells catalog which is at higher level of abstraction and applies to multiple parallel architectures.

In this context, we would like to point out that in our experience, the term anti-pattern is often a confusing term and gives an impression that there is something wrong with the original pattern. Hence, we prefer to use the term "smell" instead of anti-pattern to denote an indication of a possibly poor design solution that affects performance. There are accounts in the literature that present optimal solutions for specific parallel architectures such as CUDA GPU (Garland 2010) and multi-core (Diamond et al. 2011). However, these do not provide a generic view of performance smells across parallel architectures.

To provide a generic view of performance smells, it becomes necessary to provide abstractions for terminology used across parallel architectures. Figure 1 shows these abstract terms and the relationships between them. These terms (which will be referred in the rest of the paper) are:

- **Task**: An independent piece of work/module which may run in parallel (also considered as the smallest unit of concurrency i.e. it cannot be parallelized further). In low level design, these tasks get mapped to a Thread, Process, Server, etc.
- **Resource**: Hardware unit on which a task runs (e.g. Web Server, Database Server, CPU Core, GPU core)

- **Medium**: Mode of communication used for exchanging data/messages between tasks e.g. Profibus is a medium used for communication in the industrial domain, or Infiniband used in HPC domain, Ethernet, or PCI bus if communicating tasks sit on the same physical server.
- **Consumption**: Unit of Measurement e.g. network bandwidth is used as measurement criteria for Medium or CPU utilization is used to measure CPU performance.

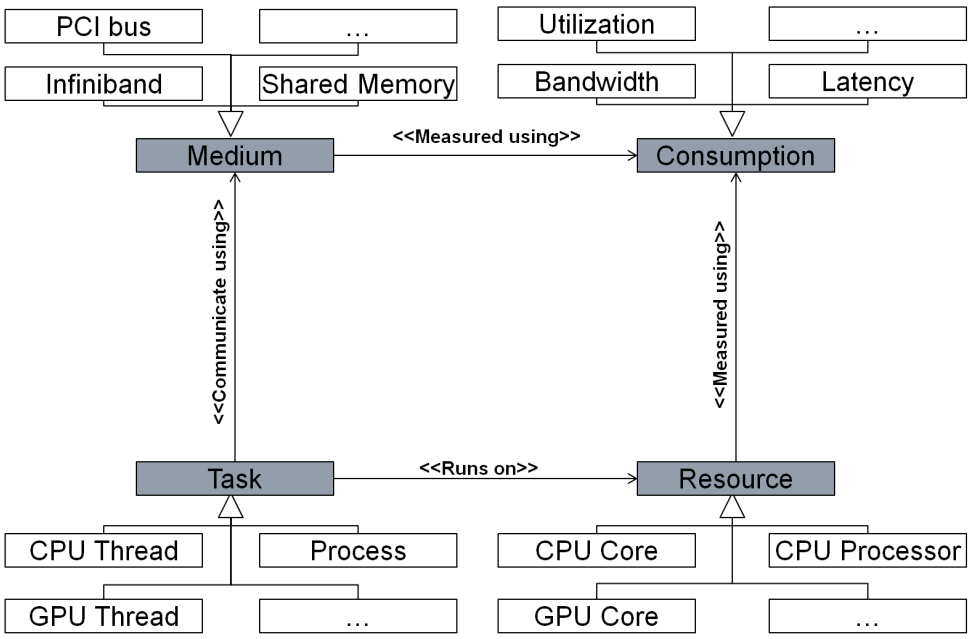

Figure 1: Parallel computing terminology used in the paper

3. CATALOG OF PERFORMANCE SMELLS FOR PARALLEL COMPUTING

This section presents our catalog of performance smells related to parallel computing. The essence of each smell is captured using the following template:

- Definition – Provides the definition for the smell.
- Signs – Lower level system signs that could indicate the presence of respective smell.
- Problem – Describes causes such as wrong assumptions and violated parallel computing principles which may cause the smell.
- Mitigation technique – Provides possible solutions that can mitigate the smell.
- Mitigation example – Examples from concurrent programming literature that use proven solutions to address the cause of the smell.

