Future Generation Computer Systems 159 (2024) 423–431

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Parallel intersection counting on shared-memory multiprocessors and GPUs

Moreno Marzolla [a,c,*], Giovanni Birolo [b], Gabriele D'Angelo [a,c], Piero Fariselli [b]

[a] Dipartimento di Informatica–Scienza e Ingegneria (DISI), Università di Bologna, Mura Anteo Zamboni, 7, Bologna, I-40126, Italy
[b] Dipartimento di Scienze Mediche, Università di Torino, Corso Dogliotti, 14, Torino, I-10124, Italy
[c] Center for Inter-Department Industrial Research ICT, Università di Bologna, Bologna, I-40126, Italy

ARTICLE INFO

Dataset link: https://github.com/mmarzolla/parallel-intersections

Keywords:
Intersection counting
Parallel algorithms
GPU programming
Shared-memory algorithm
Bioinformatics

ABSTRACT

Computing intersections among sets of one-dimensional intervals is an ubiquitous problem in computational geometry with important applications in bioinformatics, where the size of typical inputs is large and it is therefore important to use efficient algorithms. In this paper we propose a parallel algorithm for the 1D intersection-counting problem, that is, the problem of counting the number of intersections between each interval in a given set A and every interval in a set B. Our algorithm is suitable for shared-memory architectures (e.g., multicore CPUs) and GPUs. The algorithm is work-efficient because it performs the same amount of work as the best serial algorithm for this kind of problem. Our algorithm has been implemented in C++ using the Thrust parallel algorithms library, enabling the generation of optimized programs for multicore CPUs and GPUs from the same source code. The performance of our algorithm is evaluated on synthetic and real datasets, showing good scalability on different generations of hardware.

1. Introduction

Counting or enumerating intersections among sets of one-dimensional intervals is a recurrent problem in many application domains such as computer graphics [1, Ch. 42], distributed simulation [2], and bioinformatics [3]. Given two real values $left \le right$, the closed interval $a = [left, right]$ denotes the set of values between $left$ and $right$, inclusive. We say that two intervals a, b overlap iff their intersection is not empty, i.e., $a \cap b \neq \emptyset$.

Different types of interval intersection problems have been considered in the literature. The *intersection enumeration problem* consists on enumerating all pairs of overlapping intervals (a, b), $a \in A$, $b \in B$. Fig. 1 shows an example of two sets of closed intervals A, B, the intersections being:

$$\{(a_0, b_0), (a_0, b_1), (a_1, b_1), (a_1, b_2), (a_2, b_3), (a_2, b_4), (a_2, b_5)\}$$

The *intersection counting problem* consists on computing the number of intervals in B that overlap with each interval $a \in A$. More formally, given two sets of intervals A, B with $|A| = n$, $|B| = m$, the goal is to compute a function $c : A \to \mathbb{N}$ that maps each interval $a \in A$ to the number of intervals $b \in B$ that overlap with a:

$$c(a) = \left|\{b \in B \mid a \cap b \neq \emptyset\}\right| \quad (1)$$

For the example of Fig. 1, we have $c(a_0) = 2$, $c(a_1) = 2$, $c(a_2) = 3$.

Finally, the *all-intersections counting problem* consists on computing the total number of intersections between two sets of intervals A, B. The total number of intersections is the summation of the number of intersections per interval, i.e., $\sum_{a \in A} c(a)$. For the scenario shown in Fig. 1, there is a total of 7 overlaps.

In this paper we consider the one-dimensional intersection counting problem, which enables us to also solve the all-intersections counting problem as a special case. Our interest stems from their important applications in bioinformatics: finding overlaps of genetic features, computing coverage of aligned reads in DNA sequencing, and expression quantification in RNA sequencing are only some examples of analyses that need some intersection-finding algorithms at their core. The sets of intervals involved are often quite large; for instance, the number of exons (a commonly used genetic feature) in the human genome is around 180,000. DNA sequencing of a whole human genome can yield several billions of reads. Moreover, while the human genome has three billion bases and is quite long and complex, longer genomes abound in nature. As a consequence, efficiency becomes a major concern.

Most genomic analyses are performed on workstations or High-Performance Computing (HPC) infrastructures normally equipped with multicore processors and GPUs. Many sequential tasks can be run in parallel by splitting the data into independent chunks, for instance, by

* Correspondence to: Department of Computer Science and Engineering (DISI), University of Bologna, Cesena Campus, via dell'Università 50, I-47521 Cesena, Italy.
E-mail addresses: moreno.marzolla@unibo.it (M. Marzolla), giovanni.birolo@unito.it (G. Birolo), g.dangelo@unibo.it (G. D'Angelo), piero.fariselli@unito.it (P. Fariselli).

https://doi.org/10.1016/j.future.2024.05.039
Received 16 October 2023; Received in revised form 15 February 2024; Accepted 17 May 2024
Available online 20 May 2024

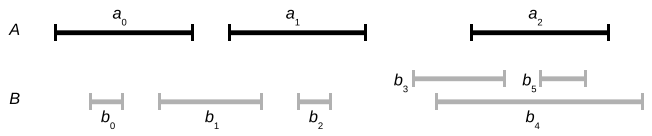

Fig. 1. Example of overlapping sets of intervals.

individual or by chromosome. However, this approach is not always optimal or even feasible, so having inherently parallel and efficient algorithms is a big advantage. Also, GPUs, when available, can provide substantial boosts in computing speed for software that can use them. For these reasons, we developed an inherently parallel intersection counting algorithm implemented for both multicore and GPU architectures.

Counting intersections among two sets of intervals plays an important role in supporting range queries in temporal databases [4]. Temporal databases store information that are associated to some instant in time or time interval; queries involve the computation of overlaps among sets of time intervals, which could be answered efficiently using the algorithm described in the following.

In this paper we propose a work-efficient, parallel algorithm for the one-dimensional intersection-counting problem that is suitable for shared-memory architectures and GPUs. Our algorithm is based on a novel parallel implementation of the sweep-line technique that represents the state of the art sequential solution. The sweep-line technique involves the traversal of the sorted list of endpoints, keeping track of which intervals are active at each step (details will be provided in Section 3). Unfortunately, the sweep-line algorithm does not lend itself to a parallel realization due to its intrinsically serial nature. We address this problem by casting the core of the serial algorithm into a sequence of parallel scans with an ad-hoc associative operator. The parallel algorithm is work-efficient, since it performs the same total amount of work as the sequential algorithm. We implemented the parallel intersection-counting algorithm using the Thrust library that targets both multicore CPU and GPUs. This allows the same code base to be used to generate efficient parallel code for multicore CPU and GPUs.

