Index Dependent Nested Loops Parallelization with an Even Distributed Number of Steps

Ádám Pintér and Sándor Szénási
John von Neumann Faculty of Informatics, Óbuda University, Hungary
Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University, Hungary
E-mail: pinter.adam@nik.uni-obuda.hu

John von Neumann Faculty of Informatics, Óbuda University, Hungary
Faculty of Economics, J. Selye University, Slovakia
E-mail: szenasi.sandor@nik.uni-obuda.hu, szenasis@ujs.sk

Keywords: parallel algorithm, optimization, nested loop parallelization, index distribution

Received: April 27, 2020

Parallel processing of algorithms is an effective way to achieve higher performance on multiprocessor systems rather. During parallelization, it is critical to minimize the difference between the processing time for threads. It is necessary to choose a method that can efficiently distribute the workload evenly across the threads. This paper deals with a special kind of nested loops where the internal loop iterator depends on the outer loop iterator. In such cases, the process can be represented as an upper (or lower) triangular matrix. This paper introduces a method for partitioning the outer loop according to the indices in an almost optimal manner, so that the partial loops in each thread will take nearly the same number of steps. In addition, we examine the potential of a perfect partition and try to determine the maximum (but still meaningful) partition size.

Povzetek: Predstavljena je metoda paralelizma za posebno vrsto vgnezdenih zank.

1 Introduction

There are multiple ways to find duplicated elements of a dataset, but in the cases where the size of the set drastically increases, simple sequential solutions have serious runtime limitations. In such cases, it is advisable to parallelize this process for faster execution.

The most common way to find duplicates is the brute force method, which compares each element to every other element in the dataset. This solution has the time complexity of $O(n^2)$ and is rarely used in the real world. [17] An advanced version of this method considers that an element will always be self-consistent (reflexivity) and assumes that if an element is identical to the other, the inverse of the condition is also satisfied (symmetricity), or necessarily (for a more general solution) transitivity is not allowed. Given these rules, we got a non empty $S \neq \emptyset$ set and a reflexive symmetric non-transitive R relation:

$$\forall a \in S : (aRa)$$
$$\forall a, b \in S : (aRb \Leftrightarrow bRa)$$
$$\exists a, b, c \in S : (aRb \wedge bRc) \nRightarrow aRc$$

Suppose that we want to construct equivalence classes between the elements of the dataset so that the matching elements (duplicates) are considered as one class. If the dataset element pairs are represented as a matrix where the rows and columns represent the i-th element of the set, and each cell indicates whether a comparison is needed, then the matrices are shown in Figure 1. If the R relation is ignored, that is, the brute force algorithm is used in the processing, then for a set of 20 elements 400 comparisons (Figure 1.b.) are needed. If we consider the reflexivity attribute of the relation, we can reduce the number of comparisons by 380 (Figure 1.c.), since the elements of the main diagonal can be omitted. If the symmetric property is included along with reflexivity, the number of comparisons will be reduced to 190 (Figure 1.d.), since the property guarantees that the order of two elements is interchangeable. Because of non-transitivity, it will be necessary to traverse this upper triangle to find all duplication for each element.

In this case at best options $\frac{(N^2-N)}{2}$ comparisons are required (where N is the number of elements in the dataset), which cannot be further reduced (or more precisely, cannot be further reduced without additional information about the dataset). For example, if the dataset contains 10,000 elements, then 4,999,500 comparisons are required.

Although transitivity exists as an attribute during the testing for equality of elements in a dataset, non-transitivity may be unavoidable for other tasks satisfying the R relation. Examples of such cases include:

- In the first example, the elements of the dataset represent the countries, where the R relation represents the neighborhoods. It is said that country *A* borders *B*

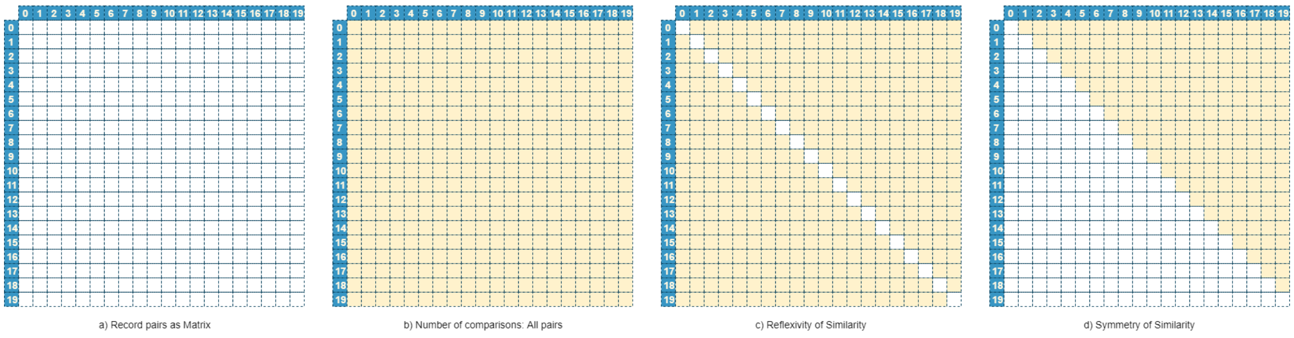
a) Record pairs as Matrix
b) Number of comparisons: All pairs
c) Reflexivity of Similarity
d) Symmetry of Similarity

Figure 1: Dataset element pairs as a matrix marked with the different type of comparison requirement.

country, as well as *B* and *C* are neighbours, but that does not mean that countries *A* and *C* are also neighbours.

– In the second example, the S dataset contains a set of one-dimensional points where the relation R describes the distance between elements. It is also true that if there is γ distance between the *A* and the *B* and γ distance between *B* and *C*, it does not follow that there is also γ distance between *A* and *C*. In this case, non-transitivity will be met for all elements.

– Our last example concerns the topic of graphs, where S dataset contains the nodes of the graph and the relation R describes the edges between them. Here, to give an example, if a vertex C can be reached from A through B, it does not follow that there is an edge between A and C: $A \longrightarrow B \longrightarrow C$.

The third relation is the non-transitivity which cannot simplify the algorithm further. This case must check every potential pairs in the S set to determine the equivalence classes.

1.1 Sequential approach

To process the S dataset it is necessary to implement nested loops where the loops iterate through twice the whole set of data in case of brute force algorithm. (Algorithm 1) shows the pseudo code of the algorithm, called Naive Sequential Approach. In this case the main problem is the number of steps of the algorithm. For example, if the set size is $N = 20$ the number of iterations (incrementation of one of the loop variables) is $F_N = N^2 = 400$ and the number of comparisons is $C_N = N^2 = 400$.

