# Load balance in Parallel Loop Scheduling

Xingran Ruan

September 28, 2020

## 1 Fundamentals description

The syntax of a $FOR$ loop (usually referred to as a single loop) is illustrated below:

**for** $index = lower\_bound$ to $upper\_bound$,(stride) **do**
  $loop\_body$
**end for**

This permits repeat execution of the statement (referred to as the loop body) between $for$ and $end\ for$ a number of times, which depends on three parameters: the **lower bound**, the **upper bound**, and the **stride**[1]. Each execution of the loop body is known as an **iteration**. For each iteration, the index value is different. At first, the index is assigned with the value of lower bound. The index is increased by stride in each subsequent iteration, i.e., $index_{i+1} = index_i + stride$. The loop terminates once the index satisfies the terminal condition (for example, index exceeds lower bound or upper bound). In general, loop body may contain some internal loops; that is to say, an internal loop is surrounded by external loops. This case is known as **loop nest** and the number of loops surrounding a loop plus 1 indicates its **depth**. Henceforth, each loop is considered as a loop nest with depth $m \geq 1$, where $m$ is an integer value. The pseudocode shown below is an example of a loop nest with $depth = 3$.

**for** $index_i = lower\_bound_i$ to $upper\_bound_i$,(stride) **do**

  **for** $index_j = lower\_bound_j$ to $upper\_bound_j$,(stride) **do**

    **for** $index_k = lower\_bound_k$ to $upper\_bound_k$,(stride) **do**
      $loop\_body$
    **end for**
  **end for**
**end for**

A **parallel loop** is a kind of loops which there are no dependencies among its iterations, i.e. all iterations can be executed simultaneously and in any sequence. **Regular parallel loop** is a kind of parallel loops in which the execution time of each iteration is uniformly or linearly distributed (increasing or decreasing). There are two examples here showing regular parallel loops with uniform and linear execution time distributions.
*/*Uniformly distributed parallel loop*/*

[1]These three parameters, lower bound, upper bound and stride can be constants, variables, or arithmetic expression. Herein these three parameters are constants. If the stride is omitted, its value is 1 in default.

**for** $i = 0$ to $10$ **do**
  $Array[i] = Array[i] + c$;
**end for**

*/*Linearly increasing distributed parallel loop*/*

**for** $i = 0$ to $10$ **do**
  $Factorial(i)$;
**end for**

As can be seen in the above examples, the execution time of iterations has uniform distribution and increases linearly, respectively. Parallel loops are not always regular. In **irregular parallel loops** the execution time of each iteration is unknown and cannot be accurately predicted at compile-time. For example, Fig.1 shows the execution time of different iterations of an irregular parallel loop. As it can be seen in the figure, execution time of iterations various significantly.

Regular and irregular parallel loops can be scheduled on multi-processors in parallel to reduce the total completion time. In this mechanism, the iterations of a loop is partitioned into several **chunks**, and chunks are mapped to processors; each chunk contains a number of iterations and the number of iteration in the chunk expresses its size. How to **partition** a loop into chunks and **map** the chunks to processors to achieve a balance computation load is a problem. This is an NP-hard problem and is known as Parallel Loop Scheduling Problem (PLSP).

