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, Anti-Patterns
Additional Key Words and Phrases: Multi-Core, Many-Core

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 [19], CT [20] 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 in order 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 apply suitable design 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 a (possibly suboptimal) design decision for an underlying parallel architecture that negatively impacts the performance of an application.

It should be noted that it is possible that a particular technique when applied to a certain parallel architecture may improve the performance but may act as a smell and degrade the performance when applied to a different parallel architecture. For example, in CUDA [8] architecture for Nvidia GPU, in order to hide latency it is recommended to launch more threads than the number of available cores; same optimization when applied on multi-core CPU architecture may result in performance degradation because more time 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 in order 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 tuning strategy.

This work is supported by Research and Technology Center, Siemens Technology and Services Pvt. Ltd., India
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, a relevant example, and the associated mitigation techniques to address that smell. It also documents the relationship between smells. Each smell is generally related to other smell and hence we have attempted to establish relationship between them 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 catalogs of 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 [1][2]. We extend this term to the topic of performance in the context of parallel computing, and define a performance smell as “a (possibly sub-optimal) design decision for an underlying parallel architecture that negatively impacts the performance of an 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 [3][4]. 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 [5]. Similarly, Bradbury and Jalbert present a catalog of programming anti-patterns for concurrent Java [6]. In a similar fashion, there exists a catalog of anti-patterns which is focused only on the .NET framework [7]. 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 [16] is emerging where different parallel architectures are present on the same die/chip. Due to this, application designers are moving to higher level frameworks [17][18] 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 the choice of a poor design solution that affects performance. There are accounts in the literature that present optimal solutions for specific parallel architectures such as CUDA GPU [8] and multi-core [9]. However, these do not provide a generic view of performance smells across parallel architectures.

In order to provide a generic view of performance smells, it becomes necessary to provide abstractions for terminology used across parallel architectures. Diagram 1 shows these abstractions and also the relationships between them. These abstract terms that will be used in rest of the paper are as follows:

- **Task**: 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.
3. CATALOG OF PERFORMANCE SMELLS FOR PARALLEL COMPUTING

In this section, we present our catalog of performance smells related to parallel computing. The essence of each smell is captured using the following template:

- **Context**: Describes the context that caused the smell to occur.
- **Problem**: Describes causes such as wrong assumptions and violated parallel computing principles which may cause the smell.
- **Definition**: Provides the definition for the smell under the mentioned context.
- **Signs**: Lower level system signs that could indicate the presence of respective smell.
- **Mitigation technique**: Provides possible solutions that can mitigate the smell.
- **Mitigation example**: Examples from the existing proven concurrent programming literature that addresses the smell.

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

**Context**: Multiple active tasks try to access limited resources at the same time.

**Problem**: Most parallel architectures allow launching 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, usually because resources have become exhausted or are too limited to perform needed operations. When this happens, a pattern typically develops in which upon a request by a task, the operating system tries to find resources by taking them from some other task, which in turn makes new requests of resources that 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.
**Definition:** This smell arises when resources are overloaded with tasks.

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

**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 one of the well known patterns which is based on the concept of best mapping of resources to task/threads.
- CUDA makes use of occupancy calculator to get optimal number of threads per Symmetric Multiprocessor. [11]

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

**Context:** Tasks with different workloads utilize the resources unevenly resulting in under-utilization of some resources.

**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 input data provided. Some tasks may end up performing more work while some may have lesser responsibility. A sample code shown below will result in this scenario:

**Pseudo Code**

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

Also new generation architectures are not homogeneous in nature i.e. each resource has a different processing power. The architect/developer may not be aware of this and may end up underutilizing resources that have more computing power.

**Definition:** This smell arises when tasks are assigned statically to resources and consume the resources unevenly due to unbalanced work allocation to tasks.

**Signs:**
- All tasks do not have equal responsibility.
- Resource utilization is uneven.

**Mitigation technique:**
- Allocation of work to tasks should be dependent on resource’s computing power.
- 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 for resources.

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

**Context:** Application does not consume all the resources available even though the application will benefit from using them.
**Problem:** Architect/developer is generally not aware of different types of parallel resources available on the system and hence may not utilize all the parallel resources.

