Parallel Algorithm for Concurrent Computation of Connected Component Tree

P. Matas[1,2], E. Dokladalova[1], M. Akil[1], T. Grandpierre[1],
L. Najman[1], M. Poupa[2], and V. Georgiev[2]

[1] IGM, Unité Mixte CNRS-UMLV-ESIEE UMR8049, Université Paris-Est,
Cité Descartes, BP99, 93162 Noisy le Grand, France
{matasp, e.dokladalova, akilm, grandpit, l.najman}@esiee.fr
[2] Department of Applied Electronics and Telecommunications, University of West
Bohemia, Univerzitní 26, 306 14 Plzeň, Czech Republic
{pmatas, poupa, georg}@kae.zcu.cz

Abstract. The paper proposes a new parallel connected-component-tree construction algorithm based on line independent building and progressive merging of partial 1-D trees. Two parallelization strategies were developed: the parallelism maximization strategy, which balances the workload of the processes, and the communication minimization strategy, which minimizes communication among the processes. The new algorithm is able to process any pixel data type, thanks to not using a hierarchical queue. The algorithm needs only the input and output buffers and a small stack. A speedup of 3.57 compared to the sequential algorithm was obtained on Opteron 4-core shared memory ccNUMA architecture. Performance comparison with existing state of the art is also discussed.

1 Introduction

Computer vision systems are asked to furnish high performance and be flexible for a large variety of existing or possible applications. One of the global problems of the vision system design is how to achieve these two characteristics simultaneously. If the high performance is achieved by an optimization effort which means a kind of system specialization, it will (by definition) limit its flexibility.

The connected component tree (CCT) based image processing algorithms seem to be very promising from this point of view. They allow bridging the gap between low- and high-level processing implementations. They have been used for filtering [1, 5] as well as the image analysis: motion extraction [1], watershed segmentation [6, 2, 14], segmentation of astronomical images [3] or data visualization [13].

Fig. 1 shows typical stages of an application based on CCT. We can see the advantage of these methods: once the CCT is constructed, the processing is performed on the tree by graph transformation(s), and only one data structure is used from low-level to high-level processing. In addition, the graph transformations are applicable to any dimension (1D, 2D, 3D …).

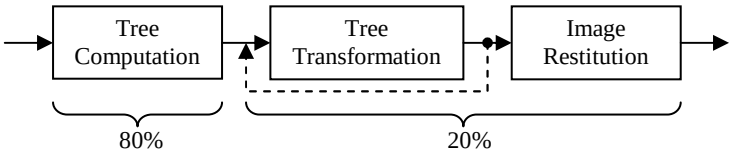

Fig. 1 Stages of a typical CCT-based application and times of execution

On the other hand, the main bottleneck is represented by the CCT construction, consuming about 80% of the application execution time (see Fig. 1), which is penalizing for a lot of practical applications.

In the past, several algorithms have been proposed in order to solve this problem. In majority of cases they remain sequential and the improvement relies on fast data structures (FIFO-like) [1] or on optimization of computational complexity [2]. Some parallelization effort has been done on shared-memory computers by [10, 5]. However, if the obtained performances are interesting they are insufficient for real-time and not adapted to the embedded systems [8].

In this paper, we present new parallel algorithm for computation of the CCT. The algorithm proceeds line by line (inspired by [4]). Then the line trees are progressively merged. Since the line-based CCT construction is extremely fast, a new parallelization strategy is needed for concurrent implementation. This paper presents and evaluates two parallelization strategies: i) parallelism maximization, ii) limited communication among computing blocks. In addition, our algorithm makes use of memory-aware data structure hence it is more suitable for embedded system implementation.

The paper is organized as follows. Section 2 shortly discusses existing sequential and parallel algorithms and analyses their computational complexity. In Section 3, we give basic mathematical background. Section 4 presents the new proposed algorithm, the parallelization approaches and their theoretical evaluation. Finally, some experimental results on real multi-processor systems are presented in Section 5.

2 State of the art

Three main classes of sequential algorithms exist in the literature:

- *Flooding-based algorithms* [1]: processing starts from the image pixel having the lowest level. A depth-first traversal of tree components, similar to flood-fill, is performed. In general, the flooding process relies on the use of ordered data structures: i) hierarchical queues, unusable for floating-point pixel representation, ii) priority queues, inefficient from the time and memory point of view.
- *Emerging-based algorithms*: image pixels are processed in decreasing order of luminosity. It requires prior pixel sorting which could be done efficiently. The emerging components are processed as disjoint sets of pixels, based on Tarjan's Union-Find algorithm [7]. In [2], both total path compression and weighting are used to speedup the disjoint set algorithm and the algorithm complexity is quasilinear. [3, 8, 9] use only total path compression in order to save memory.

