Towards an Overhead Estimation Model for Multithreaded Parallel Programs

Virginia Niculescu[a], Camelia Şerban[b] and Andreea Vescan[c]
Babeş-Bolyai University, Faculty of Mathematics and Computer Science, Computer Science Department, Cluj-Napoca, Romania

Keywords: Parallel Programming, Metrics, Overhead, Multithreading, Synchronization, Estimation, Validation.

Abstract: The main purpose of using parallel computation is to reduce the execution time. To reach this goal, reducing the overhead time induced by the additional operations that parallelism implicitly imposes, becomes a necessity. In this respect, the paper proposes a new model that evaluates the overhead introduced into parallel multithreaded programs that follows SPMD (Single Program Multiple Data) model. The model is based on a metric that is evaluated using the source code analysis. Java programs were considered for this proposal, but the metric could be easily adapted for any multithreading supporting imperative language. The metric is defined as a combination of several atomic metrics considering various synchronisation mechanisms. A theoretical validation of this metric is presented, together with an empirical evaluation of several use cases. Additionally, we propose an AI based strategy to refine the evaluation of the metric by obtaining accurate approximation for the weights that are used in combining the considered atomic metrics.

1 INTRODUCTION

A general expectation from the parallel programming is that by doubling the hardware resources, one can reasonably expect a program to run twice as fast. However, in typical parallel programs, this is rarely the case, due to many overheads associated with parallelism. An accurate quantification of these overheads is critical for understanding and improving the parallel programs performance. The parallel computation is also limited by overhead costs such as: threads/processes start, management and termination costs, synchronization problems – task coordination, cost of communication among multiple tasks (threads/processes), the overhead of some software libraries, parallel compilers or interpreters and supportive OS.

SPMD (Single Program, Multiple Data) programming model (Grama et al., 2003) is characterized by the fact that all the processes/threads execute a copy of the same program on different data. SPMD is widely used in parallel programming, due to the ease of designing a program that consists of a single code running on all processing elements. The differentiation between the execution on each processing element (thread/process) can be done based on the ID of each process/thread.

The overhead time (T_O) is estimated, in general, as being the difference between the product of the parallel time with the number of processing elements, and the sequential time:

$$T_O = p * T_p - T_s \quad (1)$$

This is a very general evaluation and it is difficult to be used in the design stage of the development. For a certain class of parallel programs – as SPMD on shared memory platforms – written in a particular language (as Java) – where we know the specific synchronization mechanisms that could be used, we may estimate more accurately the overhead time. We propose in this paper such a model for SPMD multithreaded Java programs, based on a synchronization overhead metric. Java programs were considered for this model, but it could be easily adapted for any multithreading supporting imperative language.

Thus, the contribution of the paper is three-fold: (1) a new model for overhead evaluation, based on a metric defined as an aggregation of several atomic metrics related to various synchronization mechanisms; (2) the theoretic validation of the proposed

[a] https://orcid.org/0000-0002-9981-0139
[b] https://orcid.org/0000-0002-5741-2597
[c] https://orcid.org/0000-0002-9049-5726

Niculescu, V., Şerban, C. and Vescan, A.
Towards an Overhead Estimation Model for Multithreaded Parallel Programs.
DOI: 10.5220/0011083400003176
In *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2022)*, pages 502-509
ISBN: 978-989-758-568-5; ISSN: 2184-4895

metric using Weyuker's metric properties (Misra and Akman, 2008), (EJ, 1988), and (3) an empirical evaluation of the weights used in the metric definition, based on concrete experiments. In addition, an AI strategy to refine the metric by obtaining a more accurate approximation for its weights is proposed.

Regarding the usability of the proposed metric we could mention the possibility to estimate the overhead time at the design stage, compare different design solutions for a given problem, or estimate the need for refactorization.

The paper is structured as follows. Section 2 discusses related work regarding theoretical validation of software metrics and tools for performance assessment in concurrent programs. Section 3 describes the main synchronization mechanisms in Java, while the definition of the new metric is given in Section 4, together with its theoretical and empirical analysis. The conclusions and further work are emphasized in the last section.