This paper is organized as follows. In Section 2 we review the relevant scientific literature. In Section 3 we describe a parallel intersection-counting algorithm based on the sweep-line technique and show that the algorithm is work-efficient. Section 4 describes performance results of CPU and GPU-based implementations, on real and simulated datasets; the source code of our implementation is freely available online at the URL provided at the end of this paper. Finally, conclusions are discussed in Section 5.

2. Related works

Intersection counting and enumeration problems have been traditionally studied in the computational geometry research area. The first efficient solution was proposed by [5] who developed a one-pass solution to the intersection-enumeration problem that requires time $O((n+m)\log(n+m)+K)$, where K is the number of intersections. The method is based on the *sweep-line technique* [6]: the idea is to sort the list of endpoints of the intervals in non-decreasing order, and scan the sorted list to keep track of which intervals are still open at each point.

Recent developments in this area have been carried out in the context of bioinformatics. Indeed, the fast development of the field of genetics and the many advances that it spurred in medicine, agriculture and many other disciplines, is largely dependent on the ability of "reading" DNA, the molecules that carry the genetic information in all living beings. Thanks to its molecular structure, DNA (and similar molecules such as RNA) can encode arbitrary strings of a four-element alphabet, whose "letters" are called "bases". In humans, for instance, the genetic information that defines our species and each individual is divided into 23 DNA molecules called "chromosomes" for a total of three billion bases. It is therefore, not surprising that the need to process large datasets has driven the development of parallel algorithms for counting and enumerating intersections.

BEDOPS [7] is a suite of tools for computing overlap and proximity relations between genomic datasets. BEDOPS uses the sweep-line technique to compute the genomic space that is common to two datasets, from which it is possible to count the number of intersections. Although some of BEDOPS components support parallel processing, the program that computes intersections does not yet.

BEDTK [8] is a software package that is specialized for solving the intersection enumeration problem. It uses the *interval tree* data structure, a balanced search tree that stores a set of intervals sorted on the left endpoint. The interval tree is optimized for listing the intersections rather than just counting them. A variation of the interval tree, called *segment tree forest*, is proposed in [9] for finding overlapping intervals across n sets. The authors developed a tool called Joint Overlap Analysis (JOA) that can compute intersections efficiently by traversing the forest in parallel.

Another tree-based data structure, called *two-dimensional range tree*, has been proposed by [10] for querying multiple types of predicates on large interval sets. All types of relationships defined by Allen's interval algebra [11] are transformed into two-dimensional range queries that are answered efficiently through 2D range trees.

All approaches above deal with the intersection-enumeration problem. Enumerating intersections is intuitively "more difficult" than counting them, since the computational cost of any enumeration algorithm must grow at least linearly with the output size K, the number of intersections to be reported. Since the maximum number of intersections is $O(nm)$, every intersection-enumeration algorithm has a worst-case cost that is proportional to the product of the sizes of the two sets of intervals. In applications where intersection counting suffices, more efficient solutions are desirable.

Binary Interval Search (BITS) [3] has been proposed as an efficient algorithm for counting intersections, although it can also report the list of overlaps. BITS operates by sorting the left and right endpoints of B into two separate lists, and performs two binary searches to count the number of intersections between a test interval $a_i \in A$ and B. The number of intersections between all intervals in two sets A and B can therefore be computed using $2n$ binary searches on B. The algorithm naturally leads to a parallel version, since the $2n$ searches are independent and can be done in parallel. In [3] it is claimed that a purely sweep-based parallel intersection-counting algorithm would be inefficient, since it would require either to partition A among the execution units, leading to load imbalance, or to frequent information exchange between threads.

In this paper we disprove the claim above by showing a parallel sweep-based algorithm that is optimal (i.e., work-efficient), as it performs the same amount of work of the sequential algorithm. We first describe a serial intersection-counting algorithm, and then show that the steps it performs are instances of known parallel programming patterns that naturally lead to an efficient parallel algorithm.

3. Parallel intersection counting

In this section we describe a novel parallel intersection-counting algorithm based on the sweep-line technique. For the sake of readability, we break the description into two steps: the parallel algorithm is first introduced in sequential form, and then we show how the key steps can be executed on a shared-memory parallel architecture.

Given two sets A, B of 1D intervals of cardinality $|A| = n$, $|B| = m$, we follow the idea from [5] of sorting the list T of the endpoints of A and B in non-decreasing order, and scan the sorted list to keep track of which intervals are active (i.e., still open) at each point. We say that endpoint x precedes y if either (i) $x < y$, or (ii) $x = y$ and x

is a left endpoint and y is a right endpoint. This criterion will be used to compare and sort endpoints in the rest of this paper. We also denote with $a.left$ and $a.right$ the left and right endpoints of interval a, respectively.

Let $\mathcal{L} = [\mathcal{L}_0, \ldots, \mathcal{L}_{n-1}]$ and $\mathcal{R} = [\mathcal{R}_0, \ldots, \mathcal{R}_{n-1}]$ be two arrays of sets of intervals, i.e., each $\mathcal{R}_i$ and $\mathcal{L}_i$ is a set of intervals where:

$$\begin{aligned} \mathcal{L}_i &:= \text{set of intervals in } B \text{ whose } \textit{left} \text{ endpoint appears before} \\ &\quad\ \text{the } \textit{right} \text{ endpoint of } a_i \text{ in } T \\ \mathcal{R}_i &:= \text{set of intervals in } B \text{ whose } \textit{right} \text{ endpoint appears} \\ &\quad\ \text{before the } \textit{left} \text{ endpoint of } a_i \text{ in } T \end{aligned}$$

Then, the set difference $\mathcal{L}_i \setminus \mathcal{R}_i$ is by definition the set of intervals in B that intersect a_i. For example, considering Fig. 1 we have:

$$\begin{aligned} \mathcal{L}_0 &= \{b_0, b_1\} & \mathcal{R}_0 &= \emptyset \\ \mathcal{L}_1 &= \{b_0, b_1, b_2\} & \mathcal{R}_1 &= \{b_0\} \\ \mathcal{L}_2 &= \{b_0, b_1, b_2, b_3, b_4, b_5\} & \mathcal{R}_2 &= \{b_0, b_1, b_2\} \end{aligned}$$

from which we get that the intervals in B that intersect, e.g., a_1 are $\mathcal{R}_1 \setminus \mathcal{L}_1 = \{b_0, b_1, b_2\} \setminus \{b_0\} = \{b_1, b_2\}$.

