Parallel Light Speed Labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors

Laurent Cabaret, Lionel Lacassagne, Daniel Etiemble

Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Laurent Cabaret[1] **Lionel Lacassagne**[2] **Daniel Etiemble**[1]

Parallel Light Speed Labeling: an efficient connected component algorithm for labeling and analysis on multi-core processors

Received: date / Revised: date

Abstract In the last decade, many papers have been published to present sequential connected component labeling (CCL) algorithms. As modern processors are multi-core and tend to many cores, designing a CCL algorithm should address parallelism and multithreading. After a review of sequential CCL algorithms and a study of their *variations*, this paper presents the parallel version of the Light Speed Labeling for Connected Component Analysis (CCA) and compares it to our parallelized implementations of State-of-the-Art sequential algorithms. We provide some benchmarks that help to figure out the intrinsic differences between these parallel algorithms. We show that thanks to its run-based processing, the LSL is intrinsically more efficient and faster than all pixel-based algorithms. We show also, that all the pixel-based are *memory-bound* on multi-socket machines and so are inefficient and do not scale, whereas LSL, thanks to its RLE compression can scale on such high-end machines. On a 4×15-core machine, and for 8192×8192 images, LSL outperforms its best competitor by a factor ×10.8 and achieves a throughput of 42.4 gigapixel labeled per second.

Introduction

Connected Component Labeling (CCL) algorithms play a central part in machine vision because they often constitute a mandatory step between low-level image processing (filtering) and high-level image processing (recognition, decision). As such, CCL algorithms have a lot of applications and derivate algorithms like convex hull computation, hysteresis filtering or geodesic reconstruction.

[1]Laboratoire de Recherche en Informatique (LRI)
Univ. Paris-Sud, France
E-mail: laurent.cabaret@lri.fr, daniel.etiemble@lri.fr ·
[2]Laboratoire d'Informatique de Paris6 (LIP6)
Univ. Pierre et Marie Curie (UPMC), France
E-mail: lionel.lacassagne@lip6.fr

Designing a new algorithm is challenging considering the overwhelming literature and the performance achieved by the best existing algorithms. Regarding objectives, it is comparable to developing a new version of matrix multiplication. Indeed, the final result must be the same for all the algorithms *for a given image* and only the execution time does matter.

Today, such a design should address parallel processors as all modern general purpose processors (GPP) are multi-core, whatever their segment: embedded system, workstation or server. For that purpose, CCL algorithms have to consider the specificities of GPP: the *processor pipeline* by minimizing conditional statements (like tests and comparisons) to reduce the number of pipeline stalls, the *cache memories* by limiting random sparse memory accesses to lower the cache misses and the communications/synchronizations between cores.

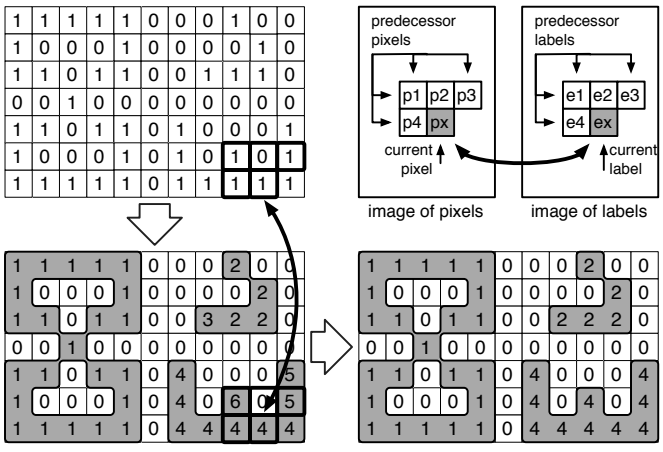

Fig. 1 An example of 8-connected component labeling with Rosenfeld algorithm. Top: binary image, bottom left: image of temporary labels, bottom right: image of final labels

CCL is an intermediate level algorithm. It processes the output data coming from low-level algorithms (typically binary segmentation) and produces an image of labels (fig. 1) easily understandable by humans. Usually, an additional computation step is done to transform the image of labels into synthetic data, called *features*. This step is called features computation (FC).

The most frequently computed features are the boundary of bounding rectangle (for target tracking) and the first order statistical moments (surface, centroid, orientation). They are used by other intermediate or high level (decision) algorithms to analyze an image (for optical character recognition) or a sequence of images (for motion detection and tracking).

The combination of connected component labeling and features computation (fig. 2) is called *Connected Component Analysis* (CCA).

This article introduces a new parallel algorithm called *Parallel Light Speed Labeling* – that is derived from the sequential Light Speed Labeling[19] – for connected component analysis. There are two reasons for considering CCA and not CCL algorithms. First, because for real applications, this is CCA that matters, not CCL. Secondly – and this is the goal of the benchmarks presented in the following sections, standalone FC algorithms can not be efficiently parallelized because they intrinsically have a concurrency issue leading to weak parallelism. This article also shows that, by combining CCL and FC together, the final labeling step (existing in all CCL algorithm) that is memory bound (that prevents efficient scaling) - can be omitted leading to more efficient parallelization.

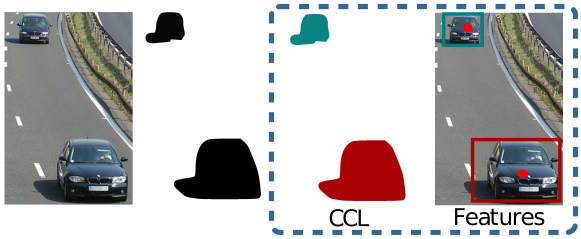

Fig. 2 Connected Component Labeling and Connected Component Analysis in a movement detection system applied to traffic surveillance

The article focuses on the parallelization of these algorithms on multi-core processors and not GPUs because GPUs are not suited for such a kind of algorithms. All the benchmarked CCL algorithms are *direct* two-pass data-dependent algorithms that use an equivalence table that is shared by all threads. Algorithms specialized for GPUs exist [16][32]. They are all *iterative* multi-pass data-independent algorithms. Moreover and as far as we known, none of the existing articles dealing with labeling algorithms for GPUs addresses the features computation issue. In order to make some comparisons feasible, we provide – at the end of the article – some synthetic figures in gigapixel labeled per second.

Our contribution consists in five elements:

- a review of the State-of-the-Art CCL sequential algorithms,
- a benchmark procedure that analyzes the duration of each stage (labeling, features computation, merging) to understand the global performance of each algorithm and especially the features computation part that is usually not addressed by other articles,
- a new parallel algorithm based on the sequential LSL using OpenMP, that efficiently combines labeling and features computation,
- an efficient parallelization of State-of-the-Art CCL algorithms with the integration of an FC step,
- benchmarks for parallel versions on various architectures.

This paper is organized as follows: the first section presents the sequential algorithms and their parallelized implementation. The second section presents the benchmark methodology. The third section presents some algorithms variations that highlight some optimizations and lead us to select a restricted set of algorithms. The fourth section present the sequential results. The fifth section presents the parallel results on two selected architectures. The sixth section presents the performance evolution across number of cores and images size.

1 Connected Component Labeling Algorithms

1.1 Sequential algorithms

Historical algorithms were designed by pioneers like Rosenfeld [27], Haralick [10] and Lumia [20] who designed pixel-based algorithms, Ronse [26] for run-based algorithms. Modern algorithms derive from the algorithms of the 80's and try to make improvements by replacing some components by a more efficient one. An extensive bibliography can be found in [11] and [31]. Except Contour Tracing algorithm [5] that is aesthetic but inefficient, all modern algorithms are two-pass (or less) algorithms, none is a data-dependent multi-pass algorithm. They share the same three steps:

- first labeling, that assigns a temporary label to each pixel and builds labels equivalence,
- label equivalences solving, that computes the transitive closure of the graph associated with the label equivalence table (where temporary labels are associated with final labels, in the equivalence table - usually the smallest one of each the component),

– final (optional) labeling, to replace temporary labels by final labels in the image of labels.

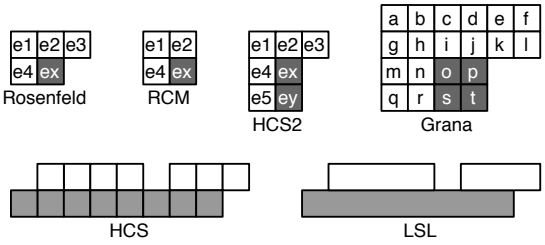

Fig. 3 Masks topology of Rosenfeld, RCM, HCS_2, *Grana*, HCS, and LSL: input labels in white boxes, output labels in gray boxes

They differ on three points: the mask topology, the number of tests for a given mask to find out the minimal value to assign to a label and the equivalence management algorithm.

Mask topology has an impact on both the load/store ratio to label a pixel and the number of temporary labels.

Rosenfeld and derived algorithms use a 4-pixel mask (fig. 3). In order to reduce the number of loads to compute a label (4:1 for Rosenfeld), RCM [14] and HCS_2 [13] provide alternative topologies of 3:1 and 5:2.

As the execution time of an image labeling is not correlated to the total amount of final labels, but to the number of *patterns* that create a new label within a component (see figure 4 and section 2.1 for details), one way to improve CCL algorithms is to widen the label mask. That leads to block-based algorithms like HCS_2 and Grana [8] that respectively compute 2 and 4 labels from a 6-pixel and a 16-pixel neighborhood (fig. 3).

For example, when using the Rosenfeld mask, only two basic patterns trigger label creation within a component (fig. 4), whatever the connectivity (here 8-connectivity). The first one is the *stair*. The second one is the *concavity*. They are responsible for the temporary labels created by pixel-based algorithms. Grana mask can detect some concavities and avoid label creation if these concavities are small enough to entirely fit in the mask.

But the only way to prevent label creation from *stairs* is to use a run-based algorithm like HCS [12] or LSL [19] that first detect the pixel adjacency in the neighborhood before to assign a label to the run. Note that HCS is a "half" run-based algorithm (run-based labeling but pixel-wise equivalence management) and LSL is a "full" run-based algorithm (both run-based labeling and equivalence management). LSL uses an additional but efficient line-relative (see paragraph on LSL) labeling to

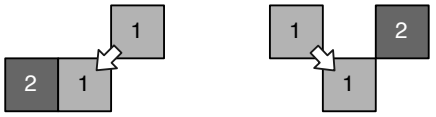

Fig. 4 Minimal 8-connected basic patterns generating temporary labels: stair (left) and concavity (right) for the Rosenfeld mask

generate RLC coding to directly find adjacent runs on the previous line whereas HCS has to perform a test on every pixel to decide to continue to propagate a label or to perform an equivalence.

Minimal positive value. For pixel-based algorithms, one has to find out and propagate the minimum positive value (min^+) to assign to the current label, based on mask topology. It requires many tests and can be optimized by a decision tree (DT) [31]. For example the min_4^+ function requires 4 loads and 7 comparisons for the Rosenfeld mask (algo. 1) whereas with DT (fig. 5), it requires an average of 2.25 loads and 2.25 tests. Decision tree saves both tests and memory accesses.

Algorithm 1: min_4^+: minimum positive value of 4 values with at least a nonzero value

Input: 4 values e_1, e_2, e_3, e_4 with at least a nonzero value
Result: $\epsilon = min_4^+(e_1, e_2, e_3, e_4)$, the minimum positive value

1 $\epsilon \leftarrow +\infty$
2 **if** $(e_1 \neq 0)$ **then** $\epsilon \leftarrow e_1$
3 **if** $(e_2 \neq 0$ ***and*** $e_2 < \epsilon)$ **then** $\epsilon \leftarrow e_2$
4 **if** $(e_3 \neq 0$ ***and*** $e_3 < \epsilon)$ **then** $\epsilon \leftarrow e_3$
5 **if** $(e_4 \neq 0$ ***and*** $e_4 < \epsilon)$ **then** $\epsilon \leftarrow e_4$

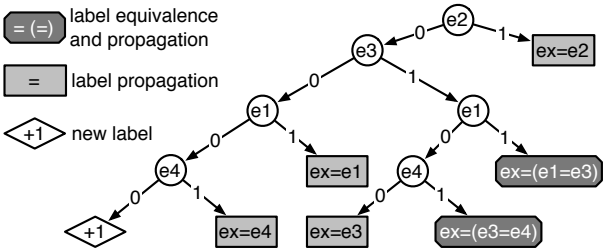

Fig. 5 8-connected Decision Tree for a 4-pixel mask. Label equivalence and the propagation to e_x in dark gray

Equivalence management. There are two main algorithms: the traditional Union-Find (UF) algorithms (algo. 2 and 3) [7] usually associated with the original Rosenfeld algorithm and the Suzuki one that requires

three tables [11]. Table R holds the root of every label: a transitive closure is applied to each equivalence management and merge of labels (algo. 5) and not at the end like UF. Table N behaves like a *next* pointer for a linked list and T points the tail of each set of labels (fig. 6). The complexity of the Suzuki algorithms *Find* and *Union* is the opposite of UF ones: $Find_{Suzuki}$ is just a lookup to R table, but the *Union* (named merge in Suzuki paper) consists in updating the three tables.

Algorithm 2: UF Find(e) algorithm

```
Input: e a label, T an equivalence table
Result: r, the root of e
r ← e
while T[r] ≠ r do
    r ← T[r]
return r
```

Algorithm 3: UF Union(e_1, e_2) algorithm

```
Input: e1, e2 two labels, T an equivalence table
Result: a, the least common ancestor of the e's
a1 ← Find_UF(e1)
a2 ← Find_UF(e2)
if a1 < a2 then
    a ← a1, T[a2] ← a
else
    a ← a2, T[a1] ← a
return a
```

Algorithm 4: Suzuki Find(e) algorithm

```
Input: e a label, R an equivalence table
Result: r, the root of e
r ← R[e]
return r
```

Algorithm 5: Suzuki Union(x, y) algorithm

```
Input: x and y two labels to merge
u ← Find_Suzuki(x)
v ← Find_Suzuki(y)
if v < u then swap(u, v)
i ← v
while i do
    R[i] ← i
    i ← N[i]
N[T[u]] ← v
T[u] ← T[v]
return u
```

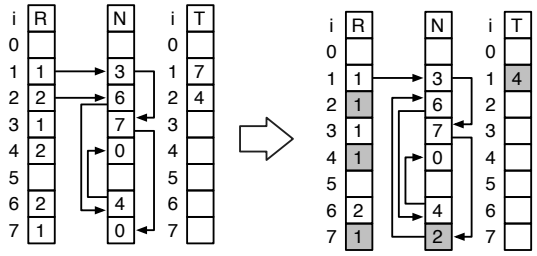

Fig. 6 Suzuki tables: Root, Next and Tail before connecting $x = 1$ and $y = 2$. $S1 = \{1, 3, 7\}$, $S2 = \{2, 6, 4\}$ and after connecting $S1 = \{1, 3, 7, 2, 6, 4\}$. Modified Data are in gray.