2 RELATED WORK

This section describes existing approaches regarding theoretical and empirical validation of software metrics and existing tools for multithreaded programs performance evaluation.

Evaluation of Software Metrics - Various Criteria. Measures and way of measuring in Software Engineering (SE) domain are different from that of other engineering domains. In comparison with measuring the length of an engineering product, measuring the size of a software, such as program length, is not so easy and there are few formal approaches that rigorously apply software measures. Formal approaches in SE are usually based on different types of metrics that are used in order to quantify those aspects that are considered important for the assessment. Many researchers (EJ, 1988), (H, 1992), (B and N, 1995), (Briand LC and S, 1995) contributed to lay a foundation for measuring software, both for metric definition and metrics validation. There are basically two categories of techniques for metrics' validation: *the empirical validation* which confirms the metrics actual applicability, and *the analytical evaluation* defined based on definite measurement theories. Prior to the empirical validation (i.e., applying to industry), every metric has to be analytically evaluated to confirm that it has scientific pedestal, and it was defined based on definite measurement theories.

Tools for Performance Evaluation. Software performance is in many cases increased by implying parallel and concurrent computation. In order to make effective the development of such programs, the developers have come to expect tools that are able to augment the design and execution infrastructure with different capabilities such as design assistance, performance evaluation, debugging or execution control. There is quite a large set of tools like these, and further on we will mention just a few that were proposed for multithreaded programming, and which are related to the overhead time.

Tmon, a tool for monitoring, analyzing and tuning the performance of multithreaded programs was proposed (Ji et al., 1998). The tool relies on two measures performed at the run-time of the program; it uses thread waiting time and constructs thread waiting graphs. A static analysis tool called *Iceberg* (Shah and Guyer, 2016) was proposed in order to identify performance bugs in concurrent Java programs. They developed an analysis called latency variability analysis, a flow-sensitive analysis trying to estimate the range of possible latencies through a block of code. Iceberg tool (Shah and Guyer, 2018) was improved in order to perform a dynamic analysis of Java programs. A tool that automatically detects the inefficiency intervals representing time periods when a concurrent application is not using all its capabilities of the parallel system was proposed in (Espinosa et al., 1998). Another lock profiler designed in the context of a new metric, critical section pressure, is proposed in paper (David et al., 2014). There are several applications serving as testing environment for this tool, the results showing that it can detect phases where a lock hinders the threads process.

In relation to the existing approaches our proposed metric can be used early in the development lifecycle, when the system design is known, to predict the overhead added by the parallelism. In this way, possible errors and crash that could appear during the life of the program can be prevented. As far as we know there are no references in literature regarding such a metric.

3 SYNCHRONIZATION MECHANISMS

In order to evaluate the overhead brought through synchronization we will consider the following synchronization mechanisms:

- critical sections
  - using `Locks`
  - using `synchronized` methods and blocks
- conditional waiting
  - using `wait`/`notify` calls

  - using `Condition` variables and the corresponding `await/signal` calls
- barriers - using `CyclicBarrier`
- 'rendez-vous' mechanisms using `Exchanger`

A *lock* is a synchronization mechanism for enforcing the access limitation to a resource in an execution environment where there are many threads of execution. A lock is designed to enforce a mutual exclusion concurrency control policy (Garg, 2004; Raynal, 2013). In Java, `Lock` class provides implementation for this mechanism (Göetz et al., 2006).

Synchronized methods and blocks in Java are implemented based on the association of each object to a monitor (Göetz et al., 2006), and they are mechanisms for defining critical sections. A monitor encapsulates: shared data structures, procedures that operate on the shared data structures, and synchronization between concurrent procedure invocations. In case of Java we may consider that the encapsulated data are the object attributes, and by defining `synchronized` methods we may define the procedures (methods) that should be called based on mutual exclusion (Göetz et al., 2006; Garg, 2004; Raynal, 2013). Monitors could also include conditional synchronization; Java provides only one condition queue per monitor using the corresponding `wait/notify` methods.