Since we are only interested in the number of intervals overlapping each a_i, we can replace the sets $\mathcal{L}_i$ and $\mathcal{R}_i$ with the scalar values $L_i = |\mathcal{L}_i|$ and $R_i = |\mathcal{R}_i|$, respectively. L_i is the number of intervals in B whose left endpoint appears before the right endpoint of a_i, while R_i is the number of intervals in B whose right endpoint appears before the right endpoint of a_i. In the case of Fig. 1 we have:

$$\begin{aligned} L_0 &= 2 & R_0 &= 0 \\ L_1 &= 3 & R_1 &= 1 \\ L_2 &= 6 & R_2 &= 3 \end{aligned}$$

from which we see that the number of intervals in B that overlap $a_i \in A$ is

$$c(i) = L_i - R_i \tag{2}$$

The observations above lead to the sequential Algorithm 1. The endpoints are stored in the array T of length $(n + m)$, where each interval x has two attributes $x.left$ and $x.right$ of type real representing the left and right endpoint of x, respectively. After T is sorted in non-decreasing order, the counts of left and right endpoints can be computed by the loops on lines 1.11 and 1.20.

The loop 1.11 initializes two arrays $nleft[i]$ and $nright[i]$ that satisfy the following properties:

$$nleft[i] := \text{Number of } left \text{ endpoints of } B \text{ that appear in } T[0..i] \tag{3}$$

$$nright[i] := \text{Number of } right \text{ endpoints of } B \text{ that appear in } T[0..i] \tag{4}$$

Note that $nleft[i]$ and $nright[i]$ are not the values L_i and R_i that are required to compute the counts; instead, $nleft$ and $nright$ contain a permutation of the arrays L and R. To compute L_i and R_i efficiently (i.e., in constant time) we define two additional arrays $left\_idx$ and $right\_idx$ of length $(n+m)$ that map positions (indexes) in T to positions in $nleft$ and $nright$. Specifically, if an endpoint $t_i \in T$ is the left (resp. right) endpoint of interval $a_k \in A$, then $left\_idx[k] = i$ (resp. $right\_idx[k] = i$). Therefore, $left\_idx$ and $right\_idx$ map the index of an interval to the position (index) of the left and right endpoints of that interval in T. At the end of the loop 1.20, the following identities hold:

$$nleft[right\_idx[i]] = L_i nright[left\_idx[i]] = R_i \tag{5}$$

that, combined with (2), yields

$$c(i) = nleft[right\_idx[i]] - nright[left\_idx[i]] \tag{6}$$

(line 1.28).

Fig. 2 ② and ③ shows the behavior of Algorithm 1 on sets A and B from Fig. 1 (the portion labeled ① will be explained later on).

Algorithm 1 INTERSECTION-COUNTING(A, B)

```
1:  n ← |A|, m ← |B|
2:  T ← ∅
3:  nleft ← 0^{n+m}, nright ← 0^{n+m}          ▷ Arrays of length n + m
4:  left_idx ← 0^n, right_idx ← 0^n            ▷ Arrays of length n
    // Initialization
5:  for all intervals x ∈ A ∪ B do
6:      Insert x.left and x.right in T
7:  end for
    // Sort the endpoints in non-decreasing order
8:  SortT
    // Initialize the nleft[] and nright[] arrays
9:  nl ← 0                                     ▷ Number of left endpoints of B seen so far
10: nr ← 0                                     ▷ Number of right endpoints of B seen so far
11: for all endpoints t_i ∈ T in non-decreasing order do
12:     if t_i is the left endpoint of an interval b ∈ B then
13:         nl ← nl + 1
14:     else if t_i is the right endpoint of an interval b ∈ B then
15:         nr ← nr + 1
16:     end if
17:     nleft[i] ← nl
18:     nright[i] ← nr
19: end for
    // Build index arrays
20: for all endpoints t_i ∈ T in non-decreasing order do
21:     if t_i is the left endpoint of a_k ∈ A then
22:         left_idx[k] ← i
23:     else if t_i is the right endpoint of a_k ∈ A then
24:         right_idx[k] ← i
25:     end if
26: end for
    // Compute final result
27: for all a_i ∈ A do
28:     c(i) ← nleft[right_idx[i]] − nright[left_idx[i]]
29: end for
```

Fig. 2 ② shows the content of the $nleft$ and $nright$ arrays at the end of the loop 1.11; $nleft[i]$ and $nright[i]$ are the number of left and right endpoints in B, respectively, that appear in $T[0..i]$. Therefore, $nleft[12] = 5$ and $nright[12] = 3$ since in $T[0..12]$ there are five left endpoints in B, namely, those of b_0, b_1, b_2, b_3, b_4, and three right endpoints in B, namely, those of b_0, b_1, b_2. Note that one must always have $nright[i] \le nleft[i]$ since at any given point there cannot be more right endpoints than left endpoints of intervals in B. Fig. 2 ③ shows the values of $left\_idx$ and $right\_idx$ computed at the end of the loop 1.20. $right\_idx[k]$ is the position (index) of the right endpoint of interval $a_k \in A$; similarly, $left\_idx[k]$ is the position of the left endpoint of a_k. We see that $left\_idx[1] = 5$ and $right\_idx[1] = 9$, since the left and right endpoints of a_1 occupy the slots $T[left\_idx[1]] = T[5]$ and $T[right\_idx[1]] = T[9]$ in the sorted array T.

Algorithm 1 uses an array T of length $(n + m)$, $nleft$ and $nright$ of length $(n + m)$, and $left\_idx$ and $right\_idx$ of length n. Therefore, the space requirement is $\Theta(n + m)$. The execution time is dominated by the time required to sort T, which is $O((n + m)\log(n + m))$ using efficient comparison-based sorting,[1]. All other phases require linear time in either $(n + m)$ (initialization of T, loops 1.11, 1.20), or n (loop 1.27). Therefore, the total execution time of Algorithm 1 is $O((n + m)\log(n + m))$.

[1] Should the endpoints be integers, as assumed in bioinformatics applications, sorting could be done in linear time; however, for the sake of generality we assume comparison-based sorting so that the proposed algorithm can be used with endpoints with real values.