Concerning UF, there are many optimizations like path-compression [15], path-halving, and path-splitting [29] that were first analyzed by Tarjan [28] and re-analyzed in [25] on modern computers. As the equivalence management algorithms are independent of the mask topology, a mask can be associated with one or the other. HCS_2 was initially proposed with Suzuki management but Gupta [9] proposed a version with a UF procedure optimized with Rem optimization combined with Splicing (SP) and named "ARemSP" (algo. 6). One can see on figure 7 that the union of labels 8 and 9 makes the biggest of the two roots (and the whole branch) to point to the first smallest value of the other branch. The section 3 will evaluate these variations.

Algorithm 6: ARemSP Union(x, y)

```
Input: x and y two labels to merge
while T[x] ≠ T[y] do
    if T[x] > T[y] then
        if x = T[x] then
            T[x] = T[y]
            return T[x]
        z = T[x], T[x] = T[y], x = z
    else
        if y = T[y] then
            T[y] = T[x]
            return T[x]
        z = T[y], T[y] = T[x], y = z
return T[x]
```

Features computation.

Considering real applications, the fourth step – after the relabeling one – is to perform the features computation (FC) step (fig. 8, left). But if the FC is performed *on-the-fly* during the first labeling, the final labeling is no more required (fig. 8, center). In that case, the first labeling step consists in a line-labeling function and a lineFC function that are applied to the whole image. The procedure to solve the equivalences is then modified to also update the features. Thus, the two passes

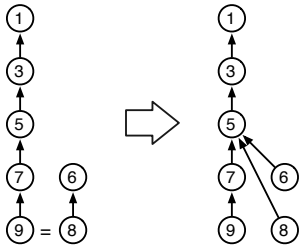

Fig. 7 AREmSP union of labels 8 and 9: 5 if the first label smaller than the root 6

can be reduced to only one pass where FC is done *on-the-fly*. With that modification, all two-pass algorithms become one-pass algorithms. In this paper, the features extracted for each component are the bounding box: $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ and the first statistical moments: S, S_x and S_y.

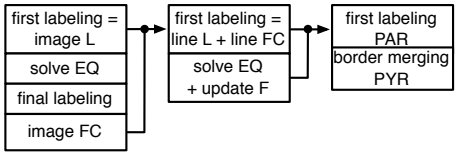

Fig. 8 Sequential (left and center) and parallel (right) CCA synopsis

Algorithm 7: solve equivalences and update features

```
1 for each e ∈ [1 : ne] do
2     r ← T[T[e]] // root of the tree
3     T[e] ← r
4     F[r] ← F[r] ∪ F[e] // features update
```

Light Speed Labeling. Let us now focus on the LSL implementation as this algorithm is more complex than the pixel-based ones. Let us also define the following notations:

- er, a relative label,
- ea, an absolute label,
- a, an ancestor label (aka the root of an absolute label ea),
- X, a binary image of size $h \times w$, X_i is the current line of X, and X_{i-1} the previous line,
- EA, an image of size $h \times w$ of absolute labels ea before equivalence resolution,
- L, an image of size $h \times w$ of absolute labels ea after equivalences resolution,
- ER_i, an associative table of size w holding the relative labels er associated with X_i,
- ner, the number of segments of ER_i,
- RLC_i, a table holding the run length coding of segments of the line X_i, RLC_{i-1} is the similar memorization of the previous line,
- ERA_i, an associative table holding the association between er and ea: $ea = ERA_i[er]$,
- EQ, the table holding the equivalence classes, before transitive closure,
- RLC, a 2D table of size $h \times 2w$ holding all segments of every line, used along LSL evolutions,
- LEA, a 2D-list of absolute labels of every line, used in LSL_{RLE} version,

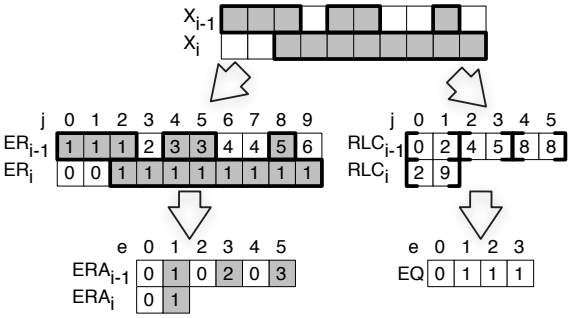

Fig. 9 LSL tables

The specificity of LSL is to be run-based and to use a line-relative labeling (fig. 9). From two consecutive lines X_{i-1}, X_i, two relative labelings are produced where runs (or segments) have odd numbers. In the same way, the associated *run-length codings* are produced (tables RLC_{i-1}, RLC_i). The table ERA_{i-1} holds the translation between *Relative* and *Absolute* labels. To find out the labels of the previous line that are connected to the current segment, one has to read in table ER_{i-1} at the position given by RLC_i the value of relative labels and translates them into absolute labels to update the equivalence table EQ. Details about the sequential implementation of the LSL are available in [19]. In order to figure out what part of the algorithm is important (that makes one algorithm to be faster than other ones), we modified the LSL: there are still the STD (standard) and RLE (RLE compression that uses LEA tables) versions, but the equivalence management can be either the classic Union-Find (named Rosenfeld in the following) or the Suzuki method.

There are two algorithms of the line-relative labeling. One algorithm (algo. 8) is associated with the LSL_{STD} version. It is *data-independent*: there is no `if-then-else` in the algorithm. Another algorithm (algo. 9) is associated with the LSL_{RLE}. It contains a test (line 8) to avoid unnecessary accesses to RLC_i table (like algorithm 8 line 8).

Algorithm 8: LSL segment detection STD

```
   Input: X_i a binary line of width w
   Result: ER_i, RLC_i and ner
 1 x_1 ← 0 previous value of X
 2 f ← 0 front detection
 3 b ← 0 right border compensation
 4 er ← 0
 5 for j = 0 to w − 1 do
 6 |   x_0 ← X_i[j]
 7 |   f ← x_0 ⊕ x_1
 8 |   RLC_i[e_r] ← j − b
 9 |   b ← b ⊕ f
10 |   er ← er + f
11 |   ER_i[j] ← er
12 |   x_1 ← x_0
13 x_0 ← 0
14 f ← x_0 ⊕ x_1
15 RLC_i[er] ← w − b
16 er ← er + f
17 ner ← er
18 return ner
```

Algorithm 9: LSL segment detection RLE

```
   Input: X_i a binary line of width w
   Result: ER_i, RLC_i and ner
 1 x_1 ← 0 previous value of X
 2 f ← 0 front detection
 3 b ← 0 right border compensation
 4 er ← 0
 5 for j = 0 to w − 1 do
 6 |   x_0 ← X_i[j]
 7 |   f ← x_0 ⊕ x_1
 8 |   if f ≠ 0 then
 9 |   |   RLC_i[er] ← j − b
10 |   |   b ← b ⊕ 1
11 |   |   er ← er + 1
12 |   ER_i[j] ← er
13 |   x_1 ← x_0
14 x_0 ← 0
15 f ← x_0 ⊕ x_1
16 RLC_i[er] ← w − b
17 er ← er + f
18 ner ← er
19 return ner
```

Such a kind of front detection and run numbering (algo. 8 line 10) is known in the field of parallel computing as referring to the *scan* concept [2]. Operations of that type are defined more fundamentally as follows:

- given an associative operator $\diamond$ and a vector $v(x)$, $0 \leq x \leq n_N$,
- the $\diamond - scan$ of v produces a vector $w = \diamond - scan(v)$ such that: $w(x) = v(0) \diamond v(1) \diamond ... \diamond v(x_N)$

In the case of LSL, we use a $+scan$ that is applied to a xor-ed vector v such as $v(k) = X(k) \oplus X(k-1)$. It comes:

$$ER_i[j] = \Sigma_{k=1}^{k=j} X_i[k-1] \oplus X_i[k] \quad (1)$$

Here, ner is equal to the number of odd and even segments by construction. So the odd segment er is the $er/2$-th odd segment of the line and its boundaries $[j_0, j_1]$ are stored into $RLC_i[er-1]$ and $RLC_i[er]$ respectively. In our example, the boundaries of the segment $er = 1$ are $RLC_i[0] = 0$ and $RLC_i[1] - 1 = 10 - 1 = 9$.

Algorithm 10: LSL equivalence construction with either UF or Suzuki management

```
   Input: ER_{i−1}, RLC_i, EQ, ERA_{i−1}, ERA_i, ner
   Result: nea the current number of absolute labels,
           update of EQ and ERAi
 1 for er = 1 to ner step 2 do
 2 |   j_0 ← RLC_i[er − 1]
 3 |   j_1 ← RLC_i[er]
 4 |   [check extension in case of 8-connect algorithm]
 5 |   if j_0 > 0 then j_0 ← j_0 − 1
 6 |   if j_1 < n − 1 then j_1 ← j_1 + 1
 7 |   e_r0 ← ER_{i−1}[j_0]
 8 |   e_r1 ← ER_{i−1}[j_1]
 9 |   [check label parity: segments are odd]
10 |   if e_r0 is even then e_r0 ← e_r0 + 1
11 |   if e_r1 is even then e_r1 ← e_r1 − 1
12 |   if e_r1 ≥ e_r0 then
13 |   |   e_a ← ERA_{i−1}[e_r0]
14 |   |   a ← FindRoot(e_a)
15 |   |   for e_rk = e_r0 + 2 to e_r1 do
16 |   |   |   ea_k ← ERA_{i−1}[er_k]
17 |   |   |   a_k ← FindRoot(ea_k)
18 |   |   |   [min extraction and propagation]
19 |   |   |   if a < a_k then
20 |   |   |   |   UpdateTable(ea_k, a)
21 |   |   |   if a > a_k then
22 |   |   |   |   UpdateTable(a, a_k)
23 |   |   |   |   a ← a_k
24 |   |   ERA_i[er] ← a [the global min]
25 |   else
26 |   |   [new label]
27 |   |   nea ← nea + 1
28 |   |   ERAi[er] ← nea
```

The algorithm 10 can use either UF or Suzuki equivalence management. The $FindRoot$ is either $Find_{UF}$ or $Find_{Suzuki}$ (algo. 2 and algo. 4). The $UpdateTable$ function is close to the pseudo-code of $Union$ algorithms but without the two internal calls to $Find$ (algo. 5 and 5, lines 1 & 2). The $UpdateTable(e, r)$ function also makes the hypothesis that e is the first argument and r the second. In that case, for UF algorithm, it is just a simple write into Q: $Q[e] \leftarrow r$.

The last optimization to be done is *zero-offset* addressing. It could seem insignificant, but benchmarks have shown a speedup of 5%. Instead of storing j_0 and j_1 – the actual boundaries of a segment – that also requires the register b to compensated j_1 (algo. 8 line 8 and 9, line 9), the value j_1+1 will be stored into RLC. This leads to an even smaller and faster algorithm for relative labeling (algo. 11 line 7). The same optimization can also be done

for RLE version. The other algorithms of LSL should be also slightly modified to take into account that modification. It appears that their complexity remains unchanged while the line-relative labeling complexity drops. That is an optimization *without counterpart*.

Algorithm 11: LSL segment detection STDZ

```
   Input: X_i a binary line of width w
   Result: ner the number of relative labels on the line X
1  x_1 ← 0 previous value of X
2  f ← 0 front detection
3  er ← 0
4  for j = 0 to w − 1 do
5  |  x_0 ← X_i[j]
6  |  f ← x_0 ⊕ x_1
7  |  RLC_i[e_r] ← j
8  |  er ← er + f
9  |  ER_i[j] ← er
10 |  x_1 ← x_0
11 if x1 ≠ 0 then RLC_i[er] ← w
12 er ← er + x_1
13 ner ← er
14 return ner
```

The FC part is *faster* to compute for LSL thanks to run-length coding: first, the min and max operations are done only twice, for the beginning and the end of a run, instead of as many times as there are pixels in the run. Secondly, the statistical moment can be calculated with the begin and the end indexes. For a given run of interval $[j_0, j_1]$ at line i, $S = j_1 - j_0 + 1$, $S_x = \phi(j_1) - \phi(j_0 - 1)$ and $S_y = i \times S$, with ϕ the first Bernoulli polynomial: $\phi(n) = n(n+1)/2$. If the average run length is greater than 2, LSL requires less arithmetic operations, tests, and memory accesses than pixel-based algorithms.

Sequential Framework. The algorithms that are evaluated have been modified from their original version: there is no more second labeling and the FC is done *on-the-fly* instead of after the second labeling. That make all sequential versions to run faster than previously [4]. A framework has been developed to easily integrate the twenty different algorithms together and simplify their parallelization and their benchmark. In the section 3 we will discuss the algorithm selection depending of their results and specificities.

1.2 Parallel algorithms

In order to simplify the parallelization of the algorithms and to evaluate them with the same fairness, the sequential framework has been extended into a parallel framework that reuses sequential functions like line-Labeling and line-FC. There are two steps for all parallelized (fig. 8, right): the first parallel labeling and the pyramidal border merging. For cache-awareness, the image is split into p horizontal strips, with p the number of threads (it could be twice or four times the number of cores on processors with hyper-threading). For each strip, a descriptor is set with the first and last line indexes. It also contains a pointer to a unique equivalence table (1 pointer for UF, 3 for Suzuki) that is shared by all threads and descriptors.

1.2.1 First parallel labeling

The first step of the parallel labeling (fig. 8, right) correspond to the combination of the two steps of the sequential labeling: line-labeling + line-FC. For a $H \times W$ image, each strip has a size of $h = H/p$ and $w = W$. In order to have all strips being labeled in parallel, one has to *provision the max number of temporary labels* for each strip and thus avoid label collisions. Such amount is equal to $\frac{h+1}{2} \times \frac{w+1}{2}$ for 8-connectivity.

After labeling each strip in parallel, a local solve (a transitive closure of the set of labels produced by each thread) and an update of the associated computed features is done in parallel. These transitive closures can be done in parallel because they are applied to *disjoint sets of labels*. It makes the pyramidal border merging to start with 1-depth trees. Thus, it minimizes the border merging duration that is done in a pyramidal way and not in parallel.

OpenMP is used with a `#pragma omp parallel for` loop applied to an array of descriptors to parallelize all the strip labelings. We have implemented the parallel framework with OpenMP in order to provide a fair comparison with the most related works (see 1.2.4). It can be implemented with OpenCL, Cilk+ or TBB. But evaluating their respective performance is beyond the scope of this article.