Number of iterations and comparisons can be highly reduced if the elements meet the reflexivity and symmetricity requirements. In this case, it is enough to start the inner loop from the $i + 1$-th element due to symmetry. (Algorithm 2) shows the modification, where the main advantage is to link both objects in the same iteration due to reflexivity; this algorithm is called Improved Sequential Approach. Due to the change, the number of iterations is reduced to $F_N = \frac{N*(N+1)}{2} = 210$ and the number of comparisons is

Algorithm 1 Naive Sequential Approach

```
Require: $S$: series of data
Require: $N$: number of elements in the series
Ensure: $S$: series of processed data
 1: function NAIVESEQUENTIALAPPROACH($S$, $N$)
 2:     for $i \leftarrow 1$ to $N$ do
 3:         for $j \leftarrow 1$ to $N$ do
 4:             if $(S[i].P_1\ R\ A[j].P_1)$ then        ▷ $R$ is the relation between elements
 5:                 $S[i].P_2 = S[i].P_2 \cup \{A[j]\}$
 6:             end if
 7:         end for
 8:     end for
 9:     return $S$
10: end function
```

In the pseudo code the dataset contains objects having P_1 and P_2 properties. Where the P_1 properties match, it links the target object to the P_2 property of the source object.

reduced to $C_N = \frac{N^2-N}{2} = 190$ for the same number of elements.

The algorithm presented in this article is generic and can handle all kinds of datasets (including ones without transitivity), such as string arrays or even object lists (as long as the R relation can be interpreted). The rest of this paper is organized as follows. In the next Section, an example of a triangular loop nest parallelization - which is distributed efficiently - using our technique is used to show the motivation of the paper and the already existing related results. Section 4 explains the mathematical aspects of the proposed technique and discusses its limitations. Experiments are presented in Section 5, highlighting the significant time improvements provided by the index-based distribution. And finally, we present our conclusion in the last section.

2 Related work

The parallelization of index dependent nested loops has been considered and developed by several authors to minimize execution time and optimize processor core utiliza-

Algorithm 2 Improved Sequential Approach

```
Require: $S$: series of data
Require: $N$: number of element in the series
Ensure: $S$: series of processed data
 1: function ImprovedSequentialApproach($S$, $N$)
 2:     for $i = 1$ to $N$ do
 3:         for $j = i + 1$ to $N$ do    ▷ Due to the reflexivity.
 4:             if ($S[i].P_1\ R\ A[j].P_1$) then    ▷ $R$ is the relation between elements
 5:                 $S[i].P_2 = S[i].P_2 \cup \{A[j]\}$
 6:                 $S[j].P_2 = S[j].P_2 \cup \{A[i]\}$
 7:             end if
 8:         end for
 9:     end for
10:     return $S$
11: end function
```

tion. [11, 3, 15, 10, 6, 4, 9, 1, 12]

The collapsing of perfectly nested loops with constant loop bounds was originally introduced by Polychronopoulos [14] as loop coalescing, and Philippe Clauss et. al. [13] was further developed to work with non-rectangular loops. The problem of the original implementation - which only operates with loops that define constant loop boundaries - has been solved by defining non-rectangular iteration spaces whose bounds are linear functions of the loop iterators.

In [8], Nedal Kafri and Jawad Abu Sbeih found a solution to how to partition the triangular iteration space nearly optimal. In their case, the iteration space is defined as a lower triangle matrix and presents a method for calculating the lower and upper bound index of the outer loop of each partition. They have been able to achieve near-optimal load balancing and minimize load imbalance in parallel processing of a perfect triangular loop nest.

In [16], Rizos Sakellariou introduced a compile-time scheme that can efficiently partition non-rectangular loops whose indexes of internal loops depend on those of outermost loop. The technology presented is based on symbolic cost estimates, which minimize the imbalance of the load while avoiding other additional sources.

Adrian Jackson and Orestis Agathokleous [7] presented a developed system that allows a code to dynamically select which parallelization method to use at runtime. During the operation of their system, the programmer only has to specify the loop to be parallelized, after which the applied parallel technique will be selected by their dynamic library during the execution of the code.

3 Naive parallel approach

Considering the nested loops, the iterators *i* and *j* doesn't carrying any dependency, so they can be parallelized. [18] However, it is worth noting that the number of *j* iterations is not constant, it depends on the current value of *i*. If parallelization is performed according to the external cycle, the result is an imbalanced runtime for the threads. Therefore, it is necessary to choose another parallelization strategy (if we keep the original approach) to improve the runtime of the algorithm.

For example, if we keep the outer loop parallelization and create threads based on the number of outer loop iterations, the problem will be the difference in the runtime of the threads. The last thread will be completed much sooner than the first, whose internal loop will iterate through the entire dataset. In this case, the total runtime of the parallel algorithm will be determined by *thread 1*, as illustrated by Figure 2 and pseudo code called Naive Parallel Algorithm shown in Algorithm 3.

Therefore, if we simply halve the outer loop executions, the total iterations of the loops are 155 in the first part and 55 in the second part. So, the second part is only one third of the first one and finishes much earlier. It would better to use equal 105-105 iterations in both threads.

In our solution, we present a method that implements outer loop parallelization for the purpose of obtaining a nearly equal iteration of each threads, thereby eliminating differences in thread runtime.

3.1 Nested loop parallelization

Numerous fields (numerical calculations, big data, etc.) use nested loops to either compute mathematical formulas or process large amounts of data, which typically stored in (often multidimensional) arrays. In the case of arrays, data is accessed through nested loops, their dependency relies on the data type and the processing method. When parallelizing such cases, it is first necessary to determine at which level of the cycles we want to parallelize, the strategies of which are illustrated in Table 1. [4, 7, 5, 19]

Table 1: Most common nested loops parallelization strategies

Type	Description
Outermost	Parallelization of the outermost loop.
Inner	Parallelization of one of the inner loop.
Nested	Parallelize multiple nested loops.
Collapsing	Nested loops collapse into a single loop.

3.1.1 Outermost

This is the most common approach for parallelizing nested loops. In this strategy, the outermost cycle will be parallelized, with iterations distributed between the threads, thereby running the threads in parallel and performing the tasks assigned to them. In this case, the internal cycles are executed sequentially.