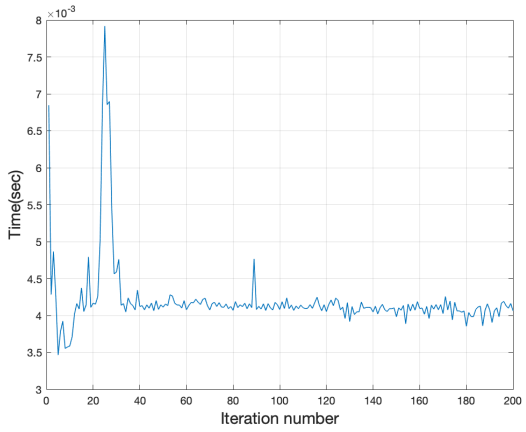


**Figure 1:** Execution time of $i$th iteration

**Load balance** is a key parameter to achieve time efficiency in PLSP. The greater volume of information about the loop is available, the easier it is to schedule the loop with a greater level of load balance. That is to say, the less volume of information about execution time of iterations results two challenges: (i) an inconsistent computation load of chunks; (ii) an inconsistent computation load assigned to processors. In other words, when the execution time is partially implicit at compile-time, unexpected load imbalance may happen which dramatically degrades the efficiency of scheduling. In order to mitigate the impact of the mentioned challenges in PLSP, a proper algorithm must be proposed for both partitioning and mapping steps. A fine-grained decomposition in combination with an appropriate mapping strategy can realize this goal. Loop scheduling algorithms boil down to following two categories: **static**, in which iterations are partitioned and mapped to processors at compile-time; and **dynamic**, in which partitioning and mapping decisions are made in run-time. Here, some classical static and dynamic loop scheduling algorithms are discussed as follows:

- *Static Block Scheduling (SBS)* [1]: It divides the iterations into $p$ chunks of size $\lceil n/p \rceil$; where $n$ and $p$ indicates the total number of iterations and the number of processor, respectively. Then, chunks of consecutive iterations are assigned to processors in round-robin method.

- *Pure Dynamic Scheduling (PDS)*[2]: In this method each iteration is scheduled on-demand. That is to say, whenever a processor becomes idle, the next waiting iteration is assigned to it. Although, this strategy achieves good load balancing, it suffers from high run-time overhead.

- *Chunk Self-Scheduling (CSS)* [2]: This method first partitions a loop into chunks with equal size to propose an extension of PDS. Partitioning loops into smaller chunks guarantees a good load balance but at the price of increasing run-time overheads. In contrast, partitioning loops into larger chunks can obviate this problem but increase load imbalance.[2]

- *Guided Self-Scheduling (GSS)* [3]: It tries to propose a tread-off between load balance and run-time overhead by decreasing the size of chunks during the run-time. In this method, iterations are dynamically mapped into chunks during run-time and chunks are assigned to processors on demand. The size of each chunk is calculated as $r/p$, where $p$ and $r$ is the number of idle processors and the number of remained iterations, respectively. However, the size of initial chunks is significantly large in this method.

Some experiments have been done to assess these classical loop scheduling algorithms. The details of these experiments and the obtained results are summarized in Appendix.

Different loop scheduling algorithms have been proposed based on these classical algorithms. In *Factoring Self-Scheduling*[4] the smaller number of iterations are partitioned into initial chunks to overcome the problem of large size of initial chunks in GSS method. The size of each chunks is increased and calculated as $Chunk\_Size = r/(2 \cdot p)$.

In *Trapezoid Self-Scheduling*[5] two parameters $l$ and $u$ are defined to determine the minimum and maximum size of chunks, respectively. For initial chunks the size is considered $u$ and the size is gradually decreased using a linear function as $cs_{i+1} = cs_i - d$, where $cs_i$ is the size of $i-$th chunk and $d$ is a decreasing rate calculated as $d = (u - l)/(s - 1)$. $s$ is the total number of chunks which can be determined as $s = 2 \cdot n/(u + l)$.

The authors in [2], [6] believe that the information provided by code editor can be applied to estimate the execution time of each iteration. Thus, taking this information into account, they develop a method named *BinLPT* for PLSP which first estimates the execution time of each iteration. Then, the iterations are sorted in decreasing order and PLSP is modelled as a bin packing problem. *BinLPT* partition iterations from the beginning of list; it starts partitioning the next chunk $Ch_{k+1}$, once $\sum_{i \in Ch_k} t_i \geq \sum_{j=1}^{n} t_j/\hat{q}$. $t_i$ is the execution time of iteration $i$. The partitioned chunks are mapped to processors by longest-time-first, i.e. the chunk with longest execution time is mapped to the first idle processor.

However, there are lots of **composite loops** in real world application which contain both regular and irregular iterations. For example, a loop with body which contains statements as $IF(\ldots)$ $Regular\_body$ $ELSE$ $Irregular\_body$ is regular in some iterations and irregular in other iteration depending on the $IF$ statement. Different patterns have

[2]CSS is a generalization of PDS in which chunk size equals to one.

been proposed to distinguish regular and irregular iterations in composite loops. Proposed method for scheduling the regular or irregular may have shortage to deal with composite loops.