1.2.2 Parallel features computation

Parallelizing the features computation has similarities with the parallel computation of a histogram with additional issues. To parallelize a histogram – a well-known concurrency paradigm – one has to duplicate histograms (one by thread), to perform the votes in these temporary histograms then merge them together. It works fine as long as its cardinal is small compared to the size of data but cannot be applied here as the amount of labels can be very high (proportional to $H \times W/4$).

There are three kinds of parallelism for FC.

The first one is spatial and consists in splitting the image into several strips and keeping one equivalence table. But as labels can spread on 2 (or more) strips, memory accesses should be *serialized* with *mutex* or *lock*

when using Posix API and *critical* or *atomic* when using OpenMP API. Such a serialization will become more and more inefficient when the number of cores/threads increases (fig. 10 right).

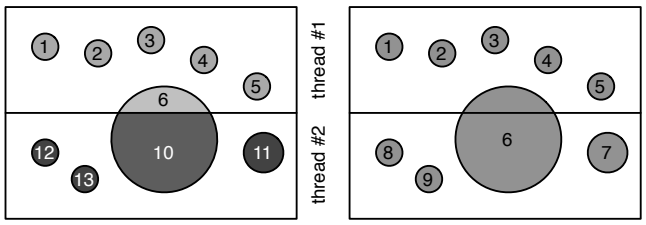

Fig. 10 Parallel features computation. Left: FC is done *on-the-fly* during the first labeling, the two threads can vote for *disjoint-set* of labels. No collision = no required serialization. Right: FC is done after the first labeling, the two sets of labels are not disjoint, serialization is required.

The second parallelism is the label space. Each thread has to vote for labels belonging to a sub-range. There is no serialization, but some load balancing issues can happen if a component is much bigger (typical in natural images) than the others and (like previously) spreads over strips (fig. 11). The second issue is that all threads have to scan the whole image resulting in a stress of the memory busses and poor transfers from external memory (memory wall) to parallel internal cache.

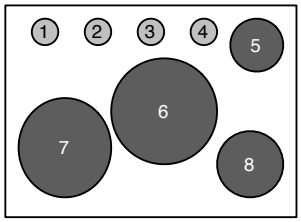

Fig. 11 Parallel features computation. Load balancing issue: first thread votes for light gray labels in the range [1,4], second thread votes for dark gray labels in the range [5,8]. No serialization but both threads must scan the whole image of labels resulting in a high stress on memory

The third one is internal features parallelism. Here there are $n_F = 7$ independent features to compute (x_{min}, x_{max}, y_{min}, y_{max}, S, S_x, S_y). Instead of having an *array of structures* that holds the n_F features, one can use a *structure of arrays* – one by feature – and create n_F threads, each of them having to compute one of the features. As for the second option, all threads have to scan the whole image, but moreover the maximum parallelism is *bound* by the number of features.

We have evaluated the three options. It appears that the first two ones have no speedup (the compiler even de-activates OpenMP pragma and warns for poor parallelism). The third one has a speedup of $\times 1.7$ whatever the number of cores and the memory bandwidth are. So, FC should be done in the same step than the line-labeling. If it avoids the second labeling for the sequential algorithm, it is mandatory for the parallel algorithm.

1.2.3 Pyramidal border merging

There are three strategies for merging the borders. The first one is sequential and inefficient: each border is sequentially processed with the same function that is used within the first labeling. The second one is parallel and false – from a concurrency point of view – as a given label (and its associated features) must be modified by exactly one actor. Like previously, the inefficient modification here would be to use mutexes (or semaphores or locks) to serialize updates which would result in many synchronizations leading to poor parallel performance and "global" serialization. The third one is pyramidal: for each level of the tree (Fig. 12), the merges can be done in parallel (with OpenMP pragma). This is allowed as such a division ensures that, at each step, the strip containing a given label, will be merged with only one strip at a time (a label can be present in only one strip). To be efficiently implemented concurrency issues should be addressed by algorithm modification and not by programming (OpenMP decoration). During the pyramidal merge, the features are updated and propagated to the root of each component. For algorithms using UF, a transitive closure should be done on the whole equivalence table.

If p is the number of strips (and the number of threads and cores) and a power of 2 ($p = 2^q$), the average degree of parallelism *par* is $(2^q - 1)/q$.

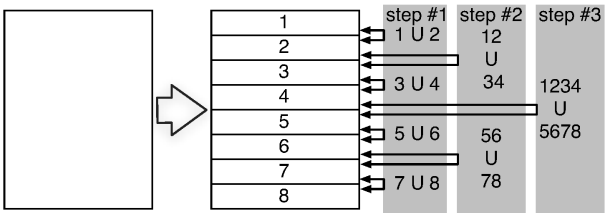

Fig. 12 Pyramidal merging of 8 strips in 3 steps

For example (fig. 12), in the case of $p = 8$ strips there are $q = log_2(p) = 3$ steps. In the first step, we merge in parallel $1 \cup 2$, $3 \cup 4$, $5 \cup 6$, $7 \cup 8$ by processing the borders $1 - 2$, $3 - 4$, $5 - 6$, $7 - 8$. In the second step we merge $\{1, 2\} \cup \{3, 4\}$, $\{5, 6\} \cup \{7, 8\}$, by processing the borders $2 - 3$ and $6 - 7$ and finally in step 3, we merge

$\{1,2,3,4\} \cup \{5,6,7,8\}$ by processing the border $4-5$. The average parallelism is then equal to $7/3 \simeq 2.3$.

Algorithm 12: LSL border merging with either UF or Suzuki management

```
   Input: ER_{i-1}, RLC_i, EQ, ERA_{i-1}, ERA_i, ner
   Result: nea the current number of absolute labels,
           update of EQ and ERAi
 1 for er = 1 to ner step 2 do
 2     j0 ← RLC_i[er − 1]
 3     j1 ← RLC_i[er]
 4     [check extension in case of 8-connect algorithm]
 5     if j0 > 0 then j0 ← j0 − 1
 6     if j1 < n − 1 then j1 ← j1 + 1
 7     er0 ← ER_{i-1}[j0]
 8     er1 ← ER_{i-1}[j1]
 9     [check label parity: segments are odd]
10     if er0 is even then er0 ← er0 + 1
11     if er1 is even then er1 ← er1 − 1
12     if er1 ≥ er0 then
13         ea ← ERA_{i-1}[er0]
14         a ← FindRoot(ea)
15         for erk = er0 + 2 to er1 do
16             eak ← ERA_{i-1}[erk]
17             ak ← FindRoot(eak)
18             [min extraction and propagation]
19             if a < ak then
20                 UpdateTable(eak, a)
21             if a > ak then
22                 UpdateTable(a, ak)
23                 a ← ak
24         [a holds the runs min value]
25         eai ← ERA_i[er]
26         ai ← FindRoot(eai)
27         if a < ai then
28             UpdateTable(ai, a)
29         if a > ai then
30             UpdateTable(a, ai)
31             a ← ai
```

The algorithm for the border merging for LSL (algo. 12) is close to the sequential version (algo. 10) except that there is neither write to ERA_i nor label creation. Instead, there is an additional part (lines 25-31) to union a, the min value of the upper runs with a_i the root of the lower run.

For the pixel-based algorithms one has to do the same kind of modifications. As far as we know the authors of the sequential algorithms cited in the first part have never published any parallel version of their algorithm, except a parallelization of Suzuki algorithm on Tile64 processor (see next section). So we have created a pixel-based merge algorithm for all these algorithms.

The equivalence management algorithm remains unchanged but has an additional part to merge the upper labels with the lower label.

The mask topology has been modified. Considering the Rosenfeld mask (fig. 13), there is no more e4 label. The equivalences are computed between the top three labels with bottom label ex that already has a value. Moreover, because of this mask modification, the parallel version of the RCM and HCS2 masks are now identical to the Rosenfeld one.

If the sequential algorithm uses a decision tree, the corresponding border merging uses a simplified one (as there is no more new label) as described in figure 14.

Fig. 13 Rosenfeld parallel mask. RCM and HCS2 masks are identical to Rosenfeld one.

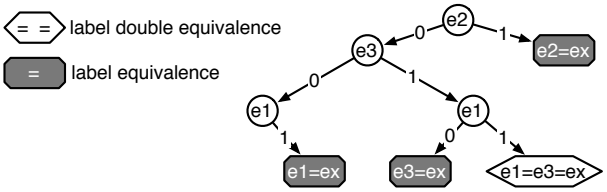

Fig. 14 8-connected parallel Decision Tree for a 4-pixel mask. Label equivalence in dark gray and label double equivalence in white

1.2.4 Related works on multicore processors

As far as we know, only 3 papers deal with parallel CCL on multicore. These three papers present CCL algorithms *without* features computation.

Niknam *et al.* [23] is the first paper presenting the parallelization of Suzuki algorithm on a 16-cores AMD Opteron 885. The max speedup is $\times 2.5$ on 4 threads for 256×256. With 16 threads, the speedup falls to $\times 1.2$. The author's explanation for poor performance is Non-Uniform Memory Access (NUMA) issues and cache misses for bigger images.

Chen *et al.* [6] presents the parallelization of Suzuki algorithm on Tilera Tile64 processor. Border merging is done in a pyramidal way. The best speedup (from three algorithm variations) is around $\times 11.38$ with 48 cores for 2000×1500 images.

Gupta *et al.* [9] presents a modification of HCS_2 algorithm. Suzuki equivalence management is replaced by UF with ARemSP (for the sequential algorithm) and PARemSP (for the parallel algorithm) optimizations. ARemSP is the most efficient UF optimization among 35 variations according to [25]. The processor is a 2×12-core AMD Magnycours. The best speedup is around ×10 with 24 threads for database images. The border merging step is done in parallel with OpenMP locks. In order to reach a higher level of performance, the authors must use a 465.20 MB image for which they reached a speedup of ×20.1. For HCS_2-ARemSP we took their ARemSP proposition but with a pyramidal merge instead of their parallel merge.

1.2.5 Other existing works

Some algorithms were also designed for specific architectures like Bailey's one [1][21] that targets Field Programmable Gate Array (FPGA) and smart cameras. The Union-Find structure is still used to hold the labels equivalence but this algorithm uses a stack in order to avoid the non-determinist `Find()` function. The reason is that, unlike general purpose processors, FPGA should implement algorithm in a data-independent way without a `While()` loop in order to determine their clock frequency. This algorithm was parallelized by Klaiber *et al.* in [18][17].

2 Benchmark methodology

We now present the images and the processors used for the benchmarking.

2.1 Random images

Usually, papers evaluate CCL performance first with random images (varying pixel density from 0% to 100%) for hard-to-label benchmarks and secondly with image data base. But data base can be biased and may favor some algorithms. As we want our benchmark to be as fair as possible (quite difficult with data-dependent algorithms) we decided to select Mersenne Twister MT19937 [22] to control the random number generation and to extend random images by changing the pixel granularity. The initial random image has a granularity of 1. Then we create g-random images whose blocks of pixels have a size of $g \times g$ (Fig. 16), with $g \in [1:16]$. This methodology highlights some algorithm behavior linked to the number of labels and the image density. An important point is that we propose a reproducible benchmark procedure [30]. As the random number generator is not the `rand` function provided into the `libC` library, but MT19937 generator with *seed* equal to zero, our procedure can be *exactly reproduced* by any reader.

The figure 15 provide the temporary labels distribution for granularity $g \in \{1, 2, 4, 8\}$ for pixel-based, run-based and *Grana* algorithms (red, magenta and blue). The number of final labels (green), concavities (cyan) and stairs (orange) is also provided.

First, if we compare run-based and pixel-based label distribution, we can see that run-based curve always has the same behavior (close to the final label curve), contrary to the pixel-based curve. The reason is that the amount of concavities is proportionally constant (from one granularity to another one) to the number of final labels. For $g \geq 2$, it appears that the amount of stairs becomes bigger than concavities, and then the pixel-based also proportionally generates more temporary labels than for $g = 1$. That is the reason run-based algorithms have a better execution time when g is growing: they avoid more and more label creation.

Concerning *Grana* algorithm, it generates quite the same number of temporary labels for $g = 1$ than pixel-based ones. For $g = 2$ it comes closer to run-based algorithms as its wide mask avoids many temporary labels. But for $g \geq 4$, its wide mask does not avoid label creation, as 4-pixel wide stairs and concavities are beyond the pixel neighborhood.

The random benchmark protocol used is: random images with density d varying from 0% to 100% with a step of 1%, and granularity g varying from 1 to 16 with a step of 1. As the sequential results are intended to be compared to the parallel results, we use the same image size for the sequential benchmark than for the first step of the parallel benchmark: 2048×2048. The parallel benchmark was also realized on 4096×4096 and 8192×8192 in order to evaluate the parallelization performance not only from the number of cores point of view but also from the amount of data point of view. As one machine aggregates 60 cores (see next section), it is also important to have images big enough – but still realistic – to produce enough workload per core.

The following results are featured for three granularities: first $g = 1$ as it is the worst configuration for all algorithms, second $g = 4$ as it was the turning point for sequential algorithms in previous works [4] and also because with this granularity all pixel-based algorithms are equal from the mask point of view, and third $g = 16$ (structured random images) to get the optimal behavior of all algorithms close to the behavior with natural images.

2.2 Image data base

The Standard Image Data-Base (SIDBA) has been used for natural image labeling. Gray-scaled images have been automatically binarized with Otsu algorithm [24]. For both random images and natural ones, we provide the

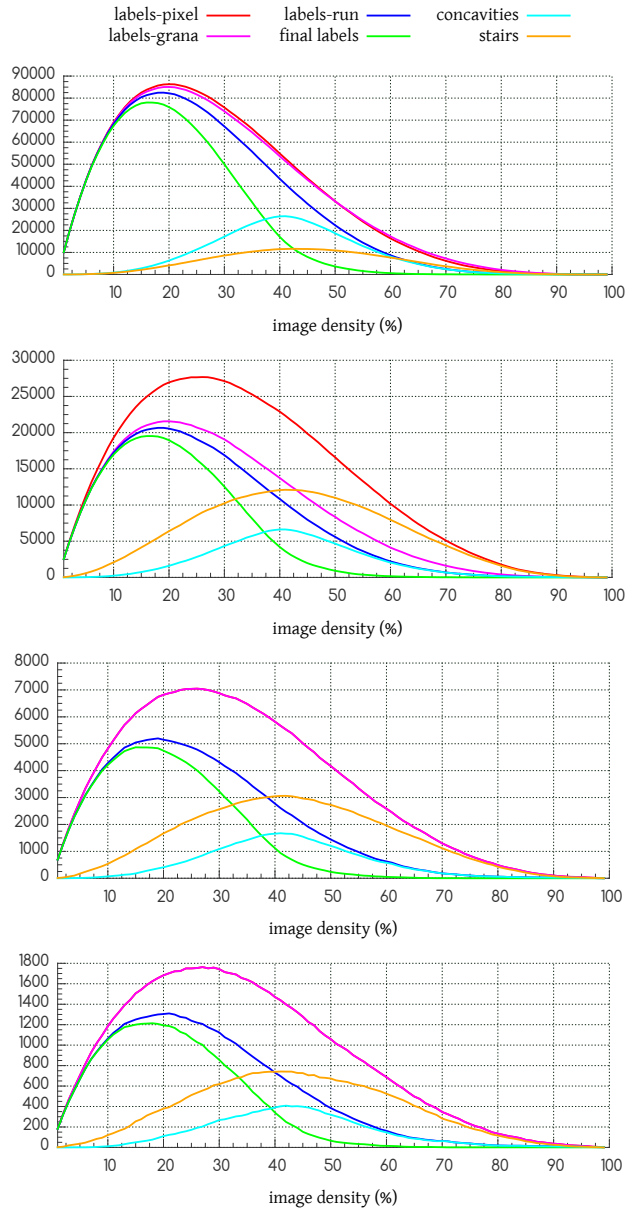

Fig. 15 Distribution of labels, concavities and stairs versus density for granularity $g \in \{1, 2, 4, 8\}$

cpp (cycles per point) of each algorithm with FC. As the parallel algorithms are intended to handle more data than sequential ones, we rescaled these images from 800×600 to 3200×2400 without anti-aliasing and named the resulting database SIDBA4. As a matter of fact, the average density of the images used in this benchmark is 55.5% that is after the percolation threshold.