3.1 Over subscription (Is it actually running in parallel?)

Definition: This smell arises when multiple active tasks try to access limited resources at the same time.

Signs: Number of tasks ready for execution is greater than number of resources available.

Problem: Most parallel architectures allow the launching of more number of tasks than the available resources on the system. When the number of active tasks is more than the number of available resources, tasks make little or no progress. This is usually because resources have become exhausted or are too limited to perform needed operations. When this happens, a behavior that is observed is that when a task requests a resource, the operating system tries to satisfy that request by taking the needed resource

from some other task. This in turn leads to the other task making a new request for the same resource which obviously can't be satisfied.

Parallelism is possible only when there are resources available for tasks to run in parallel, else tasks get serialized and wait for resources to be available. Essentially, the overhead of creation of parallel tasks and switching between tasks dominates the overall time taken and performance gets impacted if optimal number of tasks is not chosen.

Mitigation technique: Number of optimal tasks per resource is different for different parallel architectures. So, it is necessary to adopt an appropriate task-to-resource mapping strategy for different parallel hardware.

Mitigation example:

- Thread pool is a well-known pattern that does not assign a hardcoded number of threads to a core; instead, it is based on the best mapping of resources to task/threads for a specific parallel architecture.
- CUDA makes use of occupancy calculator to get optimal number of threads per Symmetric Multiprocessor (Occupancy Calculator 2008).

3.2 Uneven Workload (Am I being fair in allocating responsibility?)

Definition: This smell arises when tasks are assigned statically to resources leading to uneven consumption of resources due to unbalanced work allocation to tasks.

Signs:

- All tasks do not have equal/similar computation job to perform.
- Resource utilization is uneven.

Problem: Tasks in some cases are statically assigned to the resources due to application or parallel architecture restrictions. Different tasks take different paths during the course of execution and are dependent on the provided input data. Some tasks may end up performing more computation than other tasks.

A sample code shown below depicts one such scenario. In most parallel programming frameworks, each task is identified by a unique id (task ID) and developers use this identifier to assign different computation which can be executed in parallel to different tasks. In the scenario below, the task with taskID= 0 ends up performing more computation than the other task.

Pseudo Code

```
if(task ID == 0 )
  //Do more work
else
  //Do less work
```

Mitigation technique:

- Tasks can be scheduled dynamically on resources using migration techniques based on resource consumption.

Mitigation example:

- “Work Stealing” algorithm is used in many frameworks, where if one task has no or little work, it steals work from other tasks. This makes the workload distribution even across resources.
- Virtual Machine migration is used in cloud environment to optimally utilize the compute resources and perform load balancing (Leelipushpam et al. 2013).

3.3 Unused Resources (Am I using everything I have?)

Definition: This smell arises when some resources are unconsumed even though the application will benefit from utilizing them.

Signs: Number of tasks is lesser than the number of resources available.

Problem: Parallel computing architectures are becoming more heterogeneous in nature. Multiple types of resources are available on the same chip/die. For example, integrated GPU is a part of many of the latest chips and is used only for graphics; however, it can also be utilized for doing general purpose computation. Figure 2 shows another scenario where a CPU chip has various levels of resources available and different parallel programming models may be needed in combination to utilize them to avoid keeping the resources idle. The architect/developer is generally not aware of different types of parallel resources available on the system, and hence may not utilize all of them.

Mitigation technique: Create and launch more tasks based on resources available.

Mitigation example: A combination of programming models like OpenMP (Sato 2002) along with AVX (Weiss 1989) and CUDA (Garland 2010) can utilize CPU cores and CPU vector units along with GPU.

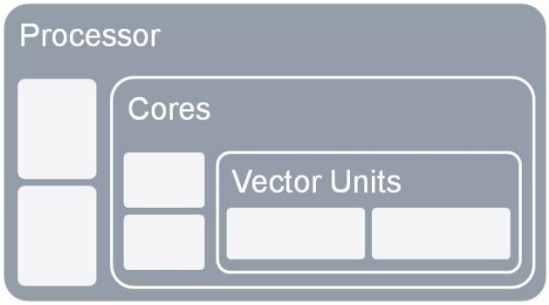

Figure 2: CPU chip (multi-processor) resource hierarchy

3.4 Short Lifetime (Lack of recycle and reuse)

Definition: This smell arises when there is less/no reuse of existing memory.