Synchronization conditions (also known as condition queues or condition variables) provide a means for one thread to suspend execution (to "wait") until notified by another thread when some state condition arrives to be true. The correspondent Java implementation is the class `Condition`, and its instances should always be associated with a lock (Göetz et al., 2006).

A *barrier* is a synchronization method that forces a group of threads or processes to stop at the point of the barrier, and not proceed until all other threads/processes reach this barrier. A Java `CyclicBarrier` object is an object that implements a barrier, which can be used (reused) cyclically - in multiple points of the program execution. The method `await` is used to specify the synchronization points (Göetz et al., 2006; Garg, 2004; Raynal, 2013).

The Java `Exchanger`(Göetz et al., 2006) class implements the "rendez-vous" concept, which specifies a two-way synchronization and communication between two threads or processes: if one thread arrives to the rendez-vous point it waits for the other to arrive and then they exchange information (Garg, 2004; Raynal, 2013).

There are also many other synchronization mechanisms and an example worth to be mentioned is the semaphore. A *semaphore* (Garg, 2004; Raynal, 2013) is a synchronization construct that is typically used to coordinate the access of multiple threads/processes to resources. It also has a Java implementation – class *Semaphore* (Göetz et al., 2006). We don't consider them in this paper because usually they are not used in SPMD type of programs. More details about other synchronization mechanisms could be found in (Göetz et al., 2006; Garg, 2004; Raynal, 2013).

As noticed, critical sections could be obtained through multiple mechanisms: - e.g. locks, synchronized blocks.
Do their choice influence the program performance? Theoretically, we may assume that the answer is affirmative, and also from some simple experiments we noticed that there are some differences. In general, solving a problem could have different solutions based on different synchronisation mechanisms.

4 OVERHEAD METRIC

We propose a model to approximate the overhead of SPMD Java programs by using a metric defined based on the different characteristics of the synchronization mechanisms.

4.1 Java SPMD Programs

In order to evaluate the overhead brought by the synchronization we will consider the synchronization mechanisms mentioned in the previous section.

We consider SPMD programs formed of three parts: threads creation, starting all the threads using the same code/program (in Java the function `run()`), and joining all threads.

Combining two such programs P1 and P2 into a new program $P3 = P1 + P2$ (here we consider $+$ the combining operator) means that the P3 program is obtained by sequentially combing the two running functions $run_{P3} = run_{P1}\,;\, run_{P2}$.

In addition, we also consider that into a program we don't have adjacent critical sections – if they are then they will be implicitly merged into one.

4.2 Metric Definition

The proposed overhead metric is an aggregated metric, which is based on atomic metrics that corresponds to the synchronization mechanisms described in Section 4.1.

The overhead time T_O can be estimated based on a synchronization metric O by using the following definitions:

$$O:\mathbb{P}\to\mathbb{R},\quad T_O:\mathbb{P}\times\mathbb{S}\to\mathbb{R},$$
$$\mathbb{P}\text{ is the set of all SPMD Java programs and}$$
$$\mathbb{S}\text{ is the set of all execution systems}$$
$$T_O(P,S)=t_S*\left[w_{th\_manag}+\ O(P)\right],P\in\mathbb{P},S\in\mathbb{S} \tag{2}$$

where

- $P=(P_1,P_2,...,P_p)$, $P_i:\ 0\leq i<p$ represents the finite set of threads defined inside a SPMD program;
- t_S represents a time that depends on the performance of the execution system, the number of processors of the system (w_{proc}), and on the threads per cores ratio (w_{load});
- w_{th_manag} = the weight for managing the threads.