This strategy is generally a good choice (especially for high iteration counts) as it minimizes the cost of par-

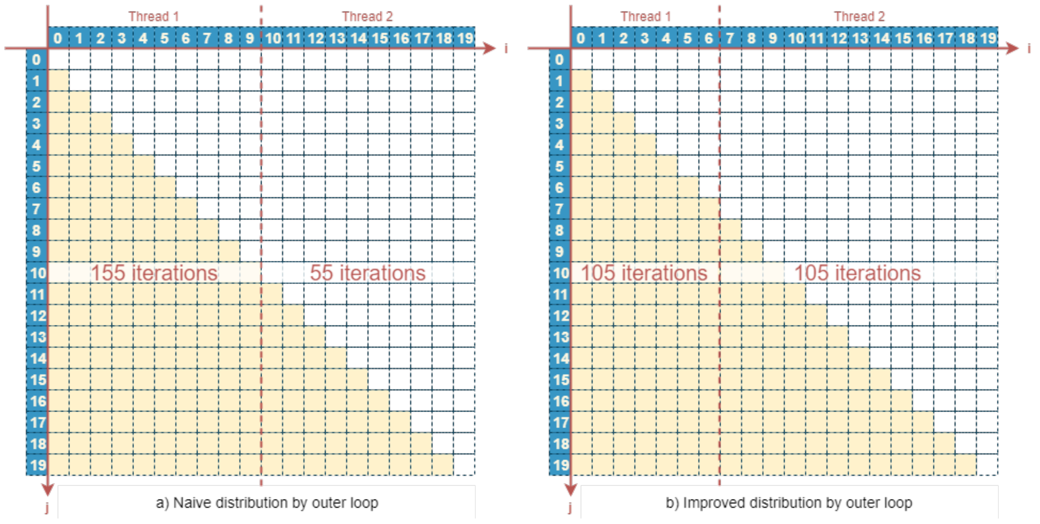

Figure 2: Unbalance of the runtime of the threads based on a uniform distribution of iterations.

allelization (for example, initializing threads, scheduling loop iterations on threads, thread synchronization).

Despite the advantages, the disadvantage is the limit on the maximum level of parallelization, which must not exceed the number of iterations of the loop to be parallelized. For example, a cycle that goes from $i : 1 \to N$ can be divided into a maximum of N parts. This can limit the number of threads, which may prevent taking advantage of using all processors and cores of the system.

3.1.2 Inner

A variant of outermost parallelization, with the difference that one of the internal cycles is parallelized (as opposed to the external) while the outermost loop is executed sequentially. This strategy is necessary or useful if the external loop does not have a sufficient number of iterations for efficient parallelization, but it has the disadvantage that initializing, managing, and synchronizing threads can lead to performance problems.

3.1.3 Nested

This strategy takes advantage of the opportunity to execute multiple loops in parallel. Unlike the previous ones, which can use up to maximum as many threads as the number of iteration loops, this strategy also provides additional parallelization, which can give good results on systems with a large number of processors.

3.1.4 Collapsing

The basis for loop collapsing is that it transforms nested loops into a single loop, and then the newly created loop will be parallelized. The advantage of the transformation is that the ratio of parallelization is increased by the fact that the newly created loop will have a greater number of iterations. The method produces better runtime results than inner and nested strategies, since reduces the amount of loop overhead (although this is not always possible as not all compilers provide this functionality).

4 Methodology

4.1 Index based equal distribution

To balance the number of iterations in every threads, it is necessary to determine the optimal number (O) of iteration in one partition based on the size of dataset (N) and the target number of partitions (P) and calculate lower (inclusive) and upper (exclusive) indices for all partitions. It is important to note that partitioning happens according to the outer loop, so perfect resolution occurs only in exceptional cases. In all other cases, it is sufficient to achieve approximately the same number of iterations.

The optimal number of iterations for a partition is obtained by determining the number of all iterations (for a dataset of N elements): $F_N = \frac{N(N+1)}{2}$ which is divided by P to get the optimal number of iterations of a partition: $O = \frac{\frac{N(N+1)}{2}}{P} = \frac{N(N+1)}{2P}$.

Using the previous example, the optimal distribution of iterations can be determined as follows:

- $N = 20 \quad \longrightarrow \quad F_N = \frac{N(N+1)}{2} = \frac{20*(20+1)}{2} = 210$ - total number of iterations,
- $P = 2 \quad \longrightarrow \quad O = \frac{N(N+1)}{2P} = \frac{F_N}{P} = \frac{210}{2} = 105$ - iteration within 1 partition.

Our goal is to achieve this O iteration count for each thread. We know that the first thread will start processing the elements from $I_0 = 0$ (inclusive) index and the last thread index is $I_2 = 20$ (exclusive). The question is how to determine internal indices. In the example where we want to split the set into two threads, we only need to define an

Algorithm 3 Naive Parallel Approach

```
Require: $S$: series of data
Require: $N$: number of element in the series
Ensure: $S$: series of processed data
 1: function NaiveParallelApproach($S$, $N$)
 2:     $P$ = NumberOfProcessors()        ▷ Determine the optimal level of parallelism.
 3:     $T$ = CreateThreads(P)            ▷ Create $P$ number of working thread.
 4:     for $p = 0$ to $P - 1$ do
 5:         $lower = \left\lceil \frac{p*N}{P} \right\rceil$      ▷ Calculate the $lower$ (inclusive) index in thread.
 6:         $upper = \left\lceil \frac{(p+1)*N}{P} \right\rceil$  ▷ Calculate the $upper$ (exclusive) index in thread.
 7:         T[p] = ThreadProcess($lower$, $upper$, $S$)
 8:     end for
 9:     WaitAll($T$)
10:     return $S$
11: end function

12: function ThreadProcess($lower$, $upper$, $S$, $N$)
13:     for $i = lower$ to $upper$ do
14:         for $j = lower + 1$ to $N$ do
15:             if ($S[i].P_1 \; R \; A[j].P_1$) then     ▷ $R$ is the relation between elements
16:                 $S[i].P_2 = S[i].P_2 \cup \{A[j]\}$
17:                 $S[j].P_2 = S[j].P_2 \cup \{A[i]\}$
18:             end if
19:         end for
20:     end for
21:     return $S$
22: end function
```

internal index (will be: $I_1 = 6$), which will be an exclusive on the first, and an inclusive one on the second thread.

4.2 Steps of individual indices calculation

1. Determine the total number of iterations (F_N) based on dataset size (N): $F_N = \frac{N(N+1)}{2}$

2. Define the target number of partitions (P), than calculate the optimal number of one partition's iterations: $O = \frac{F_N}{P}$

3. Knowing the first index ($I_0 = K_0 = 0$), determine the following index approximation (K_1) for the partition using the following equations:

   - Number of steps from K_0 to the end of the set (K_0 is known):

   $$\frac{(N - K_0)(N - K_0 + 1)}{2} \quad (1)$$

   - Number of steps from K_1 to end of the set (K_1 is unknown):

   $$\frac{(N - K_1)(N - K_1 + 1)}{2} \quad (2)$$

   - Equation (3) is obtained by combining and explaining (1) and (2).