- *1-D algorithms*: it is a special category where the CCT is built on 1-D signal. Thus, the pixel processing ordering is unnecessary and the tree can be built in linear time [4]. These algorithms are extremely fast, but they cannot process 2-D data. However, if tree merging is added [10], they are usable for any dimension, so we have chosen this approach.

In order to accelerate the execution time, Meijster [10] studies the first implementation of CCT computation on shared memory machines. Recently, Wilkinson [5] has published a modification of the Meijster's approach and has demonstrated the computation performance on 3-D data. The principle consists of image division into regular domains. A modification of Salembier's algorithm [1] is used to build a CCT of each domain independently. Then, the trees of the domains are merged in a binary-tree fashion. See **Table 1** which summarizes the complexity analysis of the existing CCT computation algorithms (sequential and parallel).

Table 1. Complexity analysis. N is the total number of pixels in the image, G is the number of grey levels of the image, α is a very slow-growing "diagonal inverse" of the Ackermann's function, $\alpha(10^{80}) \approx 4$. For the Wilkinson's algorithm, P is the number of processes

	Time complexity	**Memory requirements calculation hints**	**Data types**
Salembier [1]	$O(NG)$	$4N + 3G$ + stack	small int
Najman-Couprie [2]	$O(N\,\alpha(N))$	$7N + G$ + stack	int/float
Berger [3]	$O(N \log N)$	$4N$ + stack	int/float
Levillain [12]	$O(NG \log N)$	$2N$ + stack	int/float
Menotti (1D) [4]	$O(N)$	$3N + G$ + stack	int/float
Wilkinson (3D) [5] (building + merging)	$O(NG/P + N^{2/3}G \log N \log P)$	$3N + 3G$ + stack	small int

3 Mathematical background

Let us consider a function $f: \mathbf{Z}^2 \rightarrow \mathbf{R}$ an image associated with 4-connectivity (can be generalized to any dimension and connectivity), where below $V = \text{supp}(f)$ denotes the set of points (pixel coordinates) of the image and $\mathbf{R}$ is the real number set. We call a *k-level connected component* ($C \subset V$) if all of the following conditions are met:

1. C is connected with regard to the 4-connectivity.
2. $\forall\, x \in C:\ f(x) \geq k$
3. No other point $y \in V$ can be added to C without violating the conditions 1 or 2.

We call $h(C) = \min\{f(x)\ ;\ x \in C\}$ the *altitude* of component C. For each point $x \in V$ we define $C_f(x)$ as the component of the image f, which has the altitude $h(C_f(x)) = f(x)$ and which contains the point x.

The connected components of the image may be organized, thanks to the inclusion relation, to form a rooted tree structure called ***connected component tree***: For each

two components C_1, C_2 of the image f, we say that C_1 is the parent of C_2 precisely if $C_2 \subset C_1$ and there is no other component C_3 of image f, such that $C_2 \subset C_3 \subset C_1$.

We call the subset $C' = \{ x \in C ; f(x) = h(C) \}$ of component C as the *core* of the component C. It is the set of points of the component C, which belong to no descendant of C. Note that each point of V belongs to exactly one component core.

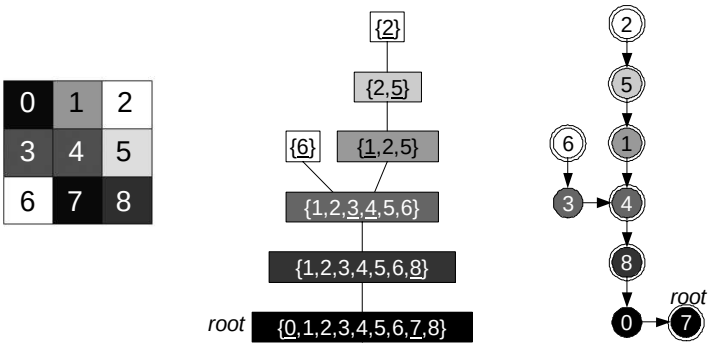

Fig. 2 Component trees**:** a) an example image f with points numbered in scan-line order; white background represents the highest level of f. b) the component tree of the image f; the points of the component's core are underlined. c) the point-tree of the image f ; level roots are marked with double circles

In our algorithm we use a memory-aware representation [10], [5] of the component tree called ***point-tree*** [11]. The point-tree (PT) of the image f is a rooted tree, whose nodes are points of the image f and where points of each component of the image f form a subtree of PT. Note that there may be more than one valid point-tree corresponding to a given image.