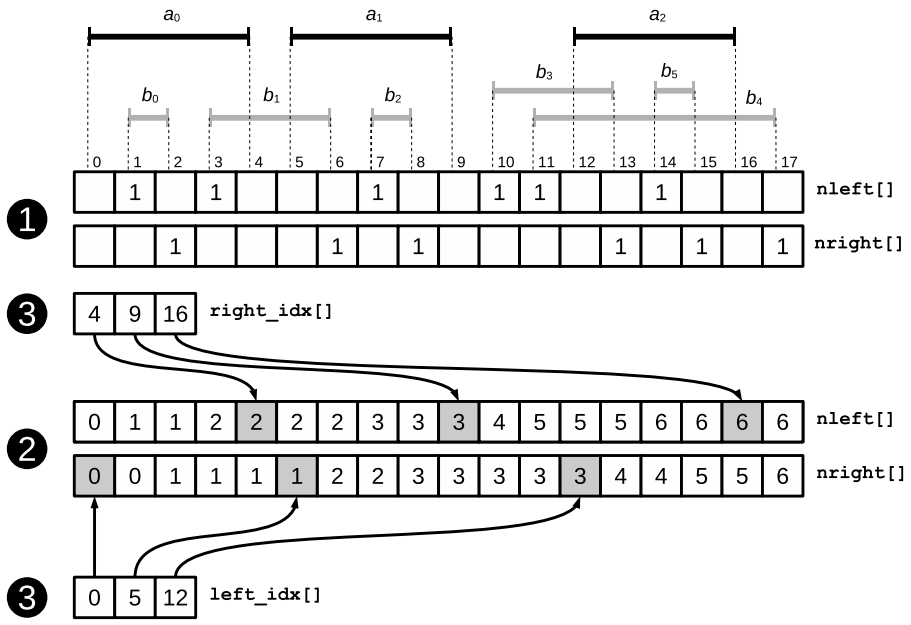

Fig. 2. Example of execution the intersection counting algorithm. ①: $nleft[i] = 1$ (resp. $nright[i] = 1$) if t_i is the left (resp. right) endpoint of some interval in B. ② The *left* and *right* arrays after application of the inclusive-scan operation. ③ *left_idx* and *right_idx* point to the positions of the left and right endpoints of the intervals in A, so that Eq. (5) holds. At the end, the number of intervals in B that overlap with interval $a_i \in A$ is $nleft[right\_idx[i]] - nright[left\_idx[i]]$.

Algorithm 1 is the most efficient known solution of the intersection counting problem. We now illustrate how it can be parallelized on a shared-memory architecture or GPU. In doing so, our goal is to make the parallel version *work-efficient*, meaning that the total amount of work performed by all execution units shall be equal to the work performed by the serial version, namely, $O((n+m)\log(n+m))$. As will be shown below, this goal has been achieved up to an additional overhead $O(\log\log(n+m))$ which is negligible in practice.

We observe that some of the steps are either embarrassingly parallel, or rely on known parallel algorithms (e.g., sorting). Unfortunately, the loop starting at line 1.11 cannot be parallelized due to *loop-carried dependencies*: indeed, the loop body depends on the values of $nleft$ and $nright$ that might be updated during previous iterations. However, it turns out that the loop can be expressed as a *prefix computation*.

A general (inclusive) prefix computation takes as input an array $x_0, \ldots, x_{n-1}$ and an associative operator $\oplus$, and produces another array $y_0, \ldots, y_{n-1}$ such that $y_i = x_0 \oplus \cdots \oplus x_i$. Prefix computations can be parallelized efficiently [12].

Let us rewrite the loop 1.11 as follows:

for all endpoints $t_i \in T$ in non-decreasing order **do**
  if t_i is the left endpoint of an interval $b \in B$ **then**
    $nleft[i] \leftarrow 1$
  else if t_i is the right endpoint of an interval $b \in B$ **then**
    $nright[i] \leftarrow 1$
  end if
end for
for $i \leftarrow 1, \ldots, n-1$ **do**
  $nleft[i] \leftarrow nleft[i-1] + nleft[i]$
  $nright[i] \leftarrow nright[i-1] + nright[i]$
end for

The idea is to initialize $nleft$ and $nright$ as binary vectors where $nleft[i] = 1$ (resp., $nright[i] = 1$) if and only if t_i is the left (resp., right) endpoint of some interval in B. We assume that each element in the list of endpoints T also contains the unique identifier of the interval it is part of, so this information can be obtained in constant time. The result is illustrated by the arrays ① in Fig. 2. This step is embarrassingly parallel, since the loop iterations are independent. The second loop computes the prefix sums of $nleft$ and $nright$ so that, at the end, $nleft[i]$ is the number of left endpoints of intervals in B up to and including t_i (similarly for $nright[i]$). This step can be realized using a parallel prefix sum algorithm [12].

The Full Algorithm Algorithm 2 is the parallel version of the intersection-counting algorithm 1. The main steps of the parallel algorithm are the following:

- The initialization of the list of endpoints T (line 2.5) can be parallelized by implementing T as an array of length $2(n+m)$. The first n elements are the left endpoints of the intervals in A, followed by the n right endpoints in A, followed by the m left endpoints in B, followed by the m right endpoints in B. This makes it possible to place each endpoint into a position in T that is pre-determined, so no race conditions are possible. Each element of T is a tuple $\langle x, side, set, k\rangle$ where x is the position of the endpoint, *side* denotes whether it is a left or right endpoint, *set* is the name of the set it belongs to (either A or B), and k is the unique identifier (index) of the interval within the set the endpoint belongs to. For example, the endpoints of $a_2 = [-3, 7]$ are represented by the tuples $\langle -3, left, A, 2\rangle$ and $\langle 7, right, A, 2\rangle$.
- Sorting the array of endpoints by nondecreasing position (line 2.8) requires a parallel sorting algorithms such as Parallel Radix Sort [12], Parallel Merge Sort [13] or Parallel Quicksort [14,15].
- The initialization of the arrays *nleft* and *nright* (Eqs. (3) and (4)) is done in two separate steps as discussed above. The first step involves an embarrassingly parallel loop (lines 2.9–2.15) that sets $nleft[i] = 1$ (resp. $nright[i] = 1$) iff $T[i]$ is a left (resp. right) endpoint of some interval $b \in B$. The second step consists of two parallel prefix computations, where the call Parallel-Prefix-Sum$v$ replaces the input array v with the array v' of prefix sums, i.e., $v'[i] := \sum_{k=0}^{i} v[k]$.
- Building the index arrays *left_idx* and *right_idx* involves an embarrassingly parallel loop (line 2.18). Note that the tests in lines 2.19 and 2.21 can be done in constant time since the tuples in T already contain the relevant information.
- Finally, the computation of the result $c(i)$, $i = 0, \ldots, n-1$ is again embarrassingly parallel.