   - Roots of (3) are:

   $$a = -1$$
   $$b = 2N + 1$$
   $$c = \frac{P(N - K_0)(N - K_0 + 1) - N(N + 1)}{P} - N^2 - N \quad (4)$$

   - where $b^2 - 4ac > 0$, $-\frac{b}{a}$, $\frac{c}{a} > 0$ and $a < 0$ so $X_1 \leq X_2$ therefore: [2]

   $$X_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad (5)$$

   - from above we can calculate the (next) approximation to the index, K_1:

   $$K_1 = X_1 = \frac{-(2N + 1) + \sqrt{(2N + 1)^2 - 4\frac{P(N-K_0)(N-K_0+1)-N(N+1)}{P}}}{-2} \quad (6)$$

   - To get the real index (exclusive) need to round the K_1 (but to calculate the next K_x index, we use the unrounded K value):

   $$I_1 = \lceil K_1 \rceil \quad (7)$$

4. The index is calculated P times, and then $K_p = N$ is obtained.

This methods is the Improved Parallel Approach (IPA), and its algorithm is illustrated by Algorithm 4.

$$\begin{aligned}
\frac{(N-K_0)(N-K_0+1)}{2} - \frac{(N-K_1)(N-K_1+1)}{2} &= \frac{N(N+1)}{2P} \\
P(N-K_0)(N-K_0+1) - P(N-K_1)(N-K_1+1) &= N(N+1) \\
P(N-K_0)(N-K_0+1) - N(N+1) &= P(N-K_1)(N-K_1+1) \\
\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} &= (N-K_1)(N-K_1+1) \\
\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} &= N^2 - NK_1 + N - NK_1 + K_1^2 - K_1 \\
\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} &= N^2 - 2NK_1 + N + K_1^2 - K_1 \\
\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} &= N^2 + N + K_1^2 - K_1(2N+1) \\
\frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N &= K_1^2 - K_1(2N+1) \\
-K_1^2 + K_1(2N+1) + \frac{P(N-K_0)(N-K_0+1) - N(N+1)}{P} - N^2 - N &= 0
\end{aligned} \tag{3}$$

4.3 Example of intermediate indices calculation

In this example the length of the set is $N = 8$, which we would like to distribute into $P = 4$ parts. The total number of iterations are $F_N = \frac{N*(N+1)}{2} = \frac{8*(8+1)}{2} = 36$. One part optimally iterations are $O = \frac{F_N}{P} = \frac{36}{4} = 9$. From Equation 4 the $a = -1$ is constant, $b = 2*N+1 = 2*8+1 = 17$ is depends by the N, and c need to calculate in every iteration. The outer loop first index is also known: $K_0 = 0$. The intermediate indices' calculations are the following:

– Step 1 - calculate K_1 and I_1:

$$\begin{aligned}
c_1 &= \frac{4*8*9-8*9}{4} - 8^2 - 8 = -18 \\
K_1 &= \frac{-17+\sqrt{17^2-4*-1*-18}}{-2} = 1,135 \\
I_1 &= 1
\end{aligned} \tag{8}$$

For the first thread, the outer loop will iterate from 0 to 1, resulting in a total of 8 iterations.

– Step 2 - calculate K_2 and I_2:

$$c_2 = -36 \qquad K_2 = 2,479 \qquad I_2 = 2 \tag{9}$$

For the second thread, the outer loop will iterate from 1 *to 2, resulting in a total of* 7 *iterations.*

– Step 3 - calculate K_3 and I_3:

$$c_2 = -54 \qquad K_2 = 4,227 \qquad I_2 = 4 \tag{10}$$

For the third thread, the outer loop will iterate from 2 *to* 4*, resulting in a total of* 11 *iterations.*

– Step 4 - calculate K_4 and I_4:

$$c_2 = -72 \qquad K_2 = 8,000 \qquad I_2 = 8 \tag{11}$$

For the fourth thread, the outer loop will iterate from 4 *to* 8*, resulting in a total of* 10 *iterations.*

As visible, we cannot reach the equal distribution, but we get close to it. The optimal iteration count for one thread is $O = 9$. The differences to this are as follows: -1 (-11%), -2 (-22%), $+2$ (22%), $+1$ (11%), with a cumulative error of 0.

To compare these results to the Naive Distribution, it generates the following indices: $0-2$, $2-4$, $4-6$, $6-8$, with iteration counts as follows: 15 (44%), 11 (22%), 7 (-22%), 3 (-44%). It is clearly visible that the first thread will run approximately five times longer than the last one. In contrast, the threads of the improved version will finish at about the same time as is visible in Figure 3.

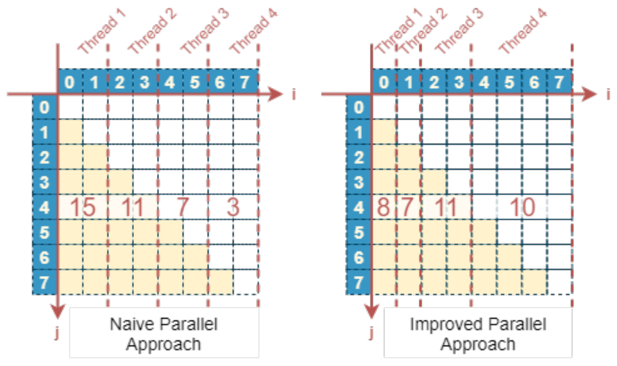

Figure 3: Difference between the two type of outer loop distribution.

4.4 Error in indices calculation

There are some cases where the last index will never be reached due to computational inaccuracy. Since rounding is an important step in defining indices, this problem will appear when the error "goes beyond" the decimal point, after that, rounding operation will not be able to correct the deviation. This error will only occur when calculating the last index, because in the case of intermediate indices, this difference is negligible as it causes minimal differences in iterations.

Algorithm 4 Improved Parallel Approach

```
Require: $S$: series of data
Require: $N$: number of element in the series
Ensure: $S$: series of processed data
 1: function IMPROVEDPARALLELAPPROACH($S$, $N$)
 2:     $P$ = NumberOfProcessors()          ▷ Determine the optimal level of parallelizm.
 3:     $T$ = CreateThreads(P)              ▷ Create $P$ number of working threads.
 4:     $a$ = -1
 5:     $b$ = $2N + 1$
 6:     $K_{lower}$ = 0
 7:     $I_{lower}$ = 0
 8:     for $p = 1$ to $P$ do
 9:         $c = \frac{P(N-K_{previous})(N-K_{previous}+1)-N(N+1)}{P} - N^2 - N$
10:         $K_{upper} = \frac{-b+\sqrt{b^2-4ac}}{2a}$
11:         $I_{upper} = \lceil K_{next} \rceil$          ▷ The $upper$ (exclusive) index in thread.
12:         T[p] = ThreadProcess($I_{lower}$, $I_{upper}$, $S$, $N$)
13:         $K_{lower} = K_{upper}$
14:         $I_{lower} = I_{upper}$
15:     end for
16:     WaitAll($T$)
17:     return $S$
18: end function

19: function THREADPROCESS($lower$, $upper$, $S$, $N$)
20:     for $i = lower$ to $upper$ do
21:         for $j = lower + 1$ to $N$ do
22:             if ($S[i].P_1\ R\ A[j].P_1$) then      ▷ $R$ is the relation between elements
23:                 $S[i].P_2 = S[i].P_2 \cup \{A[j]\}$
24:                 $S[j].P_2 = S[j].P_2 \cup \{A[i]\}$
25:             end if
26:         end for
27:     end for
28:     return $S$
29: end function
```

As an example, suppose that the size of the set is $N = 350,000,000$ and the partition is $P = 8$, where the optimal iteration within a thread is $O = 7,656,250,021,875,000$. In this case, after the calculation, the indices are shown in Table 2.