Signs: Creation and deletion of memory is frequent.

Problem: It is difficult to have advance knowledge of how much memory is required per task and hence each task is responsible for its own memory allocation and deletion. Creation and deletion of memory is a costly operation and has more pronounced effect in parallel architectures since each task may end up creating and destroying memory separately.

Further, in order to avoid conditions like deadlock and data-races, it is recommended that each task works on independent memory. Hence, architects prefer not to reuse memory across tasks leading to this smell.

Mitigation technique: Avoid premature destruction/release of memory.

Mitigation example: Flyweight (Gamma et al. 1995) and Memory Pool (Kircher and Jain 2004) patterns can be used to address the cause of this smell.

3.5 Extra Synchronization (Why are tasks waiting for each other?)

Definition: This smell arises when tasks depend frequently on outputs of other tasks that are running in parallel, leading to unnecessary synchronization overhead.

Signs: Resource consumption of a task shows repeated cycles of high utilization followed by no activity. This is a possible indicator that the task is waiting for other tasks to provide some data or is accessing shared data being used by other tasks.

Problem: When tasks share data structures, access to shared variables must be guarded with locks. This leads to a dependency among tasks and results in poor scalability because tasks keep waiting to get access to shared memory.

Mitigation technique:

- Reduce dependency between tasks.
- Delay synchronizing as much as possible.
- Use asynchronous protocols.
- Create a pool of tasks and increase their lifetime.

Mitigation example: Using Lock-Free Containers (Herlihy et al. 2003) reduces the amount of synchronization needed for shared data. This solution has shown promising results and a lot of research is currently being conducted in this area.

3.6 Unfit Parallelism (Why use a hammer to kill a fly?)

Definition: This smell arises when a parallel architecture that is used for an application is unsuitable or non-optimal for that application context.

Signs: The parallel software design violates the principle of the underlying parallel hardware architecture.

Problem: Each parallel framework comes with advantages and limitations. This situation may arise if a parallel framework is used without knowing its limitations.

Mitigation technique:

- Find an alternative design that suits the parallel architecture being used.
- Find an alternative parallel architecture that suits the design being developed.
- Understand the advantages and bottlenecks of the parallel architecture and carefully analyze its fitment to the current needs of the application.

Mitigation example:

- CUDA parallel architecture is good for performing embarrassingly parallel jobs but has limited support for other kinds of parallelism like task parallelism. In such a context, using parallel architectures like multi-core CPU for task parallelism would be a better option.
- Iterative algorithms e.g. iterative solvers which are dependent on results of previous iteration cannot be made parallel unless the algorithm is modified to reduce this dependency.

3.7 Low Locality (Why is it so far?)

Definition: This smell arises when data needed by tasks are not local and a large portion of the task execution time is unnecessarily spent in bringing the data to the tasks.

Signs:

- Tasks are executing far from the data they need.
- Increased medium consumption.

Problem: Parallel architectures have different sets of memory. The architect/developer is generally not aware of this and chooses the default location of storage which may not give good performance.

Some parallel architectures such as discrete GPU have their own memory hierarchy. In these architectures, it is required to transfer the data to GPU memory before the task running on the GPU can consume the data. Therefore, such architectures may not be a suitable candidate for applications where data transfer itself consumes more time than the parallel computation.

Figure 3 shows a scenario where task assignment to resource determines the time it takes to fetch the data and perform computation. Task1 gets assigned to Resource1 and access to Object1 will be faster as compared to Object2.

Mitigation technique:

- Tasks should be scheduled near the data they need.
- Frequently used data should be stored locally.