## 2 Problem description

In a loop nest, the number of times that statements in the body of an internal loop is executed can be calculated using Eq. (1).

$$q = \sum_{i_1=l_1}^{u_1} \sum_{i_2=l_2}^{u_2} \cdots \sum_{i_m=l_m}^{u_m} 1. \tag{1}$$

where $m$ is the depth of the internal loop; $l_j$ and $u_j$ are lower bound and upper bound of the loop in the $j-$th depth, $1 \leq j \leq m$. Scheduling composite loops, which is called *Composite loop scheduling problem (CLSP)*, has three sub-problems: (i) pattern detection which distinguishes the regular and irregular iterations in the loop; (ii) chunk determination which partitions the iterations into chunks; and (iii) chunk mapping which maps the chunks to the processors. Let $q$ indicates the number of iterations in a composite loop. The goal is to partition the $q$ iterations into $k$ chunks and map the chunks to $p$ processors to run the composite loop realizing a trade-off between load balance and run-time overhead. Following each sub-problem is explained in more details.

- Pattern Detection (PD): As it was mentioned in Section 1, a composite loop may contain both regular and irregular iterations. It is more efficient to schedule regular and irregular iterations with different strategies. Thus, how to detect the loop pattern to distinguish regular and irregular iterations is a significant sub-problem in CLSP.

- Chunk Determination (CD): In parallel loop scheduling, the total $q$ iterations is partitioned into $k$ chunks. The number of chunks affects load balance in processors. When the number of chunks equals to iteration number, i.e. $k = q$, each iteration is assigned to a processor on-demand. In this case, it achieves good load balance in processors but suffers from high run-time overhead. Decreasing the number of chunks mitigates the high run-time overhead, but results in increasing load imbalance. Therefore, how to find the best value for $k$ to achieve good balance in processors is the second sub-problem in CLSP.

- Chunk Mapping (CM): There are two approaches for mapping chunks to processors, static and dynamic. *Static approach* assigns the iterations to processor on the basis of a predefined algorithm such as round-robin or random. *Dynamic approach* aims to assign a chunks to the best idle processor. For example *longest processing time* assigns the largest chunk to the processor with the least load processor. Static approach has lower run-time overhead but at the price of increasing load imbalance; on the contrary, Dynamic approach guarantees good load balance but suffers from run-time overhead. Thus, how to map the chunks to processors to meet a good load balance with a reasonable run-time overhead is the third sub-problem in CLSP.

# 3 Evaluation strategy

There are two different scenarios to evaluate a method in CLSP: (i) application scenario which applies real world application for experiments, Rodinia benchmark suite[7] and OpenMP Source Code Repository[8] are some benchmarks for this scenario. (ii) synthetic scenario which applies Monte-Carlo method to generate artificial loops. In Monte-Carlo method irregular iterations are generated with mean $\mu$ and variance $\sigma$, while regular iterations are generated on the basis of uniformly or linearly distributions.

Following evaluation metrics[2] can be used to assess a method: (i) Parallel Time, which is the overall execution time of a parallel loop; (ii) Performance, which is the ratio of the total number of iterations to the parallel time; (iii) Coefficient of Variance, which is the ratio of the standard deviation to the mean execution time of the processors; (iv) Slowdown, which is the ratio of the execution time of the slowest processor to the fastest one.