At the last calculation, the index is below the expected value; therefore, it is necessary to correct it. To overcome this, it may be a solution to calculate the indices for $P - 1$ partitions instead, and then take the N as the last index of an additional partition.

4.5 Perfect partition

A perfect division is when every thread takes the same number of iterations, which matches the optimal number of calculated iterations. We have done practical experiences with all set sizes between 1 and 2147483647. It can be said that certain sets can be divided into two equal parts by iterations (Table 3). It is not possible to do the same with larger number of partitions (where the number of partitions is a power of two).

4.6 Maximum meaningful size of the partition

Maximum meaningful partition stands for the P value, where at least one iteration of the external loop is executed for each partition (the $I_{lower} < I_{upper}$ condition is met). This condition prevents us from creating threads that will not contain iterations. This condition is usually violated when either the set we want to process is too small, or when the level of parallelization is too high.

For example, take the previous example where $N = 8$ and the value of P should be 6. In this case, the computed indices will be: $0-1, 1-2, \mathbf{2} - \mathbf{2}, 2-4, 4-5, 5-8$, where the third thread indices violate the condition: because it has a I_{lower} and I_{upper} indices value of 2, this thread will do nothing.

The maximum value of P has not yet been determined but can be restricted between a *lower* limit and an *upper* limit. The *lower* limit is obtained by taking $\frac{N+1}{2}$, because we can create partitions at least half of the element number. This equation can be derived from the following relation-

Table 2: Index distribution and deviation when $N = 350,000,000$ and $P = 8$

Index lower value	**Index upper value**	**Number of iterations**	**Deviation**
0	22 604 979	7 656 250 123 507 270	0,0000013274%
22 604 979	46 891 109	7 656 249 998 313 340	-0,0000003077%
46 891 109	73 300 705	7 656 249 988 081 220	-0,0000004414%
73 300 705	102 512 627	7 656 250 044 133 910	0,0000002907%
102 512 627	135 669 648	7 656 250 019 577 120	-0,0000000300%
135 669 648	175 000 000	7 656 249 913 887 130	-0,0000014105%
175 000 000	226 256 314	7 656 250 113 194 860	0,0000011927%
226 256 314	349 999 995	7 656 249 974 305 130	-0,0000006213%

Table 3: Size of datasets where the perfect partitioning with partition size of $N = 2$ is possible

3	20	119	696	4 059	23 660
137 903	803 760	4 684 659	27 304 196	159 140 519	213 748 912
236 368 449	290 976 842	345 585 235	477 421 558	532 029 951	554 649 488
609 257 881	663 866 274	741 094 204	863 561 208	882 299 846	904 919 383
927 538 920	950 158 457	959 527 776	982 147 313	1 004 766 850	1 059 375 243
1 081 994 780	1 091 364 099	1 100 733 418	1 127 233 854	1 136 603 173	1 145 972 492
1 155 341 811	1 191 211 566	1 200 580 885	1 245 819 959	1 255 189 278	1 268 439 496
1 300 428 352	1 332 417 208	1 345 667 426	1 400 275 819	1 422 895 356	1 432 264 675
1 454 884 212	1 477 503 749	1 518 861 924	1 522 742 823	1 528 231 243	1 541 481 461
1 564 100 998	1 596 089 854	1 618 709 391	1 631 959 609	1 637 448 029	1 650 698 247
1 673 317 784	1 682 687 103	1 718 556 858	1 741 176 395	1 769 284 352	1 795 784 788
1 818 404 325	1 850 393 181	1 859 762 500	1 869 131 819	1 882 382 037	1 886 262 936
1 891 751 356	1 927 621 111	1 946 359 749	1 959 609 967	1 968 979 286	1 982 229 504
2 000 968 142	2 014 218 360	2 036 837 897	2 059 457 434	2 068 826 753	2 078 196 072
2 091 446 290	2 104 696 508	2 132 804 465	2 146 054 683		

ship:

$$\begin{aligned} N &\leq \frac{N(N+1)}{2P} \\ 1 &\leq \frac{N+1}{2P} \\ P &\leq \frac{N+1}{2} \end{aligned} \tag{12}$$

The upper limit has not been proved yet, but we get the following therorem about it after expanding the lower limit as is shown in Equation 13.

$$P < \frac{N+1}{2 - \frac{2}{\sqrt{N}}} - 1 \tag{13}$$

Based on the *lower* and *upper* limit equations, the first 50 partition values are shown in Table 4 and Figure 4. As visible, the upper limit follows the real maximum P value (expect some special cases), until then the lower limit is slowly getting further away.

In summary, if the P value is chosen less than or equal to the *lower* limit, the partitioning will be certainly correct, and each thread will contain processable iterations. In the case of higher values, the difference between the *lower* limit and the optimal value increasing; therefore, it is advisable to use the *upper* limit -1 to gain high level of parallelism.

5 Evaluation

This section presents the testing methods for the runtime of the different versions of nested loops algorithms. The inputs were generated by the following parameters (resulting in a total of 96 datasets).

1. Size of the dataset (N) can be 10, 100, 1 000, 10 000, 20 000, 50 000, 100 000, 200 000.

2. Minimum and maximum size of the elements in the dataset (E_l) can be 10-100, 1 000-5 000, 10 000-15 000.

3. Probability of duplicates in the dataset (β) can be 0, 0.33, 0.5, 0.8.

A C# application was developed for testing which runs each measurement 10 times using each algorithm. During the measurements, the best and worst results are discarded and the final result was the average of the remaining values.