The edges of PT are represented by parent pointers stored in an array named $par : V \rightarrow V \cup \{\perp\}$ where $\perp$ stands for null pointer. For each point $x \in V$, $par[x]$ is the parent of x if x is not the root of PT and $par[x] = \perp$ if x is the root of PT. We define $f(\perp) = -\infty$.

It can be shown that for each point $x \in V$:

- $f(par[x]) \leq f(x)$
- If $f(par[x]) < f(x)$ then $C_f(par[x])$ is the parent component of $C_f(x)$ and x is the root of PT's subtree composed of points of $C_f(x)$. We say that x is the level root of the component $C_f(x)$ and of all its core's points.
- If $f(par[x]) = f(x)$ then $C_f(par[x]) = C_f(x)$ and we say that x is not a level root.

4 New algorithm description

The algorithm proceeds in two steps: a partial point tree is computed independently for each line (Algorithm 1) and then the partial trees of neighboring lines have to be merged together.

Partial point-tree computation

Menotti's 1-D algorithm was modified to produce a PT, which is suitable for merging and parallelization, instead of a component tree structure and component mapping. The algorithm processes the points of the line in a single linear scan from left to right and stores the result as the point-tree to allow a subsequent merging. The algorithm uses only a stack (LIFO) to store the points, whose parent pointers could not be determined yet. The stack supports these operations:

- **`StackPush(`*x*`)`** : Adds the point *x* to the top of the stack.
- **`StackPop()`** : Removes one point from the top of the stack.
- **`StackLast()`** : Returns the point at the top of the stack without stack modification or $\perp$ if the stack is empty.
- **`StackEmpty()`** : Returns true if the stack is empty.

The first point of the line is a level root, because it is surely the leftmost point of some component core. Variable *r* is initialized to this first point and the scan starts from the second point of the line. Note that the leftmost point of each component core is treated as a level root.

The following invariant holds during the scan: before and after each iteration of the scan, i) the variable *r* holds the level root of the last processed point and ii) the stack contains all level root ancestors of *r* encountered so far, ordered by their levels, the highest level on the top of the stack.

The parent pointer of each point is assigned exactly once, just before the point is dropped from the stack; from the variable *r* and the variable *p* used during the scan. The parent pointer is always set to point to a level root, so a perfectly compressed point-tree is produced.

A new point *p* is processed in each iteration of the scan. Its level $f(p)$ is compared to the level $f(r)$. There are three possibilities:

- $f(r) < f(p)$: point *p* is the leftmost point of a new component core, so it is a level root. Point *r* is a (possibly indirect) ancestor of *p*, because *r* is the level root of its left neighbor. Current *r* is pushed to the stack and *p* is set as the new value of *r*. No point is dropped, so no parent pointer is set.
- $f(r) = f(p)$: point *p* belongs to the same component core as the last processed point and *r* is its level root, so *r* is assigned to $par[p]$ and *p* is dropped.
- $f(r) > f(p)$: the component represented by point *r* is completed and its parent has to be determined. Let *q* be the point on the top of the stack. Either *p* or *q* is the parent of *r*, depending on their levels, so $f(p)$ is compared to $f(q)$. Again, there are three possibilities:
  - The stack is empty or $f(q) < f(p)$: point *p* is the leftmost point of a new component core, so it is a level root. It is also the parent of *r*, so *p* is assigned to $par[r]$ and *r* is dropped. Point *p* is set as the new value of *r*.
  - $f(q) = f(p)$: the points *q* and *p* belong to the same component core. The point *q* is its level root and the parent of *r*, so *q* is assigned to both $par[p]$ and $par[r]$ and both *p* and *r* are dropped. Point *q* is removed from the stack and set as the new value of *r*.
  - $f(q) > f(p)$: point *q* is the parent of *r*, so it is assigned to $par[r]$ and *r* is dropped. Point *q* is removed from the stack and set as the new value of *r*. But

now, $f(r)$ is still greater than $f(p)$. This means that the component represented by the new value of r is completed and its parent has to be determined now. This is done by repeating the decision process with the same p, the new value of r and the new state of the stack.

After the scan finishes, r contains the level root of the last point of the line and all its ancestors are in the stack. They are removed from the stack one by one and parent pointers are set accordingly. The last point removed from the stack is the tree root.