We have

$$\begin{array}{lllll} O(P) & = & & & \\ w_{bar} & *\ p\ *\ (\#bar) & * & f(w_{load},dis) & + \\ w_{wait} & *\ (\ \#wait) & / & f(w_{load},dis) & + \\ w_{cond} & *\ (\ \#await) & / & f(w_{load},dis) & + \\ w_{ex} & *\ (\ \#ex) & / & f(w_{load},dis) & + \\ w_{sync} & *\ p*\left(\sum_{i=1}^{\#sync} cs_i\right) & / & f(w_{load},dis) & + \\ w_{lock} & *\ p*\left(\sum_{i=1}^{\#lock} cs_i\right) & / & f(w_{load},dis) & + \end{array} \tag{3}$$

where:

- $\#bar$ and w_{bar} – the number of synchronization barriers based on the calls of `await` through `CyclicBarrier`s, and the corresponding weight;
- $\#wait$ and w_{wait} – the total number of calls of `wait` method of `Object`, and the corresponding weight;
- $\#await$ and w_{cond} – the total number of `await` operations called through a `Condition` variable, and the corresponding weight;
- $\#ex$ and w_{ex} – the total number of `exchange` operations executed through an `Exchanger`, and the corresponding weight;
- $\#sync$ and w_{sync} – the number of critical sections specified with `synchronized` blocks or methods, and the corresponding weight;
- $\#lock$ and w_{lock} – the number of critical sections specified with `Lock` objects and `lock()`, and `unlock()` methods, and the corresponding weight;
- cs_i the size of a critical section ($\#i$) – this is expressed in number of statements;
- dis is a measure of the dissimilarity between the threads estimated as:

$$dis=\frac{\#conditional\_statements}{\#total\_statements}$$

- $f(w_{load},dis)=\sigma*w_{load}*(1+dis)$ is a calibration function; a barrier is delayed when the w_{load} and dis are increasing, but all the other synchronization mechanisms are favored in this case.

In general, a high level of loading (w_{load}) increases the execution time; for example for the barrier execution if the loading is low the probability that threads arrive in the same time at the barrier point is very high, and vice-versa if the loading is high is very probable the threads arrive at the barrier point at different times. Also, if the dissimilarity between the threads is higher then the probability of threads arriving at the barrier at different moments in time is also higher. This is why in the overhead metric formula the time due to barriers should be multiplied by the calibration function.

For the critical sections we have the opposite situation: it is desirable that the threads arrive to critical sections at different times, and high level of loading determine this implicitly.

When we use `Condition` and `wait-notify` for conditional synchronization, we need to use them inside the critical sections; the overhead due to these critical sections is included into the corresponding weights w_{wait} and w_{cond}.

4.3 Theoretical Validation of the Metric

Among numerous metric validation criteria that exist in the literature, as we have mentioned them in the Section 2, Weyuker's properties are most extensively used for evaluating software metrics; they are guiding tools for identifying good and complete metrics (EJ, 1988).

Property 1: Non-coarseness. $(\exists)\ (P1),\ (P2)$ two distinct programs from $\mathbb{P}$ such that $O(P1)\neq O(P2)$.

This is trivially fulfilled by O metric due to the fact that is not defined as a constant map.

Property 2: Granularity. It states that there will be a finite number of cases for which the metric value will be the same.

It is considered that this property is met by any metric measured at the program level, because the universe deals with at most a finite set of programs and so the set of those programs having the same value for the proposed metric is also finite (Chidamber and Kemerer, 1994).

Property 3: Non-Uniqueness (Notion of Equivalence). $(\exists)\ (P1),\ (P2)$ two distinct programs from $\mathbb{P}$ such that $O(P1)=O(P2)$.

We may consider two programs that each update a variable (based on some formulas) inside a critical section; for the first program we may use with `synchronized` and for the second `Lock`. The values for w_{sync} and w_{lock} could be different but by chosen the sizes m and n of the two critical sections correspondingly, we arrive to the same value for the metric. It is possible to find n and m such that

$m * w_{sync} = n * w_{lock}, \left(\frac{n}{m} = \frac{w_{sync}}{w_{lock}}\right)$.

Property 4: Design Details are Important. This property states that, in determining the metric for an artifact, its design details also matters. When we consider the designs of two programs $P1$ and $P2$, which are the same in functionality, does not imply that $O(P1) = O(P2)$.

In our case, we have different means to achieve similar things; for example we have `synchronized` and `locks` for critical sections, and `Object:wait-notify` and `Condition:await-signal` for conditional synchronization.

Property 5: Monotonicity.