The following configuration was used for testing:

- CPU: Intel(R) Core(TM) i5-7300 (2 physical cores, 4 logical cores)

- RAM: 16 GB

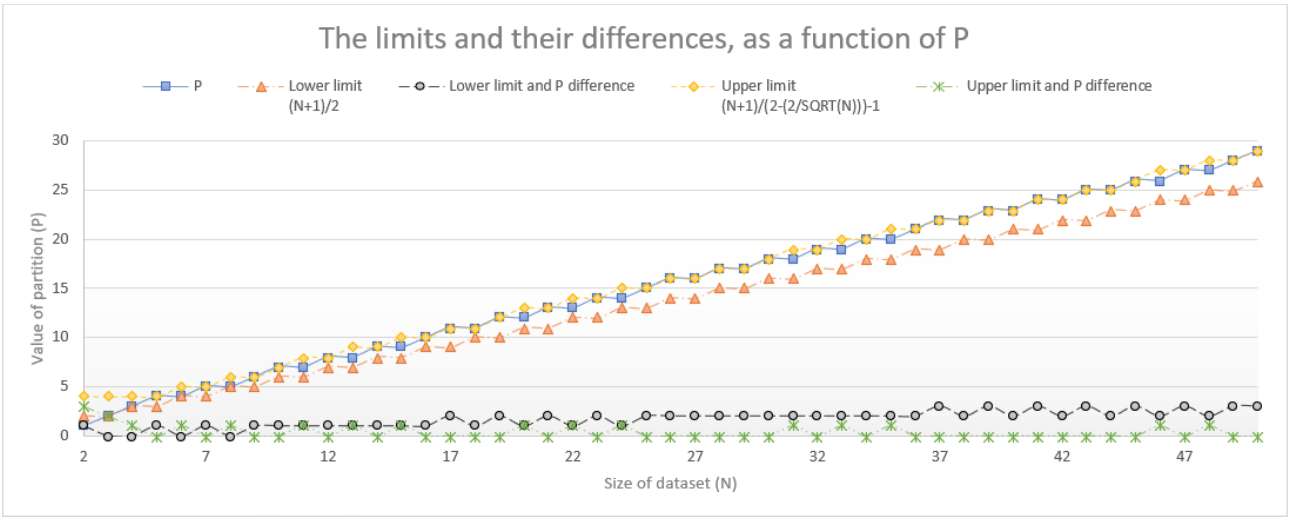

Figure 4: The evolution of limits and their differences, as a function of P

– HDD: Samsung MZVLB512HAJQ-000L7, 475,48 GB

– FileSystem: NTFS v3.1, cluster size: 4096 bytes

– Used level of parallelizm (P): 4

The results of the measurements show that the amount of duplication of elements within the set (β) and the "length" of the elements (E_l) do not effect the runtime significantly. The runtime difference between the four algorithms presented is determined by the differences in the dataset sizes. This observation is illustrated by the results of the measurements in Figure 5.

The diagrams can be seen that the slopes of the functions are nearly the same. For example, comparing the execution time of the Naive Serial algorithm in the second (E_l : $1000 - 5000$) and the third (E_l : $10000 - 15000$) diagram, can be seen, that even though in the second case the elements in the set are twice as long, runtime only increased depending on the number of elements. For example, for 50 000 elements, when the length of the elements was increased, the difference was only 100 927 317 ticks ($\approx$ 0.010092732 seconds), while for doubling the size of the set, the execution time increased by an average of 3 083 536 094 ticks ($\approx$ 0.308353609 seconds). Of course, this relationship also applies to other measurements.

It is also determined by the measurements (as visible in Figure 6) that it is worth using the parallelization in the case of more than 10,000 elements.

In cases where the size of the dataset does not reach that number, serial processing will result in faster execution time, regardless of the length of the elements.

In Figure 7, the two versions of parallel algorithms are compared and the differences between runtimes are highlighted.

As can be seen in cases where the dataset size exceeds the minimum number of elements required for parallelization ($N \geq 10\ 000$), the improved version of the parallel algorithm always results in faster execution time (between $15 - 40$ percentage) than the naive version.

6 Conclusion

To objective of this paper was to speed-up the calculation of a given R operation using all possible pairs of a set as operands. Several sequential and parallel procedures for this purpose have been presented and analysed.

The main contribution of this paper is a novel nested loop parallelization method based on the number of items. This method is able to give an efficient partitioning of the problem based on the number of items and the number of available processors.

Some further analysis was also presented about the numerical error of the method, perfect partitioning and the maximum meaningful size of the partition.

The evaluation section shows that the presented method is very effective and is able to give almost optimal results is all cases.

Acknowledgement

This work is supported by the Doctoral School of Applied Informatics and Applied Mathematics, Óbuda University.

References

[1] Volodymyr Beletskyy and Maciej Poliwoda Parallelizing. Perfectly nested loops with non-uniform dependences. 2002.

[2] H. Blinn. How to solve a quadratic equation? *IEEE Computer Graphics and Applications (Volume: 25 , Issue: 6)*, 2005.

[3] Kiran Bondalapati. Parallelizing dsp nested loops on reconfigurable architectures using data context switching. *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001. `doi:https://doi.org/10.1109/dac.2001.935519`.

[4] Beata Bylina and Jarosław Bylina. Strategies of parallelizing nested loops on the multicore architectures on the example of the wz factorization for the dense matrices. *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 629–639, 2015. `doi:https://doi.org/10.15439/2015f354`.

[5] Beata Bylina and Jarosław Bylina. Parallelizing nested loops on the intel xeon phi on the example of the dense wz factorization. *Proceedings of the Federated Conference on Computer Science and Information Systems*, 8:655–664, 2016. `doi:https://doi.org/10.15439/2016f436`.

[6] Benjamin James Gaska, Neha Jothi, Mahdi Soltan Mohammadi, and Kat Volk. Handling nested parallelism and extreme load imbalance in an orbital analysis code. *arXiv:1707.09668*, 2017.

[7] Adrian Jackson and Orestis Agathokleous. Dynamic loop parallelisation. *arXiv:1205.2367*, 2012.

[8] Nedal Kafri and Jawad Abu Sbeih. Simple optimal partitioning approch to perfect tringular iteration space. *Proceedings of the 2008 High Performance, Computing & Simulation Conference ©ECMS, Waleed W. Smari (Ed.), ISBN: 978-0-9553018-7-2 / ISBN: 978-0-9553018-6-5 (CD)*, pages 124–131, 2008.

[9] Arun Kejariwal, Paolo D'Alberto, Alexandru Nicolau, and Constantine D. Polychronopoulos. A geometric approach for partitioning n-dimensional nonrectangular iteration spaces. In *Languages and Compilers for High Performance Computing*, pages 102–116, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[10] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 235–244, 2007. `doi:https://doi.org/10.1145/1250734.1250761`.

[11] Christian Lengauer. Loop parallelization in the polytope model. *CONCUR'93*, pages 398–416, 1993.