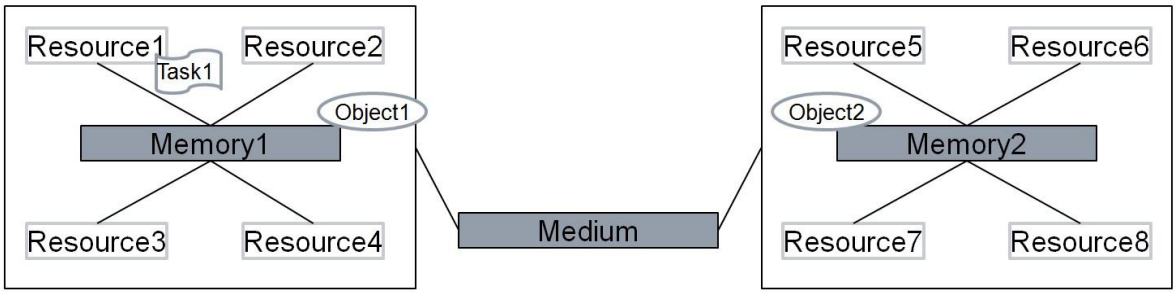

Figure 3: Resource and Memory allocation across

Mitigation example:

- Caching Pattern (Kircher and Jain 2004)
- Big-Data platforms like Hadoop (White 2012) bring the computation near the data rather than bringing data near the computation.

4. LINKED SMELLS

Performance smells are generally related to each other and hence in this section we have attempted to establish relationship between them. Figure 4 shows the consolidated view of linked smells.

4.1 Co-occurring Smells

Definition: Smells that tend to occur together or in other words co-exist, i.e. the presence of one smell indicates a high probability of occurrence of the other

- Uneven Workload <-> Unused Resources: Uneven workload usually results in the under-utilization of some of the resources and hence has a high probability of occurring together with Unused Resources.
- Short Lifetime <-> Extra Synchronization: Short Lifetime results in Fork-Join Pattern where tasks may get created and destroyed repeatedly during the course of application execution which results in additional implicit synchronization.

4.2 Mutually-exclusive Smells

Definition: Smells that contradict each other and will never exist at the same time.

- Over Subscription <-> Unused Resources: Over Subscription occurs as a result of more number of tasks and limited number of resources; hence, it is highly unlikely that these two smells occur together.

4.3 Tradeoff Smells

Definition: Smells that might occur when we try to mitigate the smell in focus.

- Extra Synchronization <-> Unused Resources: Using more tasks to mitigate Unused Resources can result in more communication and consequently synchronization if tasks are not self sufficient and are dependent on other tasks. This will bring down the overall performance. So increasing number of tasks should be done cautiously.

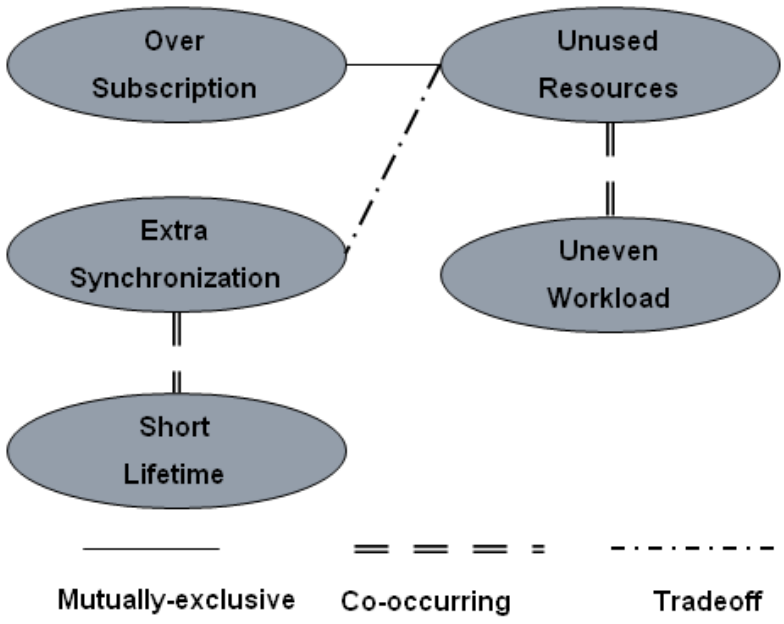

Figure 4: Linked smells

5. CONCLUSION AND FUTURE WORK

This paper presented a catalog of language-agnostic performance smells related to parallel computing. Each smell in the catalog was described in the context of a parallel architecture along with the suggested mitigation techniques to address that smell and its relationship with other smells in the catalog. In the future, we plan to refine and extend the catalog based on latest advancement in field of parallel and distributed computing.