Algorithm 2 PARALLEL-INTERSECTION-COUNTING(A, B)

```
1: n ← |A|, m ← |B|
2: T ← ∅
3: nleft ← 0^{n+m}, nright ← 0^{n+m}
4: left_idx ← 0^n, right_idx ← 0^n
   // Initialization
5: for all intervals x ∈ A ∪ B in parallel do
6:     Insert x.left and x.right in T
7: end for
   // Sort the endpoints in non-decreasing order
8: PARALLEL-SORT T
   // Initialize the nleft[] and nright[] arrays
9: for all endpoints t_i ∈ T in parallel do
10:    if t_i is the left endpoint of an interval b ∈ B then
11:        nleft[i] ← 1
12:    else if t_i is the right endpoint of an interval b ∈ B then
13:        nright[i] ← 1
14:    end if
15: end for
16: PARALLEL-PREFIX-SUM nleft
17: PARALLEL-PREFIX-SUM nright
   // Build index arrays
18: for all endpoints t_i ∈ T in parallel do
19:    if t_i is the left endpoint of a_k ∈ A then
20:        left_idx[k] ← i
21:    else if t_i is the right endpoint of a_k ∈ A then
22:        right_idx[k] ← i
23:    end if
24: end for
   // Compute final result
25: for all a_i ∈ A in parallel do
26:    c(i) ← nleft[right_idx[i]] − nright[left_idx[i]]
27: end for
```

Analysis The computational cost of Algorithm 2 depends on the cost of sorting and prefix computations. Let $N = n+m$ be the sum of the sizes of A and B; then, for the CRCW-PRAM (Concurrent-Read, Concurrent-Write, Parallel Random Access Machine) model [16] with p processors we have:

- The initialization of T (line 2.5) is embarrassingly parallel and requires time $O(N/p)$.
- Sorting T (line 2.8) requires time $O\left(\frac{N \log N}{p} + \log \log N\right)$ using Parallel Merge Sort [13].
- Prefix computations of $nleft$ and $nright$ (lines 2.16, 2.17) require time $O(N/p+\log p)$ using p processors [12], which is optimal when $N > p \log p$.
- The initialization of $left\_idx$ and $right\_idx$ (line 2.18) is embarrassingly parallel and requires time $O(N/p)$.
- The computation of $c(i)$ (line 2.25) is embarrassingly parallel and requires time $O(N/p)$.

Algorithm 2 has the desirable property that, when executed serially (i.e., with $p = 1$ processors) it has the same computational complexity of the serial algorithm 1 since they perform the same computations. The algorithm is only $O(\log \log N)$ away from being work-efficient; this small overhead comes from the parallel sorting algorithm.

4. Implementation and experimental evaluation

We implemented the parallel intersection-counting algorithm in C++ using the Thrust library [17]. Thrust provides efficient parallel implementations of several high-level algorithms (copy, sort, merge, prefix computations, reductions, and others). Using template metaprogramming, applications using Thrust can be compiled with minimal modifications for serial or parallel execution, the latter using either OpenMP or CUDA. OpenMP [18] is an open interface for shared-memory parallelism based on the C, C++ and FORTRAN programming languages. OpenMP relies on user-supplied source code annotations based on standard preprocessor directive to generate parallel code mainly for loops, although general task parallelism is also supported. CUDA [19] (Compute Unified Device Architecture) is NVidia's proprietary technology for general-purpose GPU programming; it leverages a special C/C++ compiler that interprets non-standard language extensions, plus library functions that are tailored to the data-parallel programming paradigm of supported GPUs.

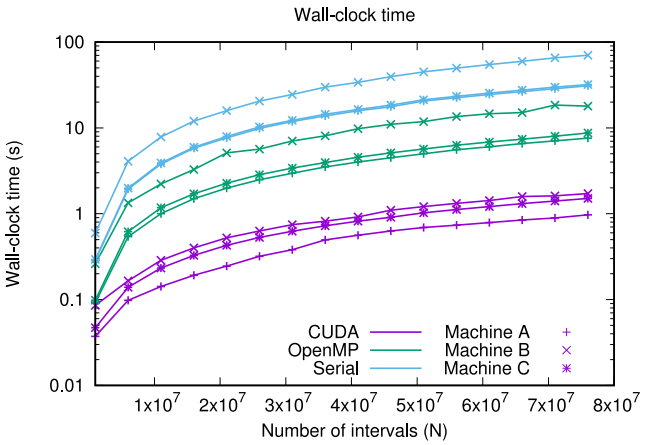

Fig. 3. Wall-clock time as a function of the total number of intervals N, log scale, synthetic workload, lower is better; best viewed in color.

Thrust is appealing because it allows programs to be written in an architecture-agnostic way. Our implementation consists of a single source file that can be compiled for sequential execution, parallel execution on multi-core CPUs, and parallel execution on CUDA-capable devices. This enhances code portability and maintainability, since a single code base is shared across different implementations. Our source program does not use conditional preprocessor directives (`#ifdef`'s), apart from a few CUDA keywords that must be enabled only when producing CUDA code.

We tested the implementations on three machines whose hardware configurations are listed in Table 1. All machines are running the Ubuntu/Linux operating system; we used the GCC compiler provided by the OS with the `-O2 -fopenmp` flags to enable optimization and OpenMP support; the C compiler and CUDA SDK versions are reported in the table. We used the latest version of the Thrust library available at the time of writing.

Some of our machines have processors that use Intel's proprietary HyperThreading (HT) technology [20]. HyperThreading provides two logical processors for each physical core. The two logical processors share part of the hardware resources; studies from Intel and others have shown that in typical applications HT contributes a performance boost between 16–28% [20].

Fig. 3 shows the wall-clock time of all realizations of our parallel algorithm (serial, OpenMP, and CUDA) on all machines under a synthetic workload. We generated N random intervals, $N \in [1 \times 10^6, 80 \times 10^6]$, half of which are assigned to set A and the other half to B (i.e., $|A| = |B| = N/2$). The performance of our algorithm does not depend on the number of intersections, so no special care has been taken to constrain the number of overlaps. The OpenMP versions have been executed by setting the number of threads equal to the number of (virtual) CPU cores. All data points are the average of five independent executions. Note that, since the sequential realization of the parallel algorithm 2

Table 1
Hardware used for the experimental evaluation.

		Machine A	Machine B	Machine C
CPU	Model	i7-9800X	Xeon E5–2603	i7-5820K
	Clock rate (GHz)	3.80 GHz	1.70 GHz	3.30 GHz
	Cores (physical/virtual)	8/16	12	6/12
	RAM	32 GB	64 GB	64 GB
	L2 cache (per core)	8 MB	3 MB	1.5 MB
	L3 cache	16.5 MB	30 MB	15 MB
	Operating System	Ubuntu/Linux 22.04	Ubuntu/Linux 20.04	Ubuntu/Linux 22.04
	GCC version	11.4.0	9.4.0	11.4.0
GPU	Model	Quadro RTX 4000	GeForce GTX 1070	GeForce GTX TITAN X
	CUDA capability	7.5	6.1	5.2
	CUDA cores	2304	1920	3072
	GPU clock rate	1.54 GHz	1.80 GHz	1.08 GHz
	Global memory	7965 MB	8114 MB	12206 MB
	Memory clock rate	6.5 GHz	4.0 GHz	3.5 GHz
	CUDA SDK version	12.3	12.2	12.3

has the same complexity of the state-of-the-art solution, in the following we are comparing the parallel programs with the fastest serial solution known.