It states that a component of a program is always simpler than the whole program. For all $(P1), (P2) \in \mathbb{P}$ either $O(P1) \leq O(P1+P2)$ or $O(P2) \leq O(P1+P2)$ should hold.

Our metric is defined as a summation of atomic metrics and based on the combing operation we have $O(P1+P2)$ is bigger or equal than $O(P1)$ or $O(P2)$.

Property 6: Non-equivalence of Interaction. If $P1$, $P2$ and $P3$ are three programs having the property that $O(P1) = O(P2)$ does not imply that $O(P1+P3) = O(P2+P3)$. This suggests that the interaction between $P1$ and $P3$ may differ from that of $P2$ and $P3$.

If the P1 program ends with a critical section, and P2 doesn't end with a critical section, while P3 starts with a critical section, then for $P1+P3$ the two corresponding critical sections will be implicitly merged, and this reduces the overhead. Then $O(P1+P3)$¡ $O(P2+P3)$.

Property 7: Permutation. It states that permutation of elements within the program being measured may change the metric value. Being given a program $P1$ that is transformed into a program $P2$ by permuting the order of the statements such that the provided functionalities are preserved, the property states that $O(P1) \neq O(P2)$.

Considering a parallel program $P1$ that defines a critical section (defined through `synchronized`) that contains n statements, but not all the n statements need mutual exclusion, we can transformed it to obtain $P2$ by moving one of these statements out of the critical section; this way the functionality is preserved and the overhead metric is reduced because the size of the critical section is reduced. For the initial variant we have $O(P1) = w_{sync} * n$ and for the second we have $O(P2) = w_{sync} * (n-1)$.

Property 8: Renaming Property. When the name of the measured artifact changes, the metric should not change. That is, if program $P2$ is obtained by renaming program $P1$ then $O(P1) = O(P2)$.

As the proposed metric is measured at the program level and, it does not depend on the name of the program nor on the name of its classes, methods and instance variables (through which the synchronization mechanisms are implemented) it also satisfies this property.

Property 9: Interaction Increases Complexity. It states that when two programs are combined, interaction between them can increase the metric value. When two programs $P1$ and $P2$ are considered $O(P1) + O(P2) \leq O(P1+P2)$.

In general for the proposed metric we have that $O(P1) + O(P1) = O(P1+P2)$ (based on the combining operator – section 4.1, and the metric definition based on summation).

Hence, the proposed metric is effective and can be utilized for the purpose of overhead estimation in SPMD programs.

4.4 Empirical Weights Evaluation

Empirical validation is extremely significant for the accomplishment of any software measurement project (Basili et al., 1996), (Fenton and Pfleeger, 1997). Metrics proposal do not have much value without demonstrating its practical utilization. Thus, studying the applicability and usefulness of the proposed metric which has been already validated theoretically is very important.

A possible approach that can be followed for evaluation is provided in the book of Yin (Yin, 2008), which emphasizes when it is possible to rely on case studies method and how to do the research design with this approach.

In order to verify and emphasize how the metric could be used we have considered a use-case that could be solved using several variants, each of these using different synchronization mechanisms.

The considered problem is so called *reduce* operation: for a given list of elements and an associative operator defined on the type of the elements we have:

$$reduce(\oplus)[e_0, e_1, \ldots, e_{n-1}] = e_0 \oplus e_1 \oplus \cdots \oplus e_{n-1}$$

If the elements are real numbers and the operator is the addition operator, *reduce* operation computes the sum of all given real numbers. The sequential computation requires n operations when applied on a list of n numbers.

An efficient parallel computation is based on a "tree-like" computation that is derived from the recursive definition of the *reduce* function:

$$reduce(\oplus)[e] = e$$
$$reduce(\oplus)[p|q] = reduce(\oplus)[p] \oplus reduce(\oplus)[q]$$

where $|$ is the concatenation operator. an example is illustrated in the Figure 1 [1].