2.3 Benchmarked machines

For the benchmarks, we use two parallel machines with processors belonging to server class to evaluate the impact of algorithm parallelization. The first one is a 2×12-core Ivy-Bridge Xeon E5 2695v2 running at 2.4 GHz. The second one is a 4×15-core Ivy-Bridge Xeon E7 8890v2 running at 2.8 GHz. As presented in the previous section, the algorithms cannot scale perfectly for some reasons: lot of control structures that is an issue for the processors pipeline, many memory accesses for few computations that is an issue for external memory and cache and a pyramidal part that could not scale. So the parallelization on the 24-core dual-socket Xeon is intended to generate a medium stress whereas the parallelization on the 60-core quadri-socket Xeon is intended to generate the highest possible stress and to get an idea of the behavior of future manycore processors.

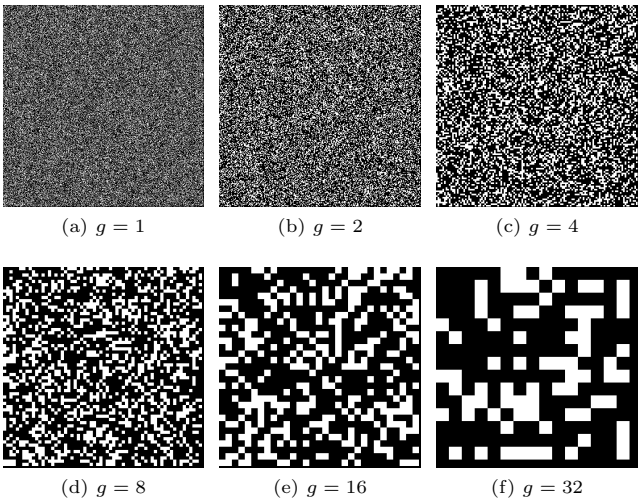

Fig. 16 random images with density = 35% at granularity $g \in 1, 2, 4, 8, 16, 32$

Fig. 17 SIDBA database sample

Most of the results are focused on these two machines because we want to compare the performance of parallel LSL with the other algorithms for machines with lots of cores. But in order to have results for more common machines, we have also used two processors belonging to workstation class.

The first one is a 4-core Sandy-Bridge i7-2600K processor at 3.4 GHz, which architecture is close to the Ivy-Bridge Xeon and a 4-core Skylake i7-6700K processor running at 4.0 GHz. As these processors are separated by

three generations (Ivy-Bridge, Haswell and Broadwell) we can observe the impact of the architecture evolution on the algorithm performance.

For all these processors, we use Intel C Compiler 2015. Processors are running at their nominal frequency (SpeedStep and Turbo technologies being disabled) and OpenMP is configured with compact mode to enforce spatial locality of strips within caches.

2.4 Benchmark metrics

The metric used in the benchmarks is the execution time measured in cycles per point (*cpp*), as processors have different frequencies. It makes also the comparisons easier as three image size are benchmarked (2048×2048, 4096×4096 and 8192×8192). With *cpp*, the comparison is straightforward, whereas with second or millisecond, the reader has to apply a scaling factor (×4 or ×16).

In order to provide a direct comparison with the related works, some results are also expressed in millisecond and in gigapixel per second. The *cpp* is useful to compare two algorithms on the same architecture or the same algorithm on two architecture while execution time in ms and throughput in Gpixels/s are useful to compare results from two publications and to get *absolute* results for realtime execution.

It has been shown in [3] that, for a *single-threaded* benchmark and for six generations of Intel processors (Conroe, Penryn, Nehalem, Sandy-Bridge, Ivy-Bridge and Haswell), the performance in *cpp* of the eight algorithms are very stable. The key processor was the Nehalem which introduces a new memory bus between the external memory and the cache hierarchy. On one side, Conroe performance is close to Penryn one. On the other side, Sandy-Bridge, IvyBridge and Haswell performance are close to the Nehalem one. But Haswell is faster than Nehalem thanks to its frequency increase, from 2.66 GHz up to 3.5 GHz.

This result enforces the utilization of *cpp* instead of the execution time in seconds to compare algorithms and architectures.

When an average *cpp* is provided, it is the average, for a given granularity g of the *cpp* for all evaluated density $d \in [0\%, 100\%]$.

3 Algorithms *Variations*

As described in the algorithms presentation, they can be modified in many ways and more particularly with equivalences management variations. In this part, we present and analyze the performance differences between these the variations.

The previously described benchmark procedure was applied to twenty different algorithm *variations*. In this section, we discuss *Rosenfeld* and HCS_2 variations. Then, we select a restricted set of algorithms with specific properties to make the subsequent tables and figures easier to analyze.

3.1 Rosenfeld *variations*

Original Rosenfeld algorithm [27], introduces the CCL principle and is a great canvas to challenge the various optimizations that literature introduces since this first communication. The Rosenfeld variations that are challenged here are: classical Rosenfeld, Rosenfeld + decision tree (DT), Rosenfeld + path compression (PC), Rosenfeld + decision tree + path compression (DT + PC), Rosenfeld + quick-union (QU), Rosenfeld + decision tree + quick-union (DT + QU), Rosenfeld + decision tree + AREmSP union-find (DT + AREmSP).

Random images:

Table 1 Rosenfeld Variations: Average *cpp* for granularity $g \in \{1, 2, 4, 8, 16\}$ on one IvyBridge core

	granularity				
algorithms	g=1	g=2	g=4	g=8	g=16
Rosenfeld DT	31.41	19.73	14.24	12.08	10.95
Rosenfeld DT AREmSP	31.19	19.76	14.29	12.13	11.03
Rosenfeld DT PC	33.12	21.64	16.13	13.22	11.58
Rosenfeld DT QU	34.42	22.69	16.93	13.92	12.44
Rosenfeld	44.73	30.77	21.62	16.95	14.64
Rosenfeld PC	50.21	33.71	22.87	17.62	14.90
Rosenfeld QU	50.96	34.74	23.69	18.18	15.31

As figure 18 shows, optimizations have varying effect on the final result. There are two sets of algorithms, those with a decision tree and those without. All the algorithms using decision tree are very close with a thin advantage to the Rosenfeld + DT variation, followed by Rosenfeld DT AREmSP, Rosenfeld DT PC and Rosenfeld DT QU, behind we find the classical Rosenfeld, followed by Rosenfeld + PC and finally Rosenfeld + Quick Union.

Real case images:

For SIDBA4 results, Rosenfeld DT QU is first followed by Rosenfeld DT PC, Rosenfeld DT AREmSP, Rosenfeld + DT, the classical Rosenfeld, Rosenfeld PC and finally Rosenfeld + Quick Union.

Results show that without decision tree some "optimizations" have a negative impact on the performance as PC or QU. For PC, each encountered label is compressed even if it was already encountered in the same mask neighborhood that is costly due to the systematic

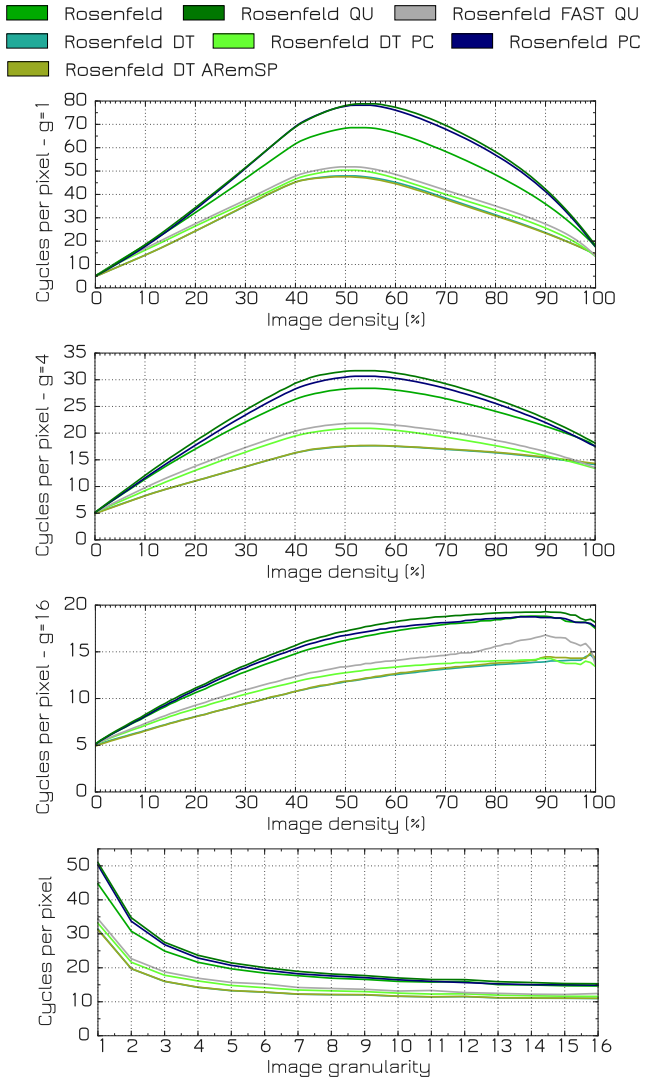

Fig. 18 Rosenfeld Variations: *cpp* for granularity $g \in \{1, 4, 16\}$ and average *cpp* vs granularity with features computation on one IvyBridge core

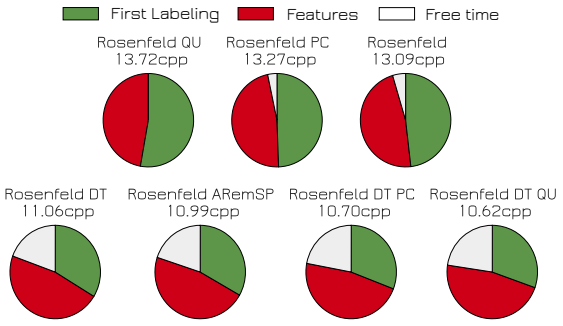

Fig. 19 Rosenfeld Variations: average *cpp* on SIDBA4 images with features computation on one IvyBridge core, piecharts are normalized according to the slowest algorithm

while loop. But when a decision tree is used, the number of labels to compress comes lower and the faster equivalence solving introduce by PC is worthy. That is especially true on natural images where the number of labels is lower than random images for a given density.

Finally, we select Rosenfeld + DT + PC as the representative variation because it belongs to the best set and in order to evaluate DT + PC behavior in a parallelized context.

3.2 HCS_2 *variations*

In [9], the authors propose HCS_2 variation with UF with DT+ARemSP. We challenged this version with the classical HCS_2, HCS_2 + decision tree, HCS_2 + decision tree + Quick-Union.

Random images:

Table 2 HCS2 Variations: Average *cpp* for granularity $g \in \{1, 2, 4, 8, 16\}$ on one IvyBridge core

	granularity				
algorithms	g=1	g=2	g=4	g=8	g=16
HCS_2 DT ARemSP	27.73	18.42	14.04	11.76	10.50
HCS_2 DT	28.42	18.64	14.10	11.81	10.53
$HHCS_2$ DT QU	28.83	18.57	13.80	11.61	10.56
HCS_2	35.97	22.97	16.70	13.36	11.85

As for Rosenfeld, the most significant algorithm modification is the decision tree, with a thin advantage to the HCS_2 DT ARemSP, followed by HCS_2 DT, HCS_2 DT + QU and finally classical HCS_2.

Real case images:

The relative order of the results is the same for SIDBA than for random images. The difference between classical HCS_2 and its variations only affects the first labeling part.

These results confirm the assertion of [9] about the performance of ARemSP for the HCS_2 algorithm but the difference with the DT variation is in the thick of the line. Finally, we select HCS_2 + DT + ARemSP as the representative variation to provide direct comparison with the results of [9].