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. Diagram 2 shows another scenario where a CPU chip has various levels of resource available and different parallel programming model may be needed in combination to utilize them else they remain idle.

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

**Signs:** Number of tasks lesser than number of resources available.

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

**Mitigation example:** A combination of programming models like OpenMP [21] along with AVX [22] and CUDA [8] can utilize CPU cores and CPU vector units along with GPU.

Diagram 2: CPU chip resource hierarchy

3.4 Short Lifetime (Lack of recycle and reuse)

**Context:** Tasks allocate and free the memory frequently.

**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/deletion. Creation/Deletion of memory is a costly operation and has more pronounced effect in parallel architectures since each task may end up creating/destroying memory separately.

Also in order to avoid conditions like deadlock/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.

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

**Signs:** Creation/Deletion of memory is frequent.

**Mitigation technique:** Avoid premature destruction/release of memory.

**Mitigation example:** Flyweight pattern [12] and Memory Pool [13] can be used to resolve this smell.

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

**Context:** Tasks may cooperate in order to accomplish the required computation and spend time waiting for each other rather than executing in parallel.
**Problem:** Shared data structures among tasks demand guarding access to these variables with locks which creates dependency among tasks. Parallel design in this case usually does not give scalable performance because tasks are waiting to get access to shared memory.

**Definition:** This smell arises when tasks depend frequently on outputs of other tasks running in parallel leading to additional overhead of synchronization.

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

**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:** Lock-Free Containers [14] reduces the amount of synchronization needed for shared data. It 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?)

**Context:** Wrong choice of parallel framework results into performance degradation.

**Problem:** Each parallel framework comes with advantages and limitations. If a parallel framework is used without knowing its limitation this situation may arise.

**Definition:** This smell arises when a wrong parallel framework is used for parallelism.

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

**Mitigation technique:**
- Find alternative design that suits the framework.
- Find alternative framework that suits the design.
- Understand the advantages/bottlenecks of the framework.

**Mitigation example:**
- CUDA parallel framework is good for performing embarrassingly parallel jobs but has limited support for other kinds of parallelism like task parallelism.
- Iterative algorithms e.g. iterative solvers which are dependent on results of previous iteration cannot be made parallel unless the algorithm is modified for reducing the dependency.

3.7 Low Locality (Why is it so far?)

**Context:** Task spends a lot of unnecessary time in fetching the data needed for computation.

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

Some parallel architectures e.g. discrete GPU have their own memory hierarchy as compared to CPU. It is required to transfer the data to GPU memory before the task running on GPU can consume them. GPU may not be a suitable candidate for application where data transfer to GPU itself is time intensive as compared to performing parallel computation.
Diagram 3 shows another such 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.

**Definition:** This smell arises when data needed by tasks are not local and large portion of 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.

**Mitigation technique:**
- Tasks should be scheduled near the data they need.
- Frequently used data should be stored locally.

**Mitigation example:**
- Caching Pattern [13]
- Big-Data platforms like Hadoop [15] bring the computation near to where data exist rather than bringing data to 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. Diagram 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.

![Diagram 4: Linked smells](image)

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 introduce our catalog to different teams and projects within the organization so that we can continue to refine and extend the catalog based on the feedback received from different stakeholders within the organization.
REFERENCES

17. Thrust Library: https://code.google.com/p/thrust/
19. MRI: https://en.wikipedia.org/wiki/Magnetic_resonance_imaging

Appendix

(Performance Smells to be discussed further)

Over-Weight Task (Why the extra luggage?)

Context: Choice of task used for parallelism results in additional overhead.

Problem: Overhead of creation of tasks varies across different parallel frameworks, for e.g. MPI creates a process whereas OpenMP creates threads (light-weight process). Architect usually has a choice to decide upon one such framework based on application needs. Lack of awareness of the overheads of different frameworks for tasks can lead to unnecessary costs e.g. higher memory footprint, context switching overhead.

Definition: This smell arises when adding tasks to the application makes it bulky.

Signs: Task management (creation/deletion/scheduling) time is significant in the application.

Mitigation technique:
- Keep a pool of tasks and reuse them.
- See if lightweight tasks can be used instead.
**Mitigation example:** Context switching between tasks in CUDA framework is faster as compared to switching tasks in other frameworks like OpenMP as CUDA threads are lightweight.