6. ACKNOWLEDGEMENTS

We thank our shepherd, Alfredo Goldman, for his careful and insightful comments that have significantly helped to improve this paper

REFERENCES

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D: Refactoring: Improving the De-sign of Existing Code. Addison-Wesley Professional. 1999.

Ganesh, S., Sharma, T., Suryanarayana, G.: Towards a Principle-based Classification of Structural Design Smells. In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–37.

Connie U. Smith and Lloyd G. Williams. 2000. Software performance antipatterns. In Proceedings of the 2nd international workshop on Software and performance (WOSP '00).

Smith, Connie U. and Williams, Lloyd G.: More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In Proceedings of International CMG Conference 2003: 717-725

Hallal, H., Alikacem, E., Tunney, W., Boroday, S., Petrenko, A.: Antipattern-based Detection of Deficiencies in Java Multithreaded Software. In proceedings of the Fourth Interna-tional Conference on Quality Software (QSIC'04)

Bradbury, J., Jalbert, K.: Defining a Catalog of Programming Anti-Patterns for Concurrent Java. In Proceedings of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09), pages 6-11, Orlando, Florida, USA, Oct. 2009.

Stephen Toub. Microsoft: Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4. Retrieved from http://www.microsoft.com/en-in/download/details.aspx?id=19222, July 2010

Garland, M., "Parallel computing with CUDA," in Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on , vol., no., pp.1-1, 19-23 April 2010

Diamond, J.; Burtscher, M.; McCalpin, J.D.; Byoung-Do Kim; Keckler, S.W.; Browne, J.C., "Evaluation and optimization of multicore performance bottlenecks in supercomputing applications," in Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on , vol., no., pp.32-43, 10-12 April 2011

Colin Campbell , Ade Miller, A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures, Microsoft Press, 2011

Occupancy Calculator, 2008. http://developer.download.nvidia. com/compute/cuda/CUDA_Occupancy_calculator.xls. Daga Bibliography 79 - CUDA

Gamma, E., Helm. R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Kircher, M., Jain, P.: Pattern-Oriented Software Architecture Volume 3: Patterns for Re-source Management. Wiley. Volume 3 edition (2004).

Herlihy, M., Luchangco, V., Moir, M.: Obstruction-Free Synchronization: Double-Ended Queues as an Example. In Proceedings of 23rd International Conference on Distributed Computing Systems. 2003. pp. 522

White, T.: Hadoop: The Definitive Guide. O'Reilly. 2012.

N. Bell and J. Hoberock. Thrust: A productivity-oriented library for CUDA. In W. W. Hwu, editor, GPU Computing Gems, volume 2, chapter 4, pages 359–372. Morgan Kaufmann, Oct. 2011.

Wright, G.A., "Magnetic resonance imaging," in Signal Processing Magazine, IEEE , vol.14, no.1, pp.56-66, Jan 1997

Kato, H., "Computed Radiography," in Image Management and Communication in Patient Care, 1989. Implementation and Impact., First International Conference on , vol., no., pp.108-116, 4-8 Jun 1989

Sato, M., "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors," in System Synthesis, 2002. 15th International Symposium on , vol., no., pp.109-111, 2-4 Oct. 2002

Weiss, M., "Parallel languages, vectorization, and compilers," in Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International , vol., no., pp.114-115, 20-22 Sep 1989

Leelipushpam, P.G.J.; Sharmila, J., "Live VM migration techniques in cloud environment — A survey," in Information & Communication Technologies (ICT), 2013 IEEE Conference on , vol., no., pp.408-413, 11-12 April 2013