3.3 Restricted algorithm set selection

These results allows us to select a representative set of algorithms for the remainder of the paper:

- Rosenfeld: original Rosenfeld [27] algorithm with UF memory management improved with DT and PC.
- Suzuki: 4-pixel mask with Suzuki tables management [11] improved with DT,

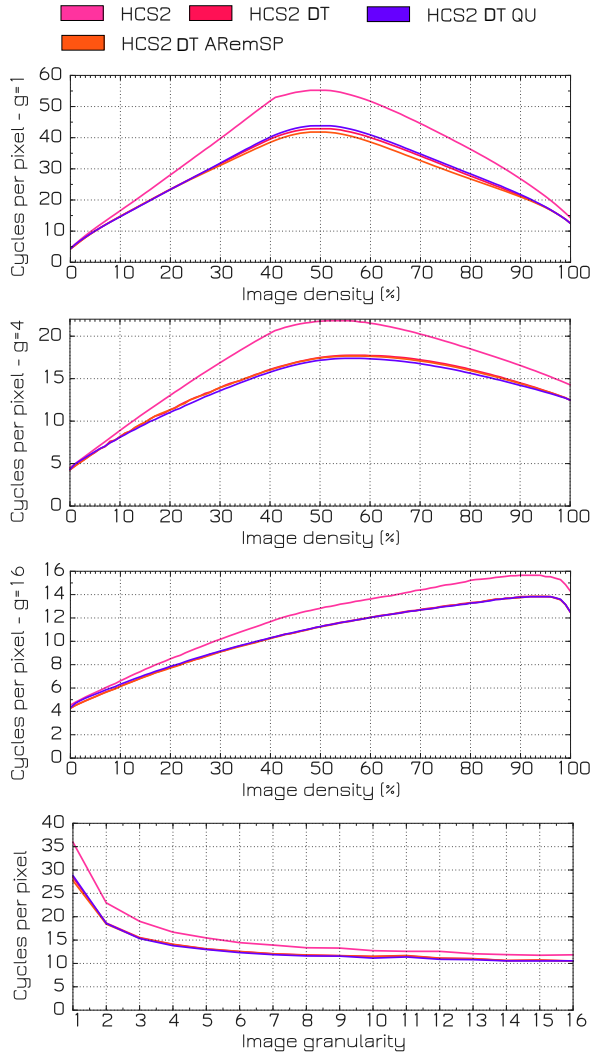

Fig. 20 HCS2 Variations: *cpp* for granularity $g \in \{1, 4, 16\}$ and average *cpp* vs granularity with features computation on one IvyBridge core

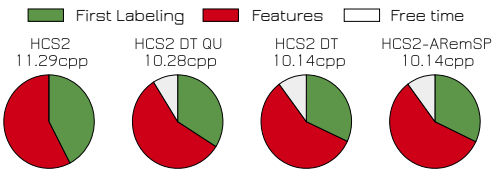

Fig. 21 HCS_2 variations: average *cpp* on SIDBA4 images with features computation on one IvyBridge core, pie-charts are normalized according to the slowest algorithm

– Grana: block-based algorithm with 128-stage DT, using Suzuki management [8],
– RCM: pixel-based algorithm with Suzuki management and DT [14],
– HCS_2 DT ARemSP: block-based algorithm with UF memory management (instead of Suzuki one) and improved with Rem+Splicing optimization [9],
– HCS: run-based algorithm with Suzuki management [12],
– LSL: run-based algorithm with either UF or Suzuki management, with two variants: LSL_{STD} (standard version, as systematic as possible) and LSL_{RLE} (version with compression).

These algorithms will be first studied in a sequential context and then parallelized on various architectures.

4 Benchmarks Results and Analysis: Sequential algorithms

4.1 Global Analysis

Density behavior: Figure 22 shows us that algorithms' curves, for $g = 1$, are symmetrical about their maximum value. The abscissas of the maximum values are contained in the $[45\%; 55\%]$ area depending on the algorithm. Concavities and stairs (fig. 15), lead to temporary label creation and labels merging, they also increase the probability of having more tests to perform in the decision tree (e.g., stair makes to traverse all the DT graph until the label creation node "+1" - figure. 5) and doing so, increase *cpp*.

One can observe that when the number of stairs and concavities decrease (g comes higher) the density curves tend to flatten.

Granularity influence: Table 3 and figure 22 describe the behavior of algorithms faced to images of different granularities. The main trend is that when g grows *cpp* drops. First quickly for $g \in \{1, 2\}$, and then slowly for $g \in [2{:}16]$. One can notice that LSL_{RLE} is the most accelerated when granularity grows while LSL_{STD} is the most regular. It comes from their construction as explained in[19].

Above $g = 2$, RLE is the absolute fastest independently of the equivalence management algorithm, and LSL_{STD} is the most stable in *cpp*. One can notice that while LSL_{STD} and all the pixel-based algorithms became stable over granularity evolution ($\times 1.1$ between $g = 8$ and $g = 16$), LSL_{RLE} is still accelerating ($\times 1.3$ between $g = 8$ and $g = 16$).

The ARemSP optimization is efficient and makes HCS_2 run as fast as HCS (without this optimization HCS_2 was one of the slowest pixel-based algorithm [4]).

Real case images: SIDBA4 natural images database benchmark confirms random images conclusion.

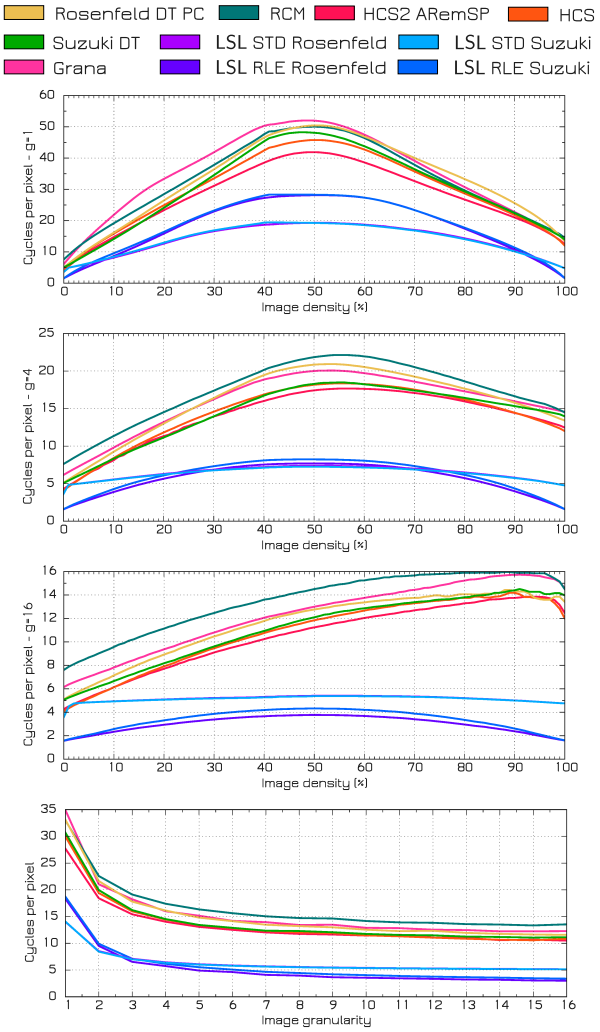

Fig. 22 Sequential algorithms: *cpp* for granularity $g \in \{1, 4, 16\}$ and average *cpp* vs granularity with features computation on one IvyBridge core with random 2048×2048 images

We give the results for each algorithm (table 4) with min, average and max values for processing time and *cpp*, for direct comparison with others articles results (considering that there is 16× more pixels in SIDBA4 than SIDBA). LSL_{RLE} is first followed by LSL_{STD}, HCS, HCS_2-ARemSP, Rosenfeld-DT-PC, Suzuki-DT, Grana, and RCM.

One can notice that LSL_{STD} is extremely stable in execution time on all images: the variation is 1.1 *ms* (< 7%) while the second most stable (LSL_{RLE}) has a variation of 2.4 *ms*.

4.2 In-Depth algorithm time-slicing analysis

In order to well understand the time distribution between each step of CCL algorithm, we monitored the first labeling and the features computation steps.

Table 3 Sequential algorithms: average *cpp* according to granularity with features computation on one IvyBridge core, with random 2048×2048 images

	granularity				
algorithms	g=1	g=2	g=4	g=8	g=16
$\text{LSL}_{\text{RLE-Rosenfeld}}$	18.29	8.99	5.27	3.66	2.84
$\text{LSL}_{\text{RLE-Suzuki}}$	18.47	8.75	5.04	3.46	2.66
$\text{LSL}_{\text{STD-Rosenfeld}}$	13.88	8.05	6.03	5.23	4.86
$\text{LSL}_{\text{STD-Suzuki}}$	13.96	8.02	5.99	5.21	4.85
HCS	29.71	18.51	12.93	10.69	9.47
HCS_2 DT ARemSP	26.97	17.07	12.45	10.61	9.70
Rosenfeld DT PC	31.35	19.94	14.54	12.02	10.76
Suzuki DT	32.58	21.02	14.68	12.03	10.81
Grana	34.89	20.09	14.91	12.41	11.24
RCM	33.23	21.49	15.89	13.33	11.98

Table 4 Sequential algorithms: average execution time and *cpp* on SIDBA4 images with features computation on one IvyBridge core at 2.4GHz

	time (ms)			*cpp*		
algorithms	min	avg	max	min	avg	max
$\text{LSL}_{\text{RLE-Rosenfeld}}$	5.2	6.4	7.9	1.64	2.01	2.47
$\text{LSL}_{\text{RLE-Suzuki}}$	5.4	6.9	8.8	1.69	2.17	2.76
$\text{LSL}_{\text{STD-Rosenfeld}}$	15.8	16.4	16.9	4.92	5.11	5.28
$\text{LSL}_{\text{STD-Suzuki}}$	15.9	16.5	16.9	4.97	5.16	5.28
HCS	25.4	30.7	37.1	7.94	9.58	11.59
HCS_2 DT ARemSP	26.9	32.5	39.1	8.40	10.14	12.21
Rosenfeld DT PC	28.9	34.2	40.6	9.03	10.70	12.69
Suzuki DT	30.6	37.5	45.0	9.57	11.72	14.06
Grana	32.3	38.2	45.3	10.09	11.95	14.17
RCM	36.7	41.0	46.4	11.47	12.80	14.49

Random Images: One can see (fig. 23 & fig. 24), that labeling and features computation parts are very similar for all pixel-based algorithms, and have quite the same duration.

The FC part is by far smaller for both LSL thanks to run-length coding: first, min and max operations are done only twice, for the beginning and the end of a run, instead of as many times as there are pixels in the run. Secondly, the statistical moment can be calculated inexpensively with the begin and the end indexes.

When g grows, the labeling part of LSL_{STD} is very similar to pixel-based algorithms while as there are fewer regions, these parts for LSL_{RLE} become faster. For pixel-based algorithms, as the duration of these parts decreases, FC becomes the main part of the total computation time, whereas, for *LSL* versions FC is so fast that it is a negligible part of the graph.

Furthermore, for a given density, there is a better temporal and spatial cache locality for $g = 4$ than for $g = 1$ (and a smaller amount of labels, and shorter decision trees) that leads to a general *cpp* decrease. This phenomenon can be observed when comparing figures 23 and 24. We can also observe that while the fraction of FC (related to the total execution time in *cpp*) increases for pixel-based algorithms, it remains approximatively constant for *LSL* algorithms. Thus, the ratio between

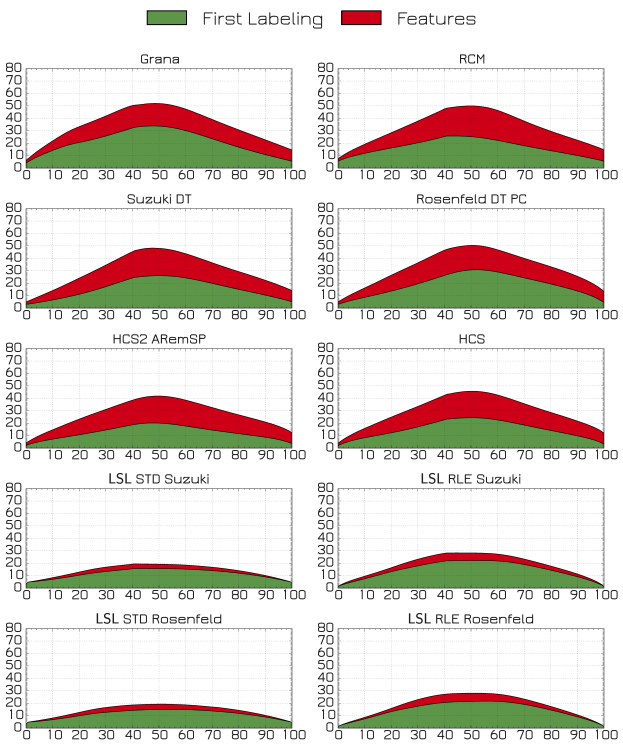

Fig. 23 Sequential algorithms: step analysis for $g = 1$ on one IvyBridge core, with random 2048×2048 images

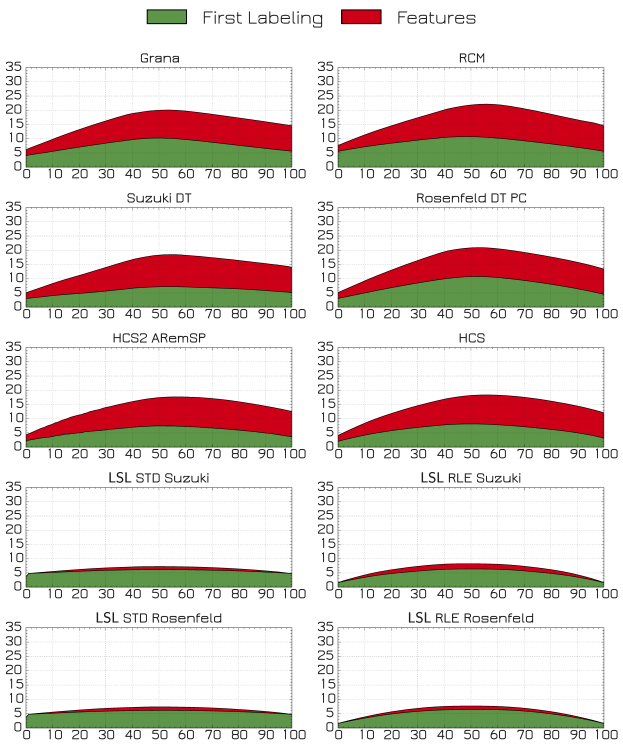

Fig. 24 Sequential algorithms: step analysis for $g = 4$ on one IvyBridge core, with random 2048×2048 images

algorithms execution times increases with g.
If we focus on the ratio (fig. 25) between $\text{LSL}_{\text{RLE-Rosenfeld}}$ and the best competitor – that is either HCS or HCS_2 – we can see that the ratio increases with g (having in mind that $g = 1$ is the worst configuration for a run-based algorithm like LSL). The average ratio is ×1.9 for $g = 1$, ×2.4 for $g = 4$ and reaches ×3.0 for $g = 16$.

For the FC step, the difference is bigger. These results enforce the fact that run-based coding and line-relative labeling (that avoids too many comparisons and tables updates) make *LSL* algorithms intrinsically better than all pixel-based algorithms.

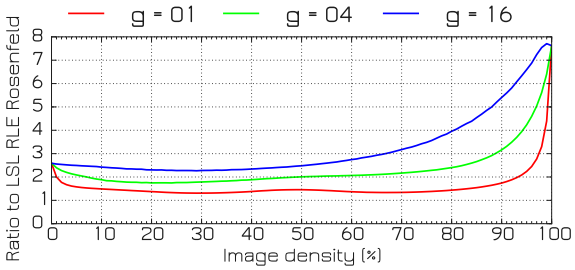

Fig. 25 $\text{LSL}_{\text{RLE-Rosenfeld}}$ vs best competitor (HCS_2 DT ARemSP/HCS) ratio on one IvyBridge core

Real case images: pie-graphs (fig. 26) show the time repartition for SIDBA4 images. The average SIDBA4 *cpp* compared to random images of granularity is in the interval $[8 \rightarrow 16]$ for pixel algorithm and $[12 \rightarrow 16+]$ for LSL versions. So random images with a granularity $g = 1$ are not representative of real use cases, they are just useful to find the synthetic/theoretical worst case. However, for a practical case, random images with $g \geq 4$ are more suitable. As for random images with high granularity, FC is invisible for LSL algorithms, whereas it exceed half of the whole processing time for all pixel-based algorithms.