Looking at the serial execution time, we observe that Machines A and C are more or less equivalent, while Machine B is considerably slower. The explanation is that both machines have approximately the same clock rate, which is considerably higher than that of Machine B. The presence of a larger Level 2 cache on Machine A does not appear to produce a significant reduction of the wall-clock-time, which is expected since Algorithm 2 exhibits a linear memory access pattern without significant data reuse. The same is observed on the Thrust/OpenMP implementations: Machines A is slightly faster than C since it has a higher number of cores, while Machine B lags behind due to the lower clock rate.

Finally, the fastest Thrust/CUDA implementation is the one running on Machine A since it runs on a more recent GPU; note that the clock rate of Quadro RTX 4000 is lower than the GTX 1070, but Algorithm 2 is memory-bound so it benefits from the higher memory clock rate of the Quadro GPU.

On each machine the throughput improves by about an order of magnitude by going from OpenMP to CUDA. A lower speedup is observed when moving from serial to OpenMP implementations. The results show that CUDA outperform CPU-based implementations by a significant margin, the only exception being when the number N of intervals is low, due to the overhead of GPU initialization. As the number of intervals increases, the performance gap between the serial, parallel, and CUDA versions widens. Indeed, the slowest GPU (on Machine B) is about one order of magnitude faster than the faster CPU (Machine A).

We now discuss the scalability of the OpenMP implementation, defined in term of the relative speedup. The *relative speedup* $S(p)$ of a parallel program executed on p processors is defined as:

$$S(p) = \frac{T_{\text{parallel}}(1)}{T_{\text{parallel}}(p)} \tag{7}$$

where $T_{\text{parallel}}(p)$ is the wall-clock time of the parallel program executed on p processors. The relative speedup satisfies the relation $0 \leq S(p) \leq p$, $S(p) = p$ being the case of perfect scalability. However, in practice $S(p) < p$ due to intrinsically serial portions or parallelization inefficiencies.

Fig. 4 shows the scalability of the Thrust/OpenMP versions. The maximum speedup is between 2.5 and 4, depending on the hardware. Machine A exhibits a slightly better scalability, which is expected since it has the highest CPU clock rate. A sharp drop in the speedup for Machine A is observed between $p = 8$ and $p = 9$ processors, and is due to the HT technology. Indeed, Machine A has eight physical cores that are truly independent: virtual cores mapped onto the same physical execution unit share resources and are therefore not independent. The Linux scheduler is HT-aware and assigns tasks to separate physical cores as long as possible; when the number of tasks increases, virtual cores are used. When two tasks are assigned to two virtual cores from the same physical unit, the speedup is severely degraded due to hardware contention.

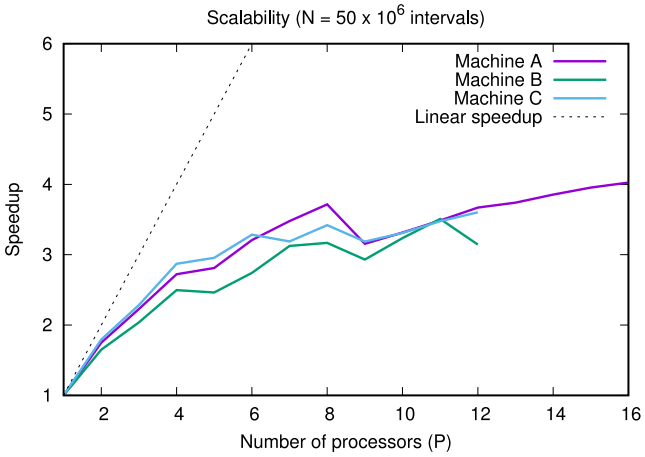

Fig. 4. Scalability of the Thrust/OpenMP implementation; each data point is the average of five independent runs. Synthetic workload ($N = 50 \times 10^6$ intervals), higher is better; best viewed in color.

From Fig. 4 we also observe that the speedup remains below the theoretical maximum p (dashed line) when the number of processors increases. There are three explanations for this: (i) the presence of serial bottlenecks (e.g., contention, intrinsically serial portions of the code) and algorithmic limitations; (ii) OpenMP overheads; and (iii) limited memory bandwidth. Amdahl's Law [21] states that, under some simplifying assumptions, the maximum achievable speedup is $1/\alpha$ where α is the fraction of time spent in parts of the code that do not benefit from additional computational resources. By solving $1/\alpha = 4$ for α, 4 being the approximate maximum speedup, we get that a fraction $\alpha = 0.25$ of the wall-clock time is spent in parts of the code that do not benefit from additional computational resources. We also observe that Algorithm 2 has low arithmetic intensity, defined as the ratio of the number of arithmetic operations and the total amount of data transferred to/from memory. The roofline performance model [22] predicts that applications with low arithmetic intensity are bounded by the available DRAM bandwidth rather than processing power.

Further analysis reveals that a significant portion of the execution time of the Thrust/OpenMP version is spent sorting T, which raises the question whether a more efficient parallel sorting algorithm would reduce the wall-clock time. The GCC compiler ships with a Standard Template Library (STL) for the C++ language that conforms to the extensions for parallelism [23], and provides parallel implementations

Table 2
Wall-clock times on real datasets, lower is better. Speedups are relative to the sequential program. For a graphical representation, see Fig. 6.

Dataset	Program	Machine A		Machine B		Machine C	
n, m		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Chr21	Thrust/Sequential	0.87	1.00×	1.92	1.00×	0.93	1.00×
$n = 509,772$	Thrust/OpenMP	0.38	2.26×	0.76	2.51×	0.42	2.20×
$m = 4,165,891$	Parallel STL	0.20	4.35×	0.43	4,47×	0.19	4.89×
	Thrust/CUDA	0.16	5.53×	0.40	4.71×	0.20	4.64×
Exome	Thrust/Sequential	47.27	1.00×	100.28	1.00×	46.45	1.00×
$n = 25,712,924$	Thrust/OpenMP	17.62	2.68×	36.71	2.73×	19.54	2.38×
$m = 192,173,832$	Parallel STL	8.69	4.87×	20.34	4.93×	9.07	5.12×
	Thrust/CUDA	1.75	27.01×	5.47	18.33×	3.89	11.94×

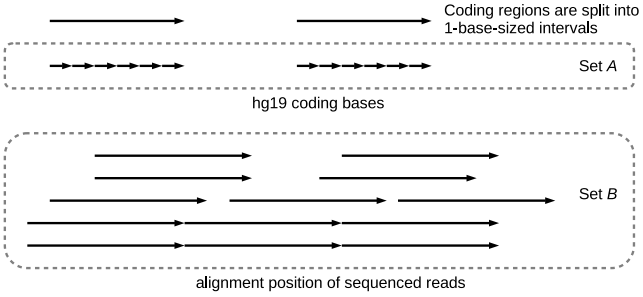

Fig. 5. Structure and derivation of the "Exome" and "Chr21" datasets.