[12] Junyi Liu, John Wickerson, and George A. Constantinides. Loop splitting for efficient pipelining in high-level synthesis. *Proceedings of the Federated Conference on Computer Science and Information Systems*, 2016. `doi:https://doi.org/10.1109/FCCM.2016.27`.

[13] Matthieu Kuhn Philippe Clauss, Ervin Altıntas. Automatic collapsing of non-rectangular loops. *Parallel and Distributed Processing Symposium (IPDPS), Orlando, United States*, pages 778–787, 2017.

[14] S. D. Polychronopoulos. Loop coalescing: a compiler transformation for parallel machines. *University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Technical Report CSRD-635*, 1987.

[15] J. Ramanujam. Optimal software pipelining of nested loops. In *Proceedings of 8th International Parallel Processing Symposium*, pages 335–342, 1994. `doi:https://doi.org/10.1109/IPPS.1994.288280`.

[16] Rizos Sakellariou. A compile-time partitioning strategy for non-rectangular loop nests. *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97 (Geneva), IEEE Computer Society Press*, pages 633–637, 1997.

[17] Thomas Schmidt and Jens Stoye. Quadratic time algorithms for finding common intervals in two and more sequences. *Proc of CPM 2004 LNCS*, 3109:347–358, 2004. `doi:https://doi.org/10.1007/978-3-540-27801-6_26`.

[18] T. H. Tzen and L. M. Ni. Dependence uniformization: a loop parallelization technique. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):547–558, 1993. `doi:https://doi.org/10.1109/71.224217`.

[19] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, 1991. `doi:https://doi.org/10.1109/71.97902`.

Table 4: The evolution of limits and their differences, as a function of P

N	P	$Lower\ limit$		$Upper\ limit$		N	P	$Lower\ limit$		$Upper\ limit$	
		$Value$	$Diff.$	$Value$	$Diff.$			$Value$	$Diff.$	$Value$	$Diff.$
2	1	2	1	4	3	27	16	14	2	16	0
3	2	2	0	4	2	28	17	15	2	17	0
4	3	3	0	4	1	29	17	15	2	17	0
5	4	3	1	4	0	30	18	16	2	18	0
6	4	4	0	5	1	31	18	16	2	19	1
7	5	4	1	5	0	32	19	17	2	19	0
8	5	5	0	6	1	33	19	17	2	20	1
9	6	5	1	6	0	34	20	18	2	20	0
10	7	6	1	7	0	35	20	18	2	21	1
11	7	6	1	8	1	36	21	19	2	21	0
12	8	7	1	8	0	37	22	19	3	22	0
13	8	7	1	9	1	38	22	20	2	22	0
14	9	8	1	9	0	39	23	20	3	23	0
15	9	8	1	10	1	40	23	21	2	23	0
16	10	9	1	10	0	41	24	21	3	24	0
17	11	9	2	11	0	42	24	22	2	24	0
18	11	10	1	11	0	43	25	22	3	25	0
19	12	10	2	12	0	44	25	23	2	25	0
20	12	11	1	13	1	45	26	23	3	26	0
21	13	11	2	13	0	46	26	24	2	27	1
22	13	12	1	14	1	47	27	24	3	27	0
23	14	12	2	14	0	48	27	25	2	28	1
24	14	13	1	15	1	49	28	25	3	28	0
25	15	13	2	15	0	50	29	26	3	29	0
26	16	14	2	16	0	51	29	26	3	29	0