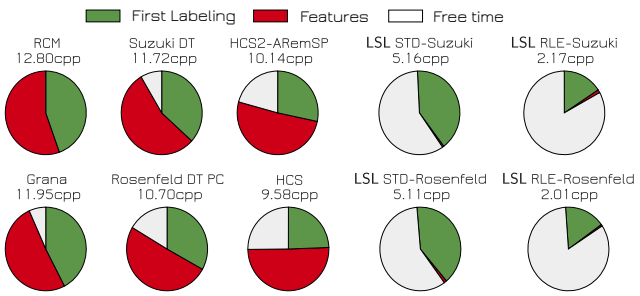

Fig. 26 Sequential algorithms: average *cpp* on SIDBA4 images with features computation on one IvyBridge core, pie-charts are normalized according to the slowest algorithm

4.3 Conclusions on sequential algorithms behavior

In this part, we exposed specificities of connected component labeling algorithms, using an extensive benchmark for in-depth analysis. That confirms that LSL algorithms are the fastest CCL algorithms. That highlights too that full run-based algorithms are necessary to reduce the features computation part and RLE variation provide a ×2.5 speedup over STD for SIDBA4.

In order to take another step in CCL algorithm acceleration, we will now focus on the multi-core adaptation of all these algorithms. That is the topic of the following parts.

5 Benchmarks results and analysis: parallel algorithms

5.1 Parallel benchmark on a 2×12-core Ivy-Bridge

Global benchmark: Figure 27 and table 5 provide the execution time in *cpp*. Two points can be noticed. First, for $g \in [1, 4]$, the equivalence management algorithm has a major impact. The Suzuki management generates a loss of performance around the percolation threshold – here, for $d \in [40\%, 60\%]$, whereas Union-Find (with or without optimization like DT and PC) does not induce such a dysfunction. The impact is very important for $g = 1$ and still observable for $g = 4$. Second point, LSL algorithms outperform all other algorithms.

Detailed Analysis: figure 30 focuses on the *cpp* of each algorithm step (first labeling in green, features computation in red and border merging in yellow) and highlights two points for $g = 1$.

First, the Suzuki equivalence management issue only affects the border merging step. For $g = 1$ (fig. 30) and $density = 43\%$, one can notice that the merge part for Suzuki management based algorithms take 44% or more (58%for RCM) of the total time. For Rosenfeld (UF) management, the pyramidal implementation is very efficient for all algorithms and does not represent more than

Table 5 *cpp* for granularity $g \in \{1, 2, 4, 8, 16\}$ on a 2 × 12-core Ivy-Bridge at 2.4GHz, with random 2048×2048 images

	granularity				
algorithms	g=1	g=2	g=4	g=8	g=16
$LSL_{RLE\text{-}Rosenfeld}$	0.86	0.45	0.28	0.20	0.16
$LSL_{RLE\text{-}Suzuki}$	1.02	0.50	0.29	0.20	0.16
$LSL_{STD\text{-}Rosenfeld}$	0.68	0.41	0.30	0.26	0.24
$LSL_{STD\text{-}Suzuki}$	0.83	0.45	0.31	0.26	0.24
HCS	1.68	1.07	0.72	0.59	0.53
HCS_2-ARemSP	1.37	0.93	0.71	0.60	0.54
Rosenfeld DT PC	1.60	1.05	0.77	0.64	0.57
Suzuki	1.74	1.10	0.73	0.60	0.53
Grana	1.89	1.06	0.79	0.65	0.59
RCM	2.13	1.22	0.87	0.72	0.67

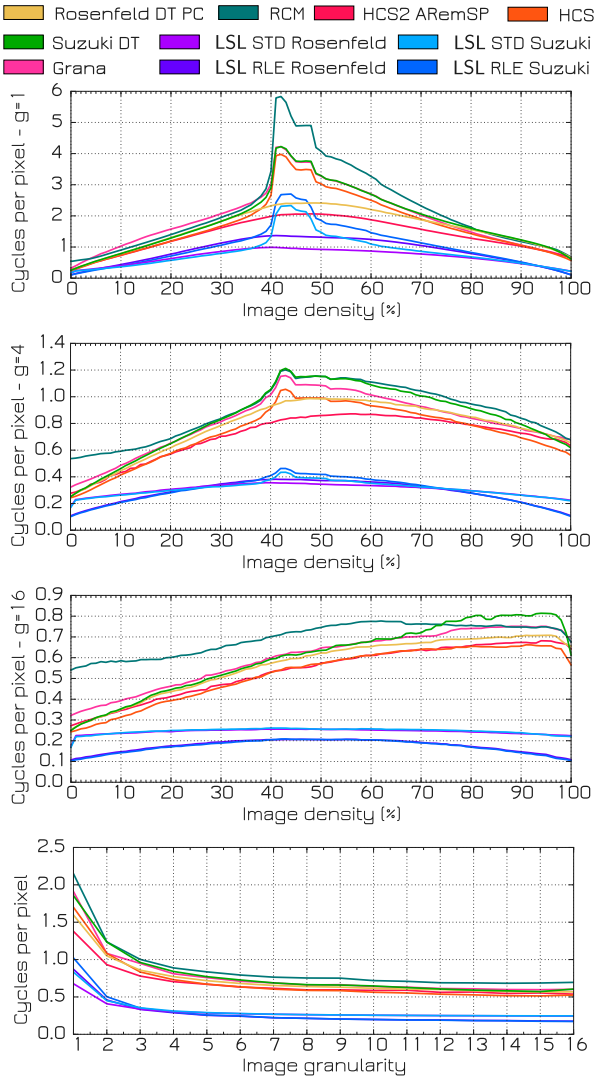

Fig. 27 *cpp* vs granularity for $g \in \{1, 4, 16\}$ and average *cpp* vs granularity with FC on a 2 × 12-core Ivy-Bridge, with random 2048 × 2048 images

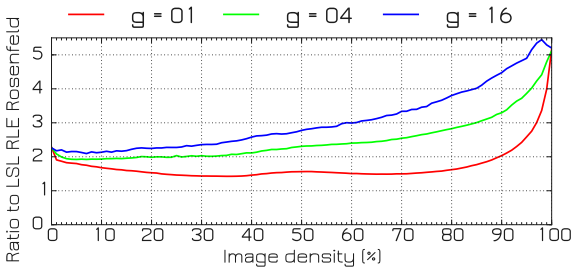

Fig. 28 $LSL_{RLE\text{-}Rosenfeld}$ vs best competitor (HCS) ratio on a 2×12-core Ivy-Bridge

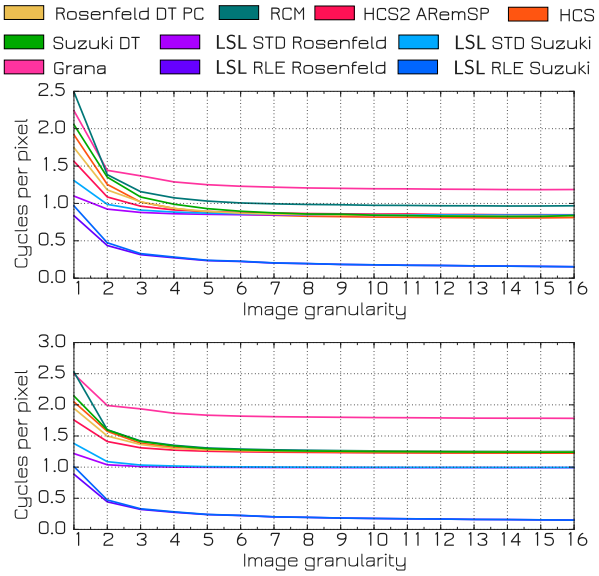

Fig. 29 Average *cpp* vs granularity on a 2×12-core Ivy-Bridge, for random images with size $\in \{4096 \times 4096, 8192 \times 8192\}$

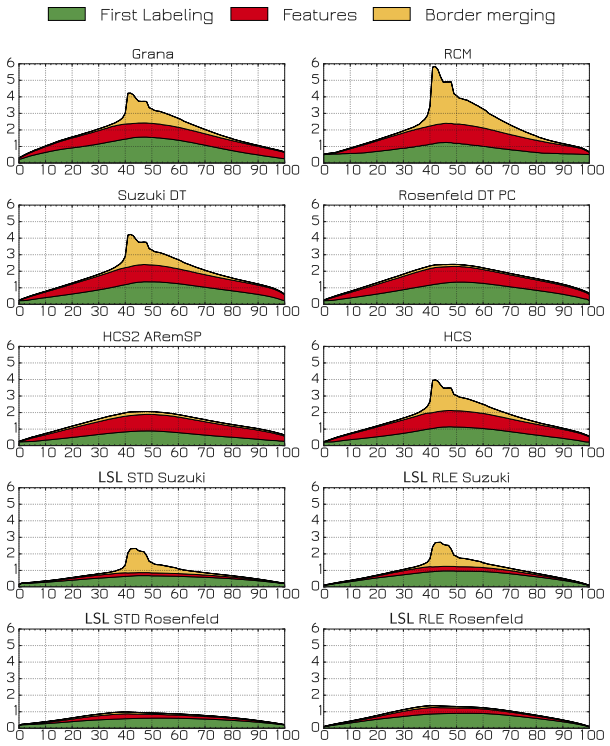

Fig. 30 Execution time (*cpp*) decomposition for $g = 1$ on a 2×12-core Ivy-Bridge, with random 2048×2048 images

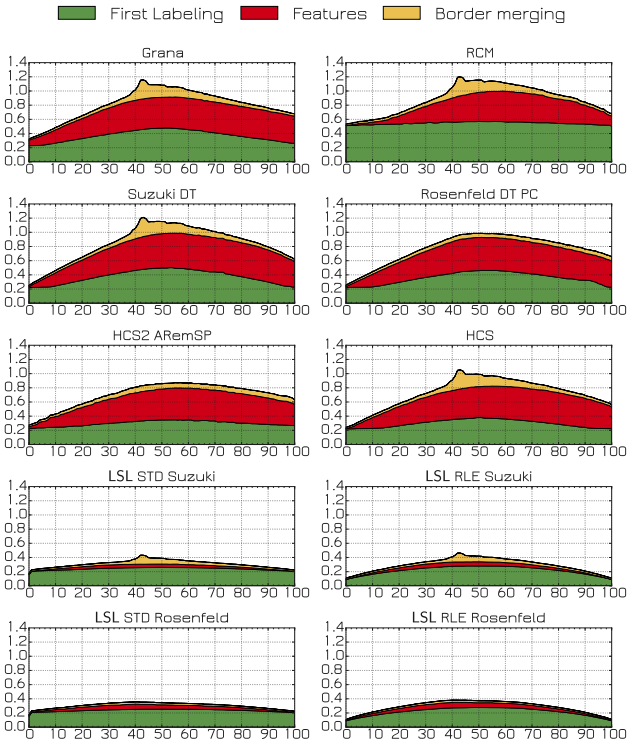

Fig. 31 Execution time (*cpp*) decomposition for $g = 4$ on a 2×12-core Ivy-Bridge, with random 2048×2048 images

8% of the total *cpp*. Second, the shape of the curves of the parallel versions is very similar to the sequential ones.

Real case images: Figure 34 highlights the efficient speedup of all algorithms and the evolution of the *cpp* of the different steps. On a 2×12-core Ivy-Bridge processor, LSL_{RLE} computes the SIDBA4 images in the average time of 0.15 ms achieving a speedup of ×13.4. For SIDBA4 the ratio between LSL_{RLE} and the best pixel-based algorithm is ×3.7.

Dependency to the image size: as figure 29 shows, all pixel-based algorithms slow down when the image size increase, especially Grana.

But in that case, LSL_{RLE} is even more efficient due to its RLE compression. Figures 32 and 33 show that, while all pixel-based algorithms and LSL_{STD} have a first labeling part uncorrelated with the image density that express a strong dependency to the memory performance, LSL_{RLE} has the same behavior than for smaller images.

Indeed, a constant first labeling part means that whether or not there is foreground pixel in the image, the computation time of the neighborhood remains constant.

5.2 Parallel Benchmark on a 4×15-core Ivy-Bridge

Global benchmark: Figure 35 and table 6 provide the execution time in *cpp*. Two points can be noticed. First, the dysfunctional behavior of the algorithms based on the Suzuki equivalence management around the percola-

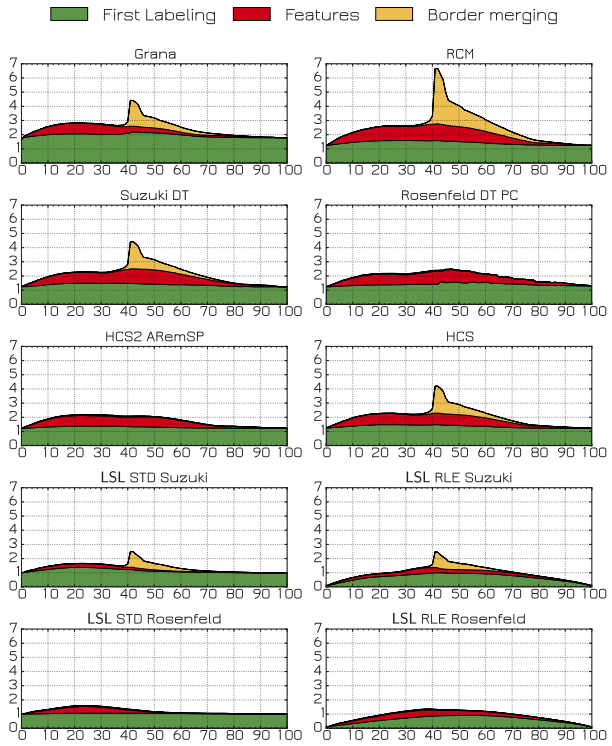

Fig. 32 Execution time (*cpp*) decomposition for $g = 1$ on a 2×12-core Ivy-Bridge, with random 8192×8192 images