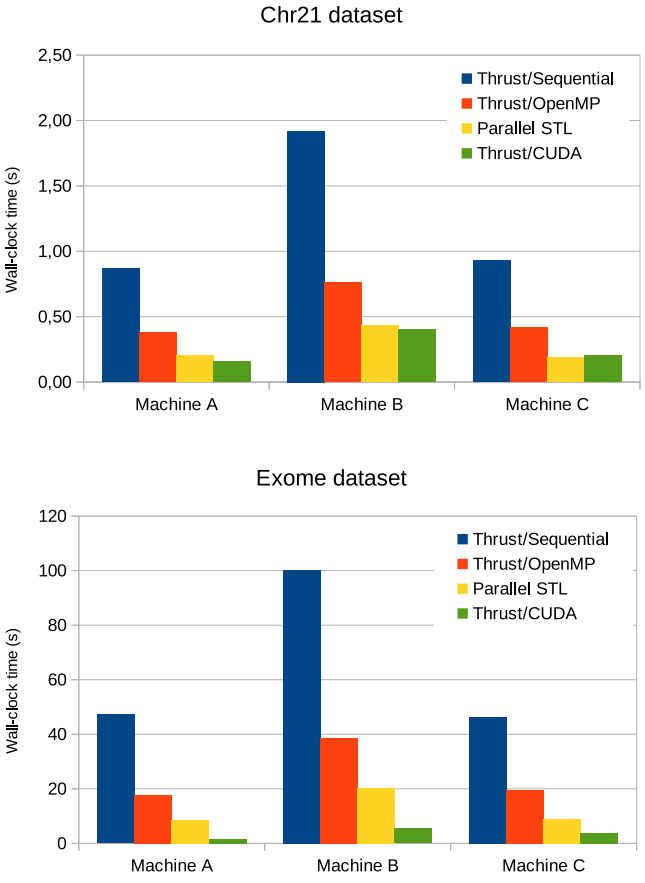

Fig. 6. Wall-clock times on real datasets, lower is better. For the raw data, see Table 2.

of various algorithms (sorting, prefix computation) that are functionally equivalent to their Thrust counterparts. To investigate the overhead incurred by Thrust/OpenMP we realized a separate implementation that relies on the STL only, with a few explicit OpenMP directive to parallelize some loops that cannot be handled using STL constructs. Note that the parallel STL relies on Intel's Threading Building Blocks (TBB)[2], so there is no simple way to constrain the number of execution units that are used.

We study the performance of the parallel algorithm on actual human genetic data (Fig. 5). To create the first interval set, we downloaded the aligned reads in BAM format from a publicly available whole-exome-sequencing of the HG00258 individual performed in the 1000 Genomes Project [24], which are already mapped to a unique region in the hg19 human reference genome. For each read, we extracted its start and end genetic coordinates, yielding a total of 192,173,832 intervals.

We count the intersection of these intervals with a second set of intervals, which was derived from the known coding genomic regions in the hg19 reference genome, downloaded from the NCBI website. We split these 152,502 regions into 1-base-sized intervals, for a total of 25,712,924 intervals. The number of aligned reads overlapping each 1-base interval is the per-base coverage on the coding regions, a commonly used statistics in genomic analyses for quality control checks and other purposes. These two sets are the first dataset, which we call the "Exome" dataset. A second smaller dataset called "Chr21" was obtained as a subset of the "Exome" dataset by taking only the intervals on chromosome 21, comprising 4,165,871 alignments and 506,772 1-base intervals.

Table 2 shows the execution times of all instantiations of the parallel algorithm 2 on the two datasets; the data is provided graphically in Fig. 6 for better understanding. The results confirm our previous findings: Thrust/OpenMP implementations are from 2 to 3 times faster than the sequential program, while the CUDA implementations provide a significant speedup (from 11× to 27× depending on the hardware) on the "Exome" dataset, which drops to 3× – 5× on the "Chr21" dataset.

The speedup achieved by Thrust/OpenMP is about the same across all machines, despite the fact that the hardware spans different generations. This is expected, since the speedup is the ratio of the serial execution time over the parallel execution time, and is therefore independent of the CPU clock frequency. Although the speedup of Thrust/OpenMP does in principle depend on the number of cores, which is different across the test machines, Amdahl's law constrains the maximum speedup in the same way on all machines as we already observed in Fig. 4. Interestingly, the implementation based on the parallel STL is about twice as fast as Thrust/OpenMP. Again, this improvement is about the same across all machines.

The speedup of Thrust/CUDA, on the other hand, varies considerably on the larger Exome dataset from 11.94× of Machine C to 18.33× of Machine B, up to 27.01× of Machine A. Care should be taken in understanding these results, since these values depend on the CPU/GPU combination which varies across the test hardware. It is therefore more appropriate to consider the wall-clock time, according which Machine A (1.75 s) is better than Machine C (3.89 s), which in turn

[2] https://github.com/oneapi-src/oneTBB

is better than Machine B (5.47 s). This is the same behavior observed in Fig. 3 and can be explained by observing that the application has low arithmetic intensity and is therefore limited by the memory bandwidth. The GPU on Machine A has the highest memory clock rate, while Machine C has significantly more CUDA cores than B, with only slightly less memory clock rate.

5. Conclusions

In this paper we presented a parallel algorithm for counting the number of intersections between two sets of one-dimensional intervals. The algorithm has been implemented on shared-memory processors and CUDA-capable GPUs using the Thrust parallel programming library. Thrust allows sequential, OpenMP, and CUDA executables to be produced from the same source code, therefore enabling users to take advantage of CPU or GPU parallelism.

We tested the program on different hardware platforms of different ages using real datasets from the biocomputing domain, as well as on synthetic data. The Thrust/OpenMP version yields a speedup in the range 2×–4× with respect to the serial program, depending on the size of the dataset. The GPU implementation improves upon the OpenMP version, since it provides a speedup of up to 20× on large datasets with respect to the serial implementation.

The performance of Thrust/OpenMP on this specific application turns out to be less than what can be achieved by the parallel STL that ships with the GNU C Compiler, although the latter is not portable to the GPU. This suggests that either the current version of the Thrust library has room for improvements, or that code portability to multicore CPUs and GPUs comes at a price. More research in the area of parallel programming libraries for performance portability is required to investigate these issues.

Funding

Moreno Marzolla was partially supported by the Istituto Nazionale di Alta Matematica “Francesco Severi” – Gruppo Nazionale per il Calcolo Scientifico (INdAM-GNCS), by the EuroHPC EU Regale project (grant number 956560) and by the ICSC National Research Centre for High Performance Computing, Big Data and Quantum Computing within the NextGenerationEU program. Giovanni Birolo and Piero Fariselli were partially supported by the European Union’s Horizon 2020 Brainteaser Project (grant number 101017598).