[1] source: https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf

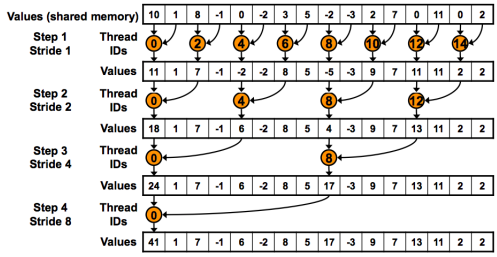

Figure 1: The computation of the sum by using tree-like computation (Mark Harris, 2022).

As it can be seen in the Figure 1 the program creates a number of threads equal to the number of elements – n, and the computation is done in $k = \log_2 n$ steps. At each step the threads with an ID divisible with the $stride * 2$ will add to its corresponding element $E[ID]$ the value on the position equal to $ID - stride$. In order to obtain the correct answer is mandatory that this operation to be executed only after the value on the position $ID - stride$ was already updated in the previous step. The thread execution is done theoretically completely in parallel but the execution of operations executed by different threads is non deterministic (i.e. implicitly not all the threads execute the operations on one level in the same time). In order to impose this synchronization different mechanisms could be used and lead to 4 variants: with a `barrier` after each level,or with synchronization between pairs that need to exchange information using `wait-notify`, `Condition`, or `Exchange`.

If we consider also multithreading variants derived from the sequential algorithm, the numbers are split between the threads and each updates of the sum variable with its own value inside a critical section. Obviously, it is not an efficient approach since the computations is still done sequentially due to the critical section. These variants are considered here just as means to evaluate the corresponding weights w_{sync} and w_{lock}.

Metric Evaluation for Reduce Variants

For each variant presented in the previous section we evaluate the O metric, and then we will compare the results with the concrete execution times obtained for the variants execution on two different systems. The goal is to obtain an estimation of the weights defined in the metric.

V1 `Barrier`
– a barrier is used after each level in the tree execution;

$$\begin{array}{lcl} \#bar & = & \log_2 n = k \\ DIS & = & k/(3k) = 0.33 \end{array}$$

V2 `synchronized + wait`
At each level there are pairs of threads (thread ID should use the value of thread $(ID-stride)$ only after this one finished previous update) that needs to synchronize one to another through `wait-notify` mechanism. All these pairs are disjunctive, and at each step the number of pairs that need to synchronize in pairs decreases by 2.

$$\begin{array}{lcl} \#wait & = & \sum_{i=1}^{\log_2 n} n/2^i = n \\ DIS & = & 2k/(5k) = 0.4 \end{array}$$

V3 `locks + await from Condition`
At each level there are pairs of threads that needs to synchronize one to another through conditional variables. (Similar to variant V2.)

$$\begin{array}{lcl} \#cond & = & \sum_{i=1}^{\log_2 n} n/2^i = n \\ DIS & = & 2k/5k = 0.4 \end{array}$$

V4 `Exchanger`
At each level there are pairs of threads that needs to synchronize one to another through an `Exchanger`. All these pairs are disjunctive.

$$\begin{array}{lcl} \#ex & = & \sum_{i=1}^{\log_2 n} n/2^i = n \\ DIS & = & 2k/(5k) = 0.4 \end{array}$$

V5 Critical section with `synchronized`
There is one critical section of size 1.

$$\begin{array}{lcl} \#sync & = & 1; \quad cs_i = 1 \\ DIS & = & 0 \end{array}$$

V6 Critical section with `Lock`
There is one critical section of size 1.

$$\begin{array}{lcl} \#lock & = & 1; \quad cs_i = 1 \\ DIS & = & 0 \end{array}$$

Depending on the value of the weights we may estimate which variant could be better on different systems. For all the variants, the source code of the `run()` function was analysed.

The evaluation of the weights is not an easy task, and we propose, as a first step, an empirical strategy based on the execution time for all previously specified variants, on different arrays' size and on different machines.