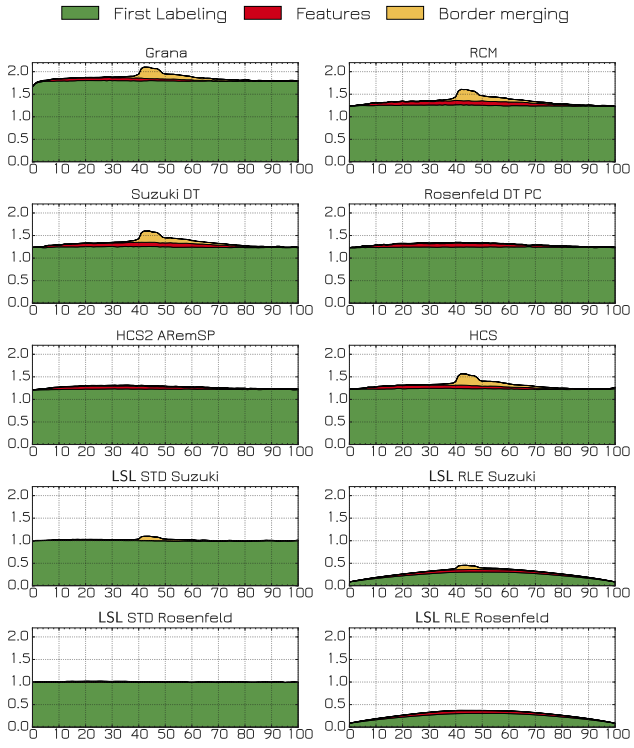

Fig. 33 Execution time (*cpp*) decomposition for $g = 4$ on a 2×12-core Ivy-Bridge, with random 8192×8192 images

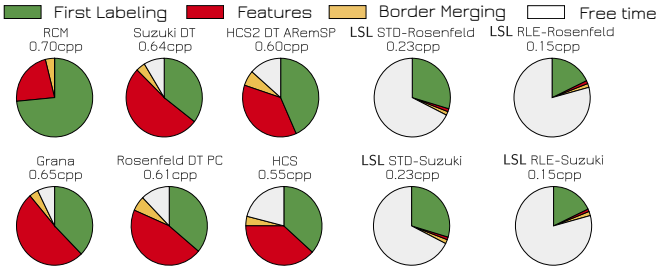

Fig. 34 Average *cpp* on SIDBA4 images with features computation on a 2×12-core Ivy-Bridge, pie-charts are normalized according to the slowest algorithm

Table 6 *cpp* for granularity $g \in \{1, 2, 4, 8, 16\}$ on a 4×15-core Ivy-Bridge, with random 2048×2048 images

	granularity				
algorithms	g=1	g=2	g=4	g=8	g=16
$\text{LSL}_{\text{RLE-Rosenfeld}}$	0.49	0.30	0.21	0.17	0.15
$\text{LSL}_{\text{RLE-Suzuki}}$	0.66	0.34	0.22	0.17	0.14
$\text{LSL}_{\text{STD-Rosenfeld}}$	0.41	0.28	0.22	0.20	0.18
$\text{LSL}_{\text{STD-Suzuki}}$	0.59	0.32	0.23	0.19	0.17
HCS	1.04	0.69	0.52	0.47	0.45
HCS_2 DTARemSP	0.77	0.59	0.54	0.51	0.50
Rosenfeld DT PC	0.85	0.63	0.52	0.48	0.47
Suzuki DT	1.07	0.71	0.53	0.48	0.46
RCM	1.39	0.78	0.62	0.57	0.55
Grana	1.11	0.71	0.60	0.53	0.51

tion threshold for $g \in [1, 4]$ is enforced with more cores. Second, the LSL_{RLE} is the absolute fastest above $g = 3$ and LSL_{STD} is faster than all the pixel-based algorithms in any conditions (from $g = 1$ to $g = 16$), meaning that no pixel based algorithm can achieve a CCL faster than LSL_{STD} whatever the image complexity is.

Detailed Analysis. Figures 37 and 38 provide two pieces of information. As for the 2×12-core configuration, the dysfunctional behavior of the algorithms based on the Suzuki equivalence management comes from the merge part. For $g = 1$ (fig. 30) and $density = 43\%$, one can notice that the merge part for Suzuki management based algorithms take 62% or more (72% for RCM) of the total time. For Rosenfeld equivalence management based algorithms, the merge part increases too but remains under 25% of the total *cpp*. Indeed, the first labeling and the FC parts were accelerated while the number of steps increase. The merge part is linked to the number of cores p by a $\lceil log_2(p) \rceil$ relation due to the pyramidal merging, so with this setup there is one more merging step than with the 2×12-core setup. One can notice that for the *Grana* algorithm, the FC part decrease less than for the others.

SIDBA4: The algorithms react differently when the core number increases. For RCM, the proportion of the first-labeling part increases whereas it decreases for the others pixel-based algorithms. As previously, the FC part for LSL is small, and one can notice that LSL_{STD} takes more advantage of the increase in the number of cores

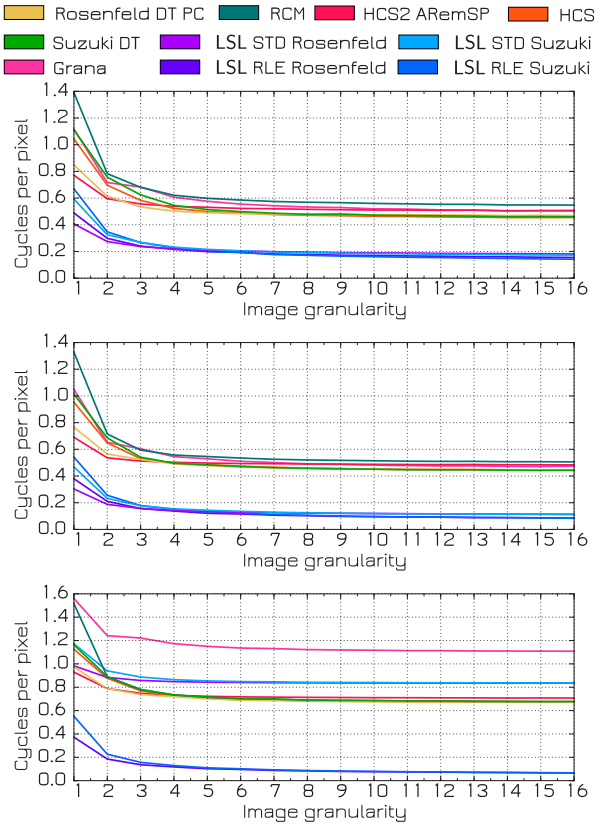

Fig. 35 Average *cpp* vs granularity on a 4×15-core Ivy-Bridge, for random images with size $\in \{2048 \times 2048, 4096 \times 4096, 8192 \times 8192\}$

than LSL_{RLE}. Due to the RLE compression, the LSL_{RLE} parallel efficiency decreases for a lower number of processors than for the other algorithms. However, the execution time remains better than for all the other competitors.

As said in the parallel algorithms section, the merge part prevents the algorithm to have a perfect scaling. We can observe here that in the case of "small" images compared to the number of available cores/threads

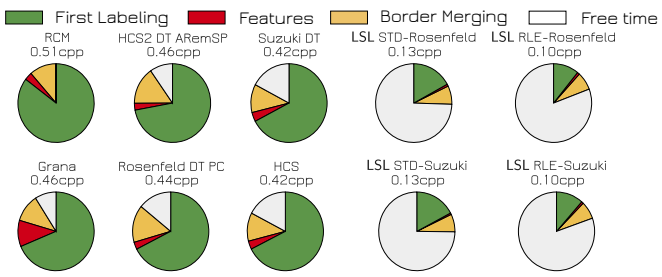

Fig. 36 Average *cpp* for SIDBA4 images with features computation on a 4×15-core Ivy-Bridge, pie-charts are normalized according to the slowest algorithm

(2048×2048 image with a 4×15-core machine) the merge part is becoming significant. For SIDBA4, the merge is proportionally longer than the FC (fig. 36 vs fig.34) and even quite as long as the first labeling for LSL_{RLE}.

Dependency to the image size: The figure 35 present the average *cpp* evolution for an image size of 2048×2048, 4096×4096, 8192×8192. As for the 2×12-core Ivy-Bridge, when the image size is 8192×8192 all the pixel-based algorithms have a constant first labeling part. However, on this machine when the image size is 4096×4096, the performance is better than for 2048×2048.

6 Performances evolution across number of cores and image size

To summarize the evolution of the LSL_{RLE} (with UF equivalence management) performance depending on the architecture and the image size, we provide the tables 7 and 8 where we report the execution time (in ms) and the throughput (in Gpixel/s). HCS_2 DT ARemSP has been selected as the fastest competitor. Table 9 provides the performance ratio between LSL and HCS_2 on the four machines.

There are two kinds of results depending on the socket number. For mono-socket machines, the max performance is reached for 2048×2048 images and sustained until 8192 ×8192 images. We can also notice the performance evolution of the i7 processor family: the Skylake is ×1.7 faster than the SandyBridge. Real-time (≤40 ms for a camera frame rate of 25 images/s) is achieved for all configurations, except the SandyBridge for 8192×8192 images with HCS_2 algorithm. On the same machines, the LSL is around ×3.7 faster than HCS_2. The sustainable throughputs of HCS_2 and LSL on the Skylake are respectively 1.8 and 6.6 Gpixel/s.

For multi-socket machines, the performances of LSL and HCS_2 diverge. For LSL performance increases with the image size whereas for HCS_2, it decreases. For LSL, the peak performance is reached for 4096×4096 images for the 2×12-core and 8192×8192 for the 4×15-core machine. LSL reaches 42.4 Gpixel/s while HCS_2 reaches only 5.8 Gpixel/s.

The performance ratio that was ×3.7 on 4-core Skylake reaches ×10.8 on 4×15-core IvyBridge. The reason of the HCS_2 performance loss is that – like all pixel-based algorithms – HCS_2 is *memory bound*. That is the reason why LSL on a 4-core skylake is faster than HCS_2 on a multi-socket machine.

As long as data (binary image of pixels, image of labels) fit in the cache, the performance can scale. But when it is no longer the case, there are too many cache

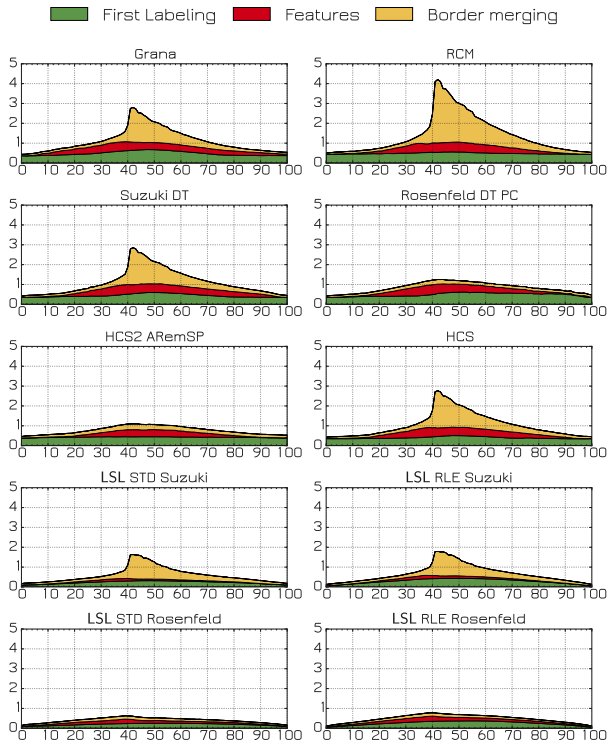

Fig. 37 Execution time (*cpp*) decomposition for $g = 1$ on a 4×15-core Ivy-Bridge, with random 2048×2048 images

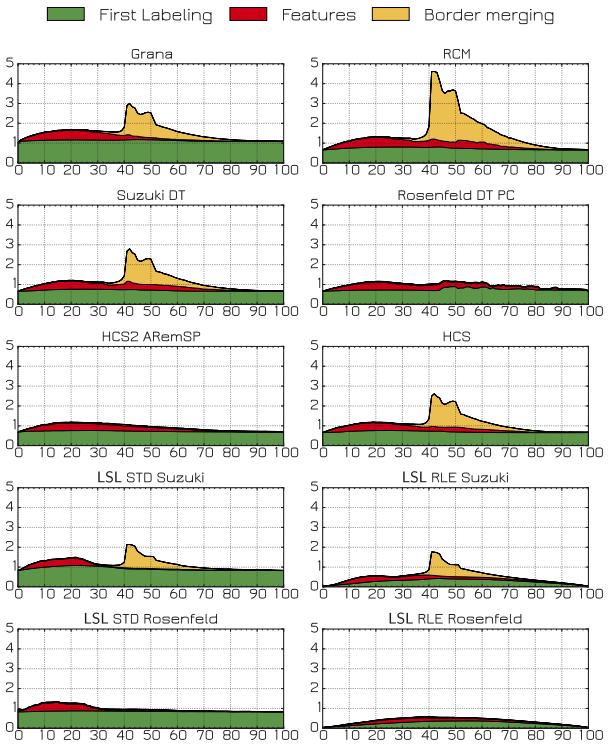

Fig. 39 Execution time (*cpp*) decomposition for $g = 1$ on a 4×15-core Ivy-Bridge, with random 8192×8192 images

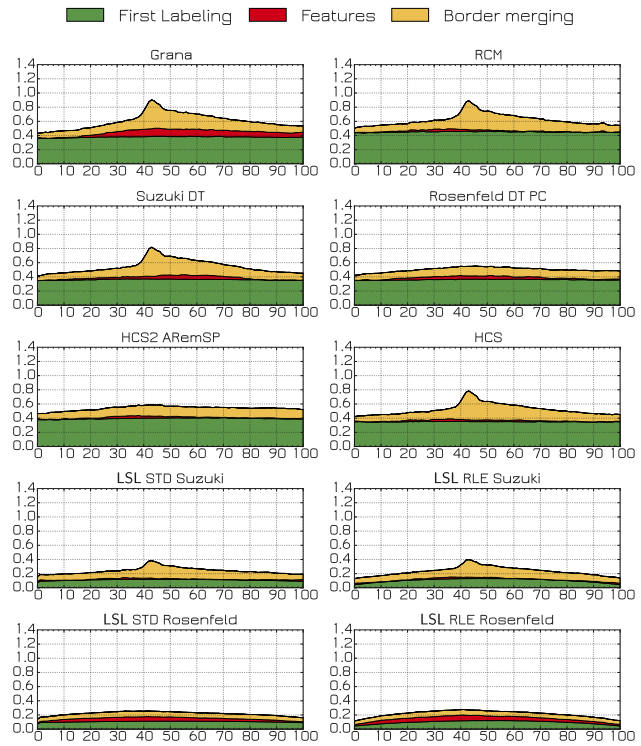

Fig. 38 Execution time (*cpp*) decomposition for $g = 4$ on a 4×15-core Ivy-Bridge, with random 2048×2048 images