CRediT authorship contribution statement

Moreno Marzolla: Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Writing – original draft, Writing – review & editing. **Giovanni Birolo:** Conceptualization, Data curation, Writing – review & editing. **Gabriele D’Angelo:** Conceptualization, Methodology, Writing – review & editing. **Piero Fariselli:** Conceptualization, Resources, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Moreno Marzolla reports financial support was provided by Ministero dell’ Universite della Ricerca. Piero Fariselli reports financial support was provided by European Union. Giovanni Birolo reports financial support was provided by European Union. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The source code of the algorithm described in this paper and the datasets used in the performance evaluation are freely available at https://github.com/mmarzolla/parallel-intersections.

References

[1] J.E. Goodman, J. O’Rourke, C.D. Tóth (Eds.), Handbook of Discrete and Computational Geometry, third ed., CRC Press, 2018.
[2] M. Marzolla, G. D’Angelo, Parallel data distribution management on shared-memory multiprocessors, ACM Trans. Model. Comput. Simul. 30 (1) (2020) http://dx.doi.org/10.1145/3369759.
[3] R.M. Layer, K. Skadron, G. Robins, I.M. Hall, A.R. Quinlan, Binary Interval Search: a scalable algorithm for counting interval intersections, Bioinformatics 29 (2013) 1–7, http://dx.doi.org/10.1093/bioinformatics/bts652.
[4] J.F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843, http://dx.doi.org/10.1145/182.358434.
[5] H. Six, D. Wood, The rectangle intersection problem revisited, BIT 20 (1980) 426–433, http://dx.doi.org/10.1007/BF01933636.
[6] M.I. Shamos, D. Hoey, Geometric intersection problems, in: 17th Annual Symposium on Foundations of Computer Science, SFCS 1976, 1976, pp. 208–215, http://dx.doi.org/10.1109/SFCS.1976.16.
[7] S. Neph, M.S. Kuehn, A.P. Reynolds, E. Haugen, R.E. Thurman, A.K. Johnson, E. Rynes, M.T. Maurano, J. Vierstra, S. Thomas, R. Sandstrom, R. Humbert, J.A. Stamatoyannopoulos, BEDOPS: high-performance genomic feature operations, Bioinformatics 28 (14) (2012) 1919–1920, http://dx.doi.org/10.1093/bioinformatics/bts277.
[8] H. Li, J. Rong, Bedtk: finding interval overlap with implicit interval tree, Bioinformatics 37 (9) (2020) 1315–1316, http://dx.doi.org/10.1093/bioinformatics/btaa827.
[9] B. Otlu, T. Can, JOA: Joint overlap analysis of multiple genomic interval sets, BMC Bioinformatics 20 (121) (2019) http://dx.doi.org/10.1186/s12859-019-2698-4.
[10] C. Mao, A. Eran, Y. Luo, Efficient genomic interval queries using augmented range trees, Sci. Rep. 9 (2019) http://dx.doi.org/10.1038/s41598-019-41451-3.
[11] J.F. Allen, Maintaining knowledge about temporal intervals, Commun. ACM 26 (11) (1983) 832–843, http://dx.doi.org/10.1145/182.358434.
[12] G. Blelloch, Scans as primitive parallel operations, IEEE Trans. Comput. 38 (11) (1989) 1526–1538, http://dx.doi.org/10.1109/12.42122.
[13] R. Cole, Parallel merge sort, SIAM J. Comput. 17 (4) (1988) 770–785, http://dx.doi.org/10.1137/0217049.
[14] M. Wheat, D.J. Evans, An efficient parallel sorting algorithm for shared memory multiprocessors, Parallel Comput. 18 (1) (1992) 91–102, http://dx.doi.org/10.1016/0167-8191(92)90114-M.
[15] P. Tsigas, Y. Zhang, A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000, in: 11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2003), 5-7 February 2003, Genova, Italy, IEEE Computer Society, 2003, p. 372, http://dx.doi.org/10.1109/EMPDP.2003.1183613.
[16] J.C. Wyllie, The Complexity of Parallel Computations (Ph.D. thesis), Cornell University, USA, 1979, URL https://hdl.handle.net/1813/7502. AAI8004008.
[17] N. Bell, J. Hoberock, Chapter 26 - Thrust: A productivity-oriented library for CUDA, in: W. mei W. Hwu (Ed.), GPU Computing Gems Jade Edition, in: Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012, pp. 359–371, http://dx.doi.org/10.1016/B978-0-12-385963-1.00026-5.
[18] L. Dagum, R. Menon, OpenMP: An industry-standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1998) 46–55, http://dx.doi.org/10.1109/99.660313.
[19] NVidia CUDA home page, 2022, URL https://developer.nvidia.com/cuda-zone. (Accessed 02 October 2022).
[20] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, A.J. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technol. J. 6 (1) (2002).
[21] G.M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, in: AFIPS ’67 (Spring), Association for Computing Machinery, New York, NY, USA, 1967, pp. 483–485, http://dx.doi.org/10.1145/1465482.1465560.
[22] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785.
[23] Programming languages – technical specification for C++ extensions for parallelism, 2015, ISO/IEC TS 19570:2015.
[24] The 1000 Genomes Project Consortium, A global reference for human genetic variation, Nature 526 (7571) (2015) 68–74, http://dx.doi.org/10.1038/nature15393.

Moreno Marzolla received the Ph.D. degree in computer science from the Università Ca' Foscari Venezia, Italy, in 2004. From 2004 to 2005, he was a Postdoctoral Researcher with Università Ca' Foscari di Venezia. From 2005 to 2009, he was a Research Engineer with the Italian National Institute for Nuclear Physics (INFN), supported by several EU-funded projects in the area of grid and cloud computing. In 2009, he joined the Department of Computer Science and Engineering, University of Bologna, where he is currently an Associate Professor of computer science. His research interests include parallel algorithms, distributed systems, performance modeling and analysis. He served as the Co-Chair for the Production Grids Infrastructure (PGI) Working Group, Open Grid Forum.

Giovanni Birolo is a staff scientist at the Department of Medical Sciences of the University of Torino, Italy. His research interests are in bioinformatics, machine learning and biomedical data analysis.

Gabriele D'Angelo received the Laurea (summa cum laude) and Ph. D. degrees in computer science from the University of Bologna, Italy, in 2001 and 2005, respectively. He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of Bologna. His research interests include parallel and distributed simulation, distributed systems, and cybersecurity. Since 2011, he is in the editorial board of the Simulation Modelling Practice and Theory (SIMPAT) journal published by Elsevier.

Piero Fariselli is Full Professor at the Department of Medical Sciences of the University of Torino, Italy. His research interests include bioinformatics, machine learning, software development and modeling of biological systems.