# 4 Future research plan

Given the both regular and irregular iterations in a composite loop, proposed methods for regular loops or irregular loops cannot guarantees load balance in CLSP. To handle the CLSP, three sub-problems, PD, CD and CM, should be redefined and be solved with compatible algorithms for composite loops. Thus, in the future research:

- It is tried to propose a method to detect the pattern of target loop and distinguish the regular and irregular iteration sets.
- Based on the detected pattern, a partitioning algorithm is proposed determine the best value for $k$ to achieve a trade-off between load balance and the total number of chunks.
- It is tried to propose a new mapping strategy to assign the chunks to processors to minimize the load imbalance among processors.

| **Gantt chart** | Sep/2020 - Sep/2021 | | | | Oct/2021 - Sep/2022 | | | |
|---|---|---|---|---|---|---|---|---|
| Synthesize literature on CLSP | | | | | | | | |
| Distinguishes the regular and irregular iterations in the loop | | | | | | | | |
| Measure best chunk size | | | | | | | | |
| Propose method which can map the chunks to the processor | | | | | | | | |
| Preliminary data | | | | | | | | |
| Design experiment and analysis proposed model | | | | | | | | |
| Make improvements | | | | | | | | |
| Write PhD thesis | | | | | | | | |

## References


[1] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *1993 International Conference on Parallel Processing-ICPP'93*, IEEE, vol. 2, 1993, pp. 140–147.

[2] P. H. Penna, A. T. A. Gomes, M. Castro, P. DM Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "A comprehensive performance evaluation of the binlpt workload-aware loop scheduler," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 18, e5170, 2019.

[3] C. D. Polychronopoulos and D. J. Kuck, "Guided self-scheduling: A practical scheduling scheme for parallel supercomputers," *Ieee transactions on computers*, vol. 100, no. 12, pp. 1425–1439, 1987.

[4] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A practical and robust method for scheduling parallel loops," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, 1991, pp. 610–632.

[5] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on parallel and distributed systems*, vol. 4, no. 1, pp. 87–98, 1993.

[6] P. H. Penna, M. Castro, P. Plentz, H. C. Freitas, F. Broquedis, and J.-F. Méhaut, "Binlpt: A novel workload-aware loop scheduler for irregular parallel loops," *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho-WSCAD*, 2017.

[7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *IEEE International Symposium on Workload Characterization (IISWC'10)*, IEEE, 2010, pp. 1–11.

[8] A. J. Dorta, C. Rodriguez, and F. de Sande, "The openmp source code repository," in *13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, IEEE, 2005, pp. 244–250.


# Appendices

The performance of SBS, PDS, CSS, and GSS is evaluated in these experiments. The chunk size in CSS method is set 4 and 5, indicating by CSS_4 and CSS_5. The number of processors is set 4 and these processors are named $P0, P1, P2, P3$.

In the first experiment performance of the methods on regular loops is investigated. In this experiment, uniform distribution is used to generate the regular iterations, where the total number of these iterations is set 20 and the body of each iteration is shutting for 1 second. The load balance achieved by each method is shown in Fig.2. As it can be seen in the figure SBS, PDS and GSS all achieve good load balance. These algorithms generate 4 chunks, 20 chunks, and 9 chunks, respectively. According to the results shown in Fig.2, CSS_5 provides better load balance in comparison to CSS_4. The number and also the size of the chunks in each of these methods are shown in Table.1. Considering both evaluation parameters, load balance in Fig.2 and the number of total chunks in Table.1, SBS outperforms the other methods. It is seen that methods which

apply dynamic schedulers, PDS, CSS and GSS, achieve good load balance but the number of total chunks in these methods is relatively greater.

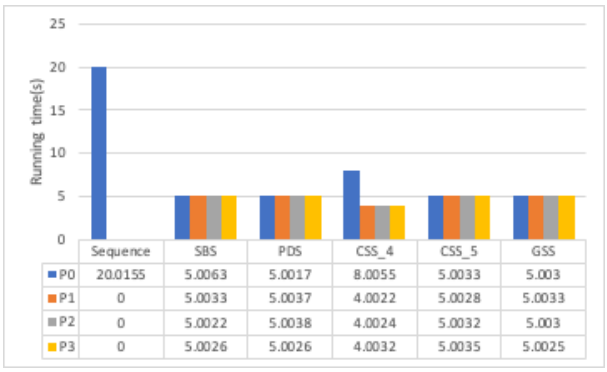


**Figure 2:** The running time (second) on processors, $P0, P1, P2, P3$, when schedule uniformly distributed *For* loop (iteration number = 20) by Sequence, Static Block Scheduling (SBS), Pure Dynamic Scheduling (PDS), Chunk Self-Scheduling with $Chunk\_Size = 4$ (CSS), Guided Self-Scheduling (GSS).