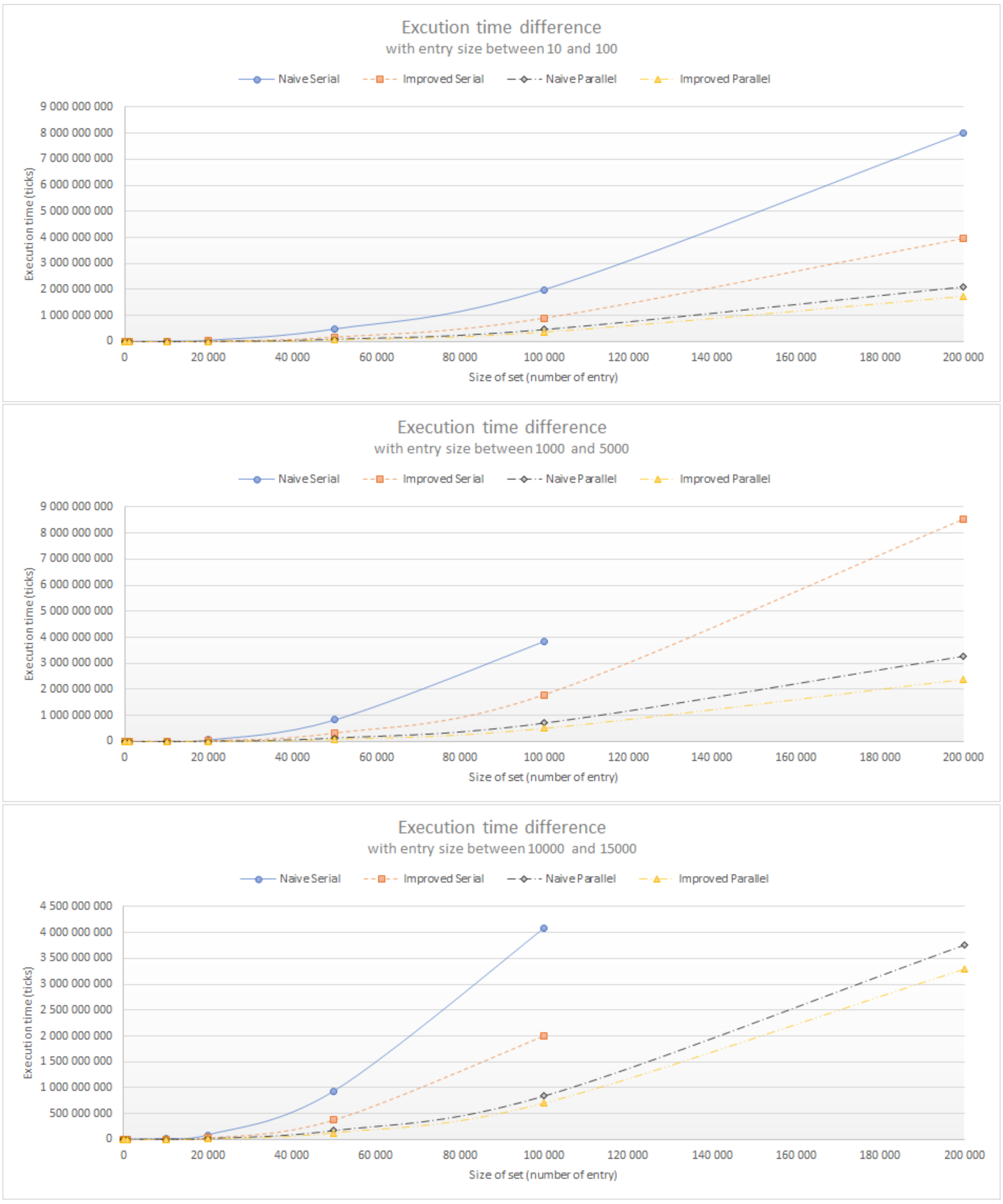

Figure 5: Execution time difference

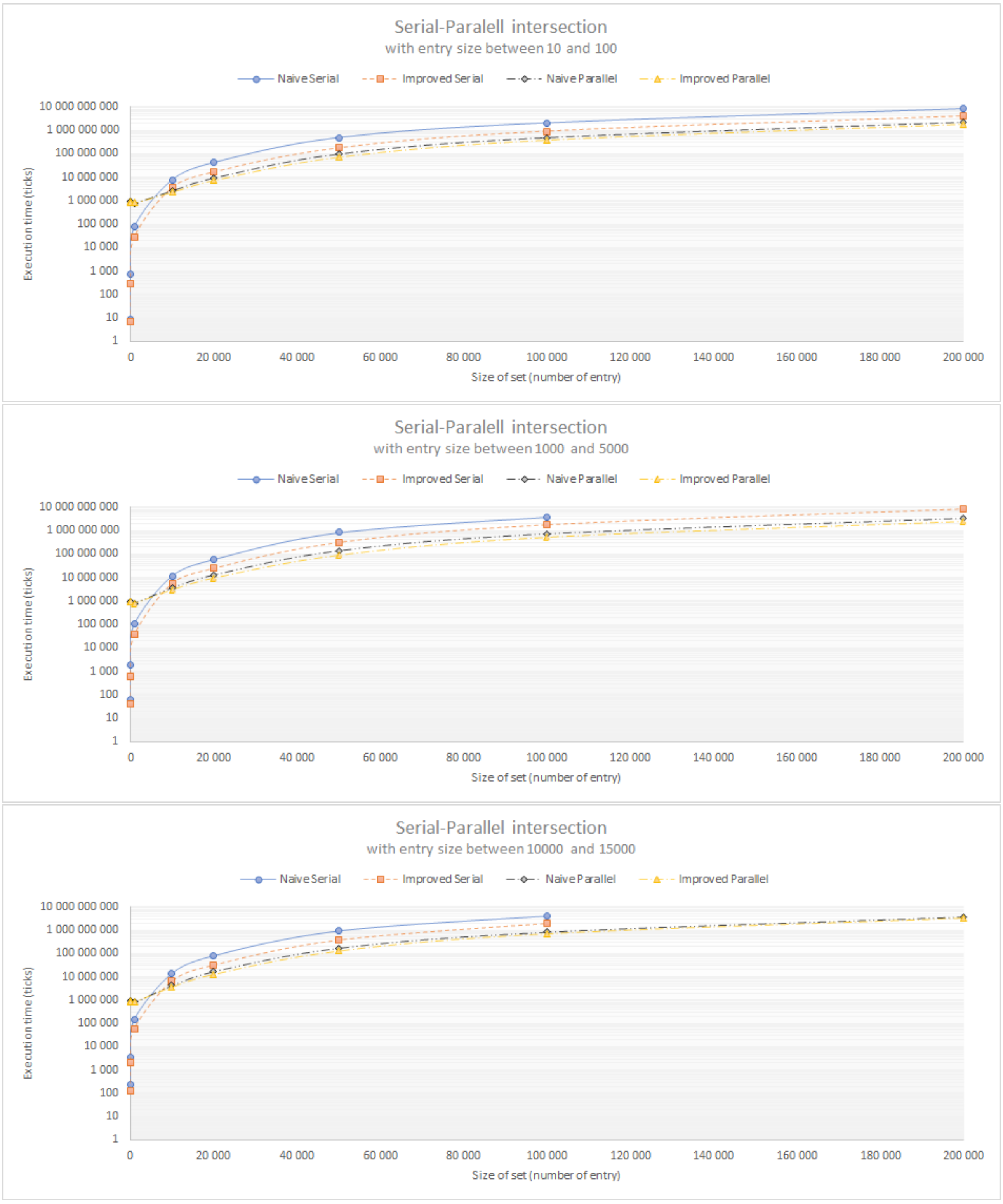

Figure 6: Serial-Parallel intersection

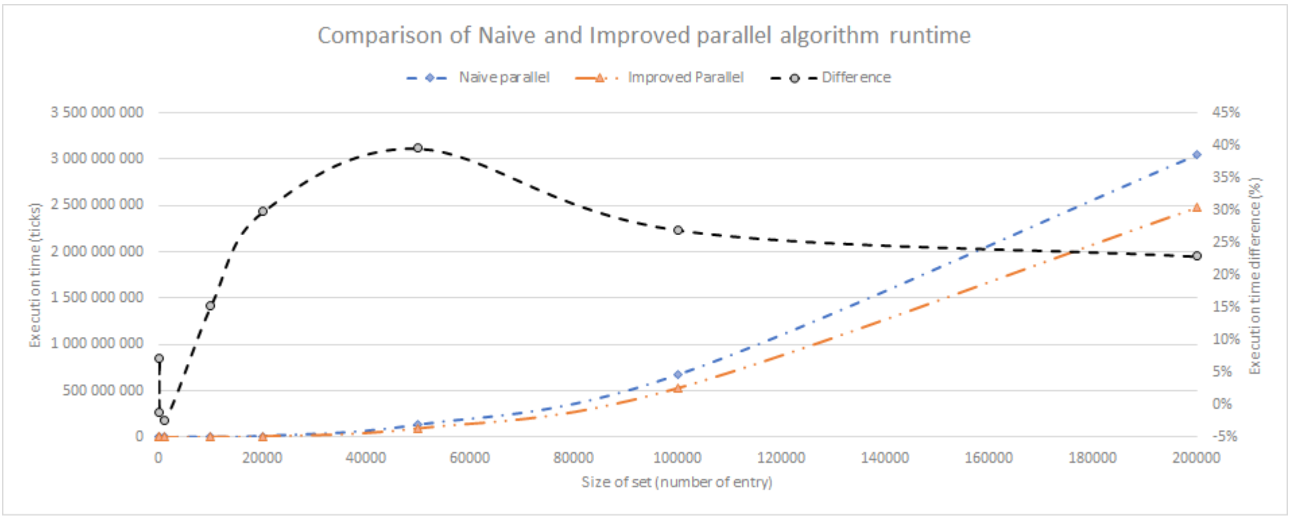

Figure 7: Comparison of Naive and Improved parallel algorithm execution runtime