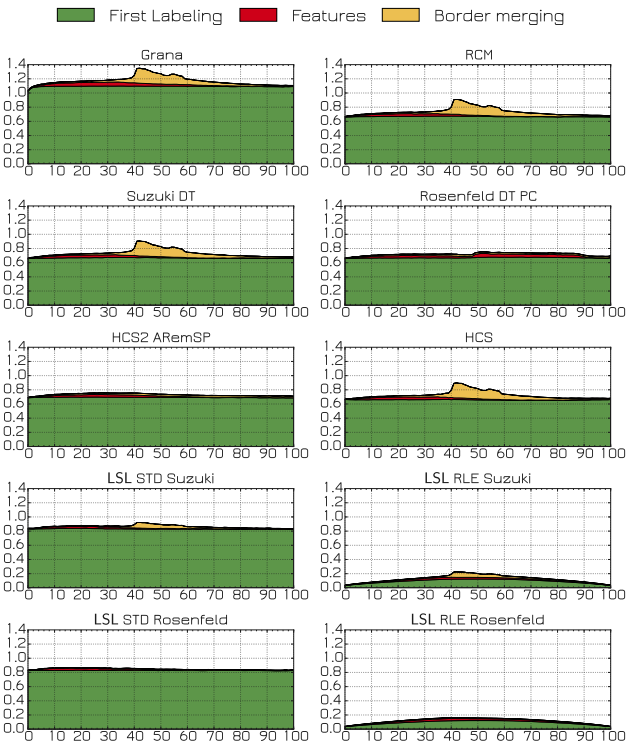

Fig. 40 Execution time (*cpp*) decomposition for $g = 4$ on a 4×15-core Ivy-Bridge, with random 8192×8192 images

misses resulting in a *cpp* increase. That is what we call a cache *overflow*. In order to highlight that issue, we have benchmarked LSL and HCS_2 in a different way. The have selected the quad-socket IvyBridge and we run the algorithms on one, two, three or four sockets (with OpenMP affinity in compact mode and still without hyperthreading). This benchmark is done for three densities: $g \in \{1, 4, 16\}$. If we look first at LSL performance (fig. 41), we can see that neither the granularity nor the image size has an impact of the *cpp*: all the curves are horizontal. In fact, there is a very small cache overflow but the image scale makes it not noticeable. If we now focus on the HCS_2 performance, we can observe cache overflows. For $g = 1$, the magnitude of cache overflow is small for 1,2 and 3 sockets. For four sockets, it is significant: for 10K images, the 4-socket *cpp* is the same than the 3-socket one (around 0.9 cycle/pixel). For $g = 4$ and 10K images, all the *cpp* are smaller and then put more stress on the memory busses: both 3-socket and 4-socket *cpp* converge to the 2-socket *ccp*. For $g = 16$ all the *cpp* are even smaller, the stress is even higher and all the *cpp* converge to the same value.

For HCS_2, using a high-end multi-socket machine is only efficient for images that fit in the cache (except for irrealistic random images with $g < 4$). Thanks to its RLE compression, LSL does not stress the memory buses too much, and then can scale on multi-socket machines.

Table 7 Execution time (in ms) and throughput (in Gpixel/s) of LSL_{RLE}, for 2048^2, 4096^2, 8192^2 and 3200×2400 SIDBA4 images

machines		random images 2048	4096	8192	SIDBA4
$SDB_{1\times4}$	t(ms)	1.1	4.3	17.4	2.0
	Gpixel/s	4.0	3.9	3.9	3.8
$SKL_{1\times4}$	t(ms)	0.65	2.6	10.2	1.0
	Gpixel/s	6.8	6.6	6.6	7.5
$IVB_{2\times12}$	t(ms)	0.31	1.05	4.2	0.48
	Gpixel/s	13.3	16.0	16.0	16.0
$IVB_{4\times15}$	t(ms)	0.24	0.50	1.6	0.27
	Gpixel/s	17.5	33.3	**42.4**	28.0

7 Conclusion

In this paper, we have presented the parallelization of the Light Speed Labeling algorithm for multi-core general purpose processors and compared it to our parallelized versions of State-of-the-Art algorithms. As far as we know, this paper is the first to present efficient parallelization of connected component labeling algorithms

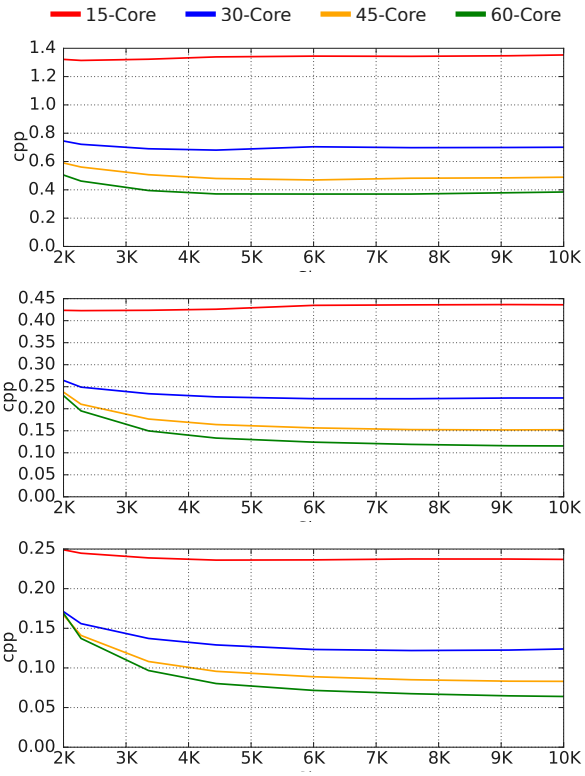

Fig. 41 *cpp* vs size for granularity for $g \in \{1, 4, 16\}$ on a 4×15-core Ivy-Bridge for LSL_{RLE}

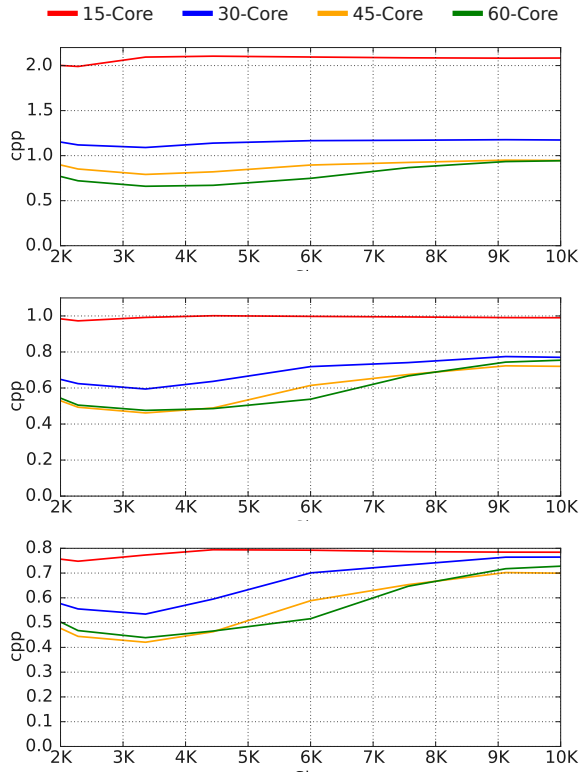

Fig. 42 *cpp* vs size for granularity for $g \in \{1, 4, 16\}$ on a 4×15-core Ivy-Bridge for HCS_2

Table 8 Execution time (in ms) and throughput (in Gpixel/s) of HCS_2 DT ARemSP, for 2048^2, 4096^2, 8192^2 and 3200×2400 SIDBA4 images

machines		random images 2048	4096	8192	SIDBA4
$SDB_{1\times4}$	t(ms)	3.7	14.8	58.0	7.8
	Gpixel/s	1.1	1.1	1.2	1.0
$SKL_{1\times4}$	t(ms)	2.4	9.4	37.4	4.9
	Gpixel/s	1.8	1.8	1.8	1.6
$IVB_{2\times12}$	t(ms)	0.94	5.9	34.1	1.9
	Gpixel/s	4.4	2.8	2.0	4.0
$IVB_{4\times15}$	t(ms)	0.75	4.3	17.0	1.3
	Gpixel/s	5.6	**5.8**	3.9	6.1

Table 9 Ratio of *cpp* between the fastest pixel-based algorithm and LSL_{RLE}, for $g = 16$

machines	Image size 2048	4096	8192
$SDB_{1\times4}$	× 3.5	× 3.4	× 3.3
$SKL_{1\times4}$	× 3.6	× 3.7	× 3.7
$IVB_{2\times12}$	× 3.0	× 5.7	× 8.1
$IVB_{4\times15}$	× 3.1	× 5.7	× **10.8**

on multi-core processors.

In order to be efficiently parallelized, we have explained and detailed why the features computation should be done on-the-fly during the first labeling, and the merge should be done in a pyramidal way.

The benchmarks have shown that for low granularity images, the Suzuki equivalence management algorithm has a major dysfunction and so, the classic Union-Find is the only choice for all parallel algorithms.

The paper enforces the results of the previous ones dealing with sequential comparisons: LSL is *intrinsically* faster than all other pixel-based algorithms as it is fully run-based (then producing less temporary labels than pixel-based algorithms) and uses RLE compression to reduce the amount of memory accesses and so the stress on memory busses.

Moreover, we show also that all the pixel-based algorithms are *memory bound* and so are inefficient and do not scale on multi-socket machines whereas LSL scales with the image size and the number of cores.

As a matter of fact, LSL is ×3.7 faster than its best competitor for 2048×2048 images and on a 4-core Skylake where it sustains a throughput of 6.6 Gpixel/s. LSL becomes up to ×10.8 faster than its best competitor on a 4×15-core Ivy-Bridge for 8192×8192 images and sustains a throughput of 42.4 Gpixel/s.

In future works, we will design new algorithms for accelerators like Xeon-Phi and GPUs. We will also consider the port of LSL on many-core architecture for which no efficient implementation has been yet proposed.

Acknowledgement

The authors gratefully acknowledge Francois Hannebicq from Intel France for his valuable help and the access to the high-end 4×15-core IvyBridge Xeon machine.

References

1. D. Bailey and C. Johnston. Single pass connected component analysis. In *Image and Vision New Zeland (IVNZ)*, pages 282–287, 2007.
2. G. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990.
3. L. Cabaret and L. Lacassagne. A review of world's fastest connected component labeling algorithms : Speed and energy estimation. In *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 1–8, 2014.
4. L. Cabaret and L. Lacassagne. What is the world's fastest connected component labeling algorithm ? In *IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 97–102, 2014.
5. F. Chang and C. Chen. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93:206–220, 2004.
6. C.-W. Chen, Y.-T. Wu, S.-Y. Tseng, and W.-S. Wang. Parallelization of connected-component labeling on TILE64 many-core platform. *Journal of Signal Processing Systems*, 75,2:169–183, 2013.
7. T. Cormen, C. Leiseirson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.
8. C. Grana, D.Borghesani, and R. Cucchiara. Fast block based connected components labeling. In *ICIP*, pages 4061–4064. IEEE, 2009.
9. S. Gupta, D. Palsetia, M. A. Patwary, A. Agrawal, and A. Choudhary. A new parallel algorithm for two-pass connected component labeling. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 1355–1362. IEEE, 2014.
10. R. Haralick. Some neighborhood operations. In *Real-Time Parallel Computing Image Analysis*, pages 11–35. Plenum Press, 1981.
11. L. He, Y. Chao, and K. Suzuki. A run-based two-scan labeling algorithm. In *ICIAR*, pages 131–142. LNCS 4633, 2007.
12. L. He, Y. Chao, and K. Suzuki. An efficient first-scan method for label-equivalence-based labeling algorithms. *Pattern Recognition Letters*, 31(1):28–35, 2010.
13. L. He, Y. Chao, and K. Suzuki. A new two-scan algorithm for labeling connected components in binary images. In W. Congress, editor, *Proceedings of the World Congress on Engineering*, volume 2, pages p1141–1146, 2012.
14. U. Hernandez-Belmonte, V. Ayala-Ramirez, and R. Sanchez-Yanez. Enhancing CCL algorithms by using a reduced connectivity mask. In Springer, editor, *Mexican Conference on Pattern Recognition*, pages 195–203, 2013.
15. J. Hopcroft and J. Ullman. Set merging algorithms. *Journal on computing*, 2,4:294–303, 1973.
16. W. W. Hwu, editor. *GPU Computing Gems*, chapter 35: Connected Component Labeling in CUDA. Morgan Kaufman, 2001.

17. M. Klaiber, D. Bailey, S. Ahmed, Y. Baroud, and S. Simon. A high-throughput FPGA architecture for parallel connected components analysis based on label reuse. In *International Conference on Field Programmable Technology (FPT)*, pages 302–305. IEEE, 2013.
18. M. Klaiber, L. Rockstroh, Z. Wang, Y. Baroud, and S. Simon. A memory-efficient parallel single pass architecture for connected component labeling of streamed images. In *International Conference on Field Programmable Technology (FPT)*, pages 159–165. IEEE, 2012.
19. L. Lacassagne and B. Zavidovique. Light Speed Labeling: Efficient connected component labeling on RISC architectures. *Journal of Real-Time Image Processing*, 6(2):117–135, 2011.
20. R. Lumia, L. Shapiro, and O. Zungia. A new connected components algorithms for virtual memory computers. *Computer Vision, Graphics and Image Processing*, 22-2:287–300, 1983.
21. N. Ma, D. Bailey, and C. Johnston. Optimised single pass connected component analysis. In *International Conference on Field Programmable Technology (FPT)*, pages 185–192. IEEE, 2008.
22. M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *Transactions on Modeling and Computer simulation*, 8(1):3–30, 1998.
23. M. Niknam, P. Thulasiraman, and S. Camorlinga. A parallel algorithm for connected component labeling of gray-scale images on homogeneous multicore architectures. *Journal of Physics - High Performance Computing Symposium (HPCS)*, 2010.
24. N. Otsu. A threshold selection method from gray-level histograms. *Transactions on System, Man and Cybernetics*, 9:62–66, 1979.
25. M. Patwary, J. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In L. . Springer, editor, *International symposium on experimental algorithms (SEA)*, pages 411–423, 2010.
26. C. Ronse and P. Dejvijver. Connected components in binary images: the detection problems. In *Research Studies Press*, 1984.
27. A. Rosenfeld and J. Platz. Sequential operator in digital pictures processing. *Journal of ACM*, 13,4:471–494, 1966.
28. R. Tarjan and J. Leeuwen. Worst-case analysis of set union algorithms. *Journal of ACM*, 31:245–281, 1984.
29. J. van Leeuwen and T. van der Weide. Alternative path compression techniques. technical report, University Utrecht, The Netherlands, 1977.
30. P. Vandewalle, J. Kovacevic, and M. Vetterli. Reproducible research in signal processing. *Signal Processing Magazine*, 26,3:37–47, 2009.
31. K. Wu, E. Otoo, and A. Shoshani. Optimizing connected component labeling algorithms. *Pattern Analysis and Applications*, 2008.
32. G. Ziegler. Connected components revisited on Kepler. In Nvidia, editor, *GPU Technology Conference*, pages 1–56, 2013.