**Table 1:** Sample chunk size of Static Block Scheduling (SBS), Pure Dynamic Scheduling (PDS), Chunk Self-Scheduling with $Chunk\_Size = 4$ and $Chunk\_Size = 5$ (CSS_4 and CSS_5), Guided Self-Scheduling (GSS) when schedule uniformly distributed *For* loop; under the condition: iteration number = 20 and processor number = 4.

| Scheme | Chunk size | Total chunks |
|---|---|---|
| SBS | 5 5 5 5 | 4 |
| PDS | 1 1 1 1 ... | 20 |
| CSS_4 | 4 4 4 4 4 | 5 |
| CSS_5 | 5 5 5 5 | 4 |
| GSS | 5 4 3 2 2 1 1 1 1 | 9 |

The second experiment is dedicated to investigation of the performance of the methods on irregular loops. Mandelbrot fractal computation application[3] is used in this experiment as irregular loop. This application generates irregular loops based on some parameters such as domain and meshed grid size. We set the application domain as $[-1.5, 1.5]\times[-1, 1]$ with meshed grid size $78 \times 22$. The load balance and the number of total chunks achieved by each method are shown Fig.3 and Table.2, respectively. The obtained results show that SBS, which uses static schedule algorithm, results bad performance based on load balance evaluation parameter despite small total number of the chunks. In opposite, PDS achieves good balance but at the price of more number of chunks. Considering a larger chunk size, CSS decreases the total number of chunks but it fails to realize the load balance among processors. Given to the adopting flexible chunk size, GSS outperforms the other methods based on the total number of chunks and load balance.

[3]The Mandelbrot set is the set of complex numbers $c$ for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$, i.e., for which the sequence $f_c(0)$, $f_c(f_c(0))$, etc., remains bounded in absolute value.

**Table 2:** Sample chunk size of Static Block Scheduling (SBS), Pure Dynamic Scheduling (PDS), Chunk Self-Scheduling with $Chunk\_Size = 4$ (CSS), Guided Self-Scheduling (GSS) to schedule Mandelbrot set computation on processors $P0, P1, P2, P3$. Under the condition: on the domain $[-1.5, 1.5] \times [-1, 1]$ with meshed grid 22 rows and 78 columns.

| Scheme | Chunk size | Total chunks |
|---|---|---|
| SBS | 429 429 429 429 | 4 |
| PDS | 1 1 1 1 ... | 1716 |
| CSS | 4 4 4 4 ... | 429 |
| GSS | 429 322 242 181 136 102 76 57 43 32 24 18 14 10 8 6 4 3 3 2 1 1 1 1 | 24 |

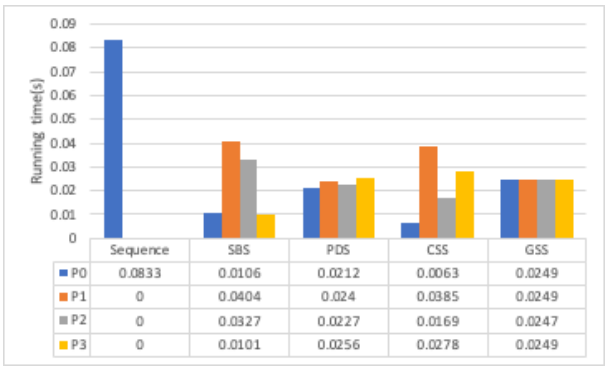


**Figure 3:** The running time (second) on processors, $P0, P1, P2, P3$, when schedule Mandelbrot set computation by Sequence, Static Block Scheduling (SBS), Pure Dynamic Scheduling (PDS), Chunk Self-Scheduling with $Chunk\_Size = 4$ (CSS), Guided Self-Scheduling (GSS); under the condition: on the domain $[-1.5, 1.5] \times [-1, 1]$ with meshed grid 22 rows and 78 columns.