4.5 Experiments and Weights Evaluation

Our replication strategy considers experiments of the variants of reduction operation implementation on different computation machines – (**C1** and **C2**). The two machines systems have the following characteristics: **C1** – a computer with 1 processor Intel(R)

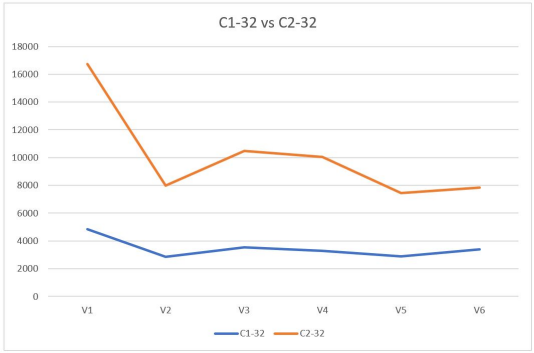

Figure 2: The execution time for n=32 on both C1 and C2 machines.

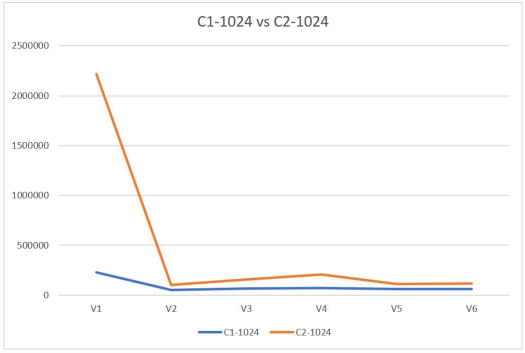

Figure 3: The execution time for n=1024 on both C1 and C2 machines.

Core(TM) i7-7500U CPU @ 2.50GHz, 4 Core(s) with hyperthreading, running Java 8 (macOS); **C2** – an x3750 M4 machine, with 4 processors Intel Xeon E5-4610 v2 @ 2.30GHz CPUs, 8 cores per CPU with hyperthreading, running Java 8 (CentOS 7). Due to stochastic nature of the programs' execution, they must be repeated several times in order to mitigate against the effect of random variation. We have used in our evaluation 30 executions and we took their average.

The results are shown in Figure 2 and Figure 3 where time is expressed in microseconds. It can be noticed that the barrier variants for the both cases are much higher, and this is why the variations on the 1024 threads cases are not quite visible in the figure. For $n = 1024$ the execution time for the barrier version is much higher, but the other variants preserve almost the same shape as for $n = 32$ case.

The strategy that we followed in order to approximate the weights was based on computing the differences between the execution times, and relies on the observed fact that the execution time of the first variant (V1- Barrier) it is always the highest.

So, the steps of our evaluation strategy were:

1. Measure the execution times for the six describe variants on the two machines C1 and C2 for two cases: U1: $n = 32$; U2: $n = 1024$.
2. Compute for each case Ci-Uj ($1 \leq i, j \leq 2$) the following differences:
   - $T(V1) - T(V2)$;
   - $T(V1) - T(V3)$;
   - $T(V1) - T(V4)$;
   - $T(V1) - T(V5)$;
   - $T(V1) - T(V6)$

   In this way, we eliminate the thread management overhead time.
3. Each variant (V1-V6) depends on only one type of weight and we can start from a fix value of one of these. We started by considering $w_{bar} = 1 + \frac{w_{load}}{1000}$.
4. Compute the values of the others weights by applying the formulas determined at step 3 based on the metric for each variant.

In order to estimate the final value for weights we also used values for t_S that have been approximated based on the systems characteristics. Also, the approximation of the function f was done based on observation ($\sigma = 100$); more different values have to be tried for further evaluations.

In Table 1 we added the results of the evaluation of the weights following the described strategy. The values show that the differences are not very big and so they are promising as a first step for the estimation.

Table 1: The estimated values for the weights based on the empirical evaluation.

	Variants				
	32		**1024**		
weights	***C1***	***C2***	***C1***	***C2***	***mean***
w_{bar}	1.004	1.008	1.128	1.03	1.04
w_{wait}	0.93	1.59	1.03	1.71	1.31
w_{cond}	2.29	2.56	1.81	1.94	2.15
w_{ex}	1.80	2.38	2.06	2.15	2.09
w_{sync}	0.97	1.38	1.43	1.76	1.38
w_{lock}	2.02	1.53	1.50	1.77	1.70

In order to arrive to a validated estimation, we propose, as a further work, to apply a more complex evaluation strategy that is based on artificial intelligence methods. The estimated values could be the starting point in the evaluation.

AI based strategies for weights evaluations. Such a strategy could be formed of the followings steps: (1) consider a set of already implemented parallel programs having different complexities; (2) evaluate the metric for each program; (3) measure the execution times of each program with different numbers of execution threads; (4) use a *Machine Learning*-based method (linear regression, neural network) or a *Genetic Algorithm*-based approach in order to tune the

values for the weights; (5) validate the results on new programs

The current paper offers preliminaries measurements and estimation for the overhead metric, which was theoretically validated.

5 CONCLUSIONS

An important desiderata in writing parallel programs is to reduce the overhead time which in some cases may cover the advantages of doing some computation in parallel. We have proposed a metric that estimates the overhead time for parallel Java SPMD multithreaded programs and we theoretically validated it; here Java programming language was considered but the metric can be easily adapted for other languages, since the synchronization mechanisms are very similar.

- estimate the overhead time at the development stage;
- compare different design solutions for a given problem, and choose the most optimal from the overhead point of view;
- estimate the need for refactorization of a given program by evaluating the improvements that could be achieved by changing the design method or only by changing the synchronization mechanisms.

We have shown that the metric definition corresponds to a proper definition of a software metric, and we provided a theoretical validation using Weyuker's properties. A first evaluation of the weights associated to the aggregated atomic metrics was done based on different implementation variants of the *reduce* operation; the results proved to be promising. Still, for a proper empirical validation this should be improved, and so, we proposed an AI based strategy that could be followed as a next step for attaining this goal.

REFERENCES

B, K. and N, F. (1995). Towards a framework for software measurement validation. *IEEE Transactions on software Engineering*, 21(12):929–943.

Basili, V., Briand, L., and Melo., W. (1996). A validation of object-oriented design metrics as quality indicators. 20(10):751–761.

Briand LC, E. K. and S, M. (1995). On the application of measurement theory in software engineering. Technical report, ISER Technical Report.

Chidamber, S. and Kemerer, C. (1994). A Metric Suite for Object- Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

David, F., Thomas, G., Lawall, J., and Muller, G. (2014). Continuously measuring critical section pressure with the free-lunch profiler. *SIGPLAN Not.*, 49(10):291–307.

EJ, W. (1988). Evaluating software complexity measure. *IEEE Transactions on software Engineering*, 14(9):1357–1365.

Espinosa, A., Margalef, T., and Luque, E. (1998). Automatic performance evaluation of parallel programs. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 43—-49.

Fenton, N. and Pfleeger, S. (1997). *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition.

Garg, V. K. (2004). *Concurrent and Distributed Computing in Java*. John Wiley fsSons, Inc., USA.

Göetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency In Practice*. Addison Wesley Professional.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing, Second Edition*. Addison-Wesley.

H, Z. (1992). On weyuker's axioms for software complexity measures. *Software Quality Journal*, 1(4):225–260.

Ji, M., Felten, E. W., and Li, K. (1998). Performance measurements for multithreaded programs. In *ACM SIGMETRICS Joint IC on Measurement and Modeling of Computer Systems*, pages 161–170. ACM.

Mark Harris. Optimizing Parallel Reduction in CUDA. https://docs.nvidia.com/cuda/. Online; accessed 25 January 2022.

Misra, S. and Akman, I. (2008). Applicability of weyuker's properties on oo metrics: Some misunderstandings. *Computer Science and Information Systems*, 5(1):17–23.

Raynal, M. (2013). *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer-Verlag Berlin Heidelberg.

Shah, M. D. and Guyer, S. Z. (2016). Iceberg: A tool for static analysis of java critical sections. In *ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 7–12. ACM.

Shah, M. D. and Guyer, S. Z. (2018). Iceberg: Dynamic analysis of java synchronized methods for investigating runtime performance variability. In *ISSTA/ECOOP Workshops*, pages 119–124. ACM.

Yin, R. K. (2008). *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, fourth edition. edition.