Algorithm 1 1-D algorithm for computation of point-tree

$V = \{0 \dots W-1\} \times \{0 \dots H-1\}$
$line \in \{0 \dots H-1\}$ is the number of the line to be processed
$V_{line} = \{0 \dots W-1\} \times \{line\} = \{ (i, j) \in V ; j = line \}$ is the set of points of one line
Input: image $f : V_{line} \to \mathbf{R}$
Output: point-tree $par : V_{line} \to V_{line} \cup \{\perp\}$

```
procedure build1D(line : integer) =
  1     var r : point := (0, line) ;
  2     for p : point := (1, line) ... (W - 1, line) do
  3          if f(r) < f(p) then
  4               StackPush(r) ;
  5               r := p ;
  6          elsif f(r) = f(p) then
  7               par[p] := r ;
  8          else loop
  9                    var q : point := StackLast() ;
 10                    if f(q) < f(p) then
 11                         par[r] := p ; r := p ;
 12                         break ;
 13                    elsif f(q) = f(p) then
 14                         par[r] := q ; par[p] := q ; r := q ;
 15                         StackPop() ; break ;
 16                    else
 17                         par[r] := q ; r := q ;
 18                         StackPop() ;
 19     end; end; end; end;
 20     while StackEmpty() = false do
 21          par[r] := StackLast() ; r := StackLast() ;
 22          StackPop() ;
 23     end;
 24     par[r] := ⊥ ;  (* r is root *)
end;
```

Merging of partial point-trees

Generally speaking, we take two adjacent point-trees as input and modify their parent pointers to create a single point-tree. The merging operation is done in procedure `connect`(x, y), which is executed for each pair of points x and y, where x is in the first point-tree, y is in the second point-tree and x and y are neighbors in 4-connectivity sense. Procedure `connect` follows the parent pointer paths from x and y respectively to the root of the tree and changes the parent pointers to form a single path. The new path includes nodes visited along both paths in correct order of levels. When the two parent pointer paths meet, the procedure is terminated. Note that the principle is the same as used in [5], with some simplifications.

Algorithm 2 Merging process of two adjacent point-trees

The first point-tree starts at line a and ends at line $border - 1$, the second point-tree starts at line $border$ and ends at line b ($0 \le a < border \le b < H$)
$V_{a,b} = \{0 \dots W-1\} \times \{a \dots b\} = \{ (i, j) \in V ; a \le j \le b \}$ is the set of points of the two trees
Input: image $f : V_{a,b} \to \mathbf{R}$; point-tree $par : V_{a,b} \to V_{a,b} \cup \{\perp\}$
Output: point-tree $par : V_{a,b} \to V_{a,b} \cup \{\perp\}$

```
 1    procedure levroot(x : point) returns point =
 2         if f(x) = f(par[x]) then
 3              par[x] := levroot(par[x]) ; return par[x] ;
 4         else
 5              return x ;
 6    end; end;

 7    procedure connect(x, y : point) =
 8         x := levroot(x) ;  y := levroot(y) ;
 9         if f(y) > f(x) then swap(x, y) ; end;
10         while x ≠ y do
11              if par[x] = ⊥ then
12                   par[x] := y ;  x := y ;
13              else
14                   var z := levroot(par[x]) ;
15                   if f(z) > f(y) then
16                        x := z ;
17                   else
18                        par[x] := y ; x := y ; y := z ;
19    end; end; end; end;

20    procedure merge(border : integer) =
21         for all neighbors x, y between lines (border – 1) and border do
22              connect(x, y) ;
23    end; end;
```

4.1 Concurrent implementation

Parallelism Maximization Strategy. The point-trees of individual lines are independent after the building; there are no parent pointer links between the nodes of different lines. To create a complete point-tree of the image, which consists of H lines, each two adjacent lines have to be merged. When two lines merging started, two originally independent point-trees become connected and a larger point-tree is formed. No other process should work on the two trees being merged. To achieve maximal parallelism, it is the best to merge point-trees of the same or similar sizes, whenever possible. For this reason, each merge step of our algorithm merges two point-trees, which consist of 2^k lines each, into one point-tree of 2^{k+1} lines. Only if the height H of the image is not a power of two, then the point-tree, which includes the last line of the image, may be smaller than the point-tree which it is merged with.

The algorithm consists of H building operations and $(H - 1)$ merging operations. Their dependencies are shown in figure 3. Each building operation accepts one line of the image, i.e. $W \times 1$ pixels, n bits per pixel as the input and produces the point-tree of the same size, m bits per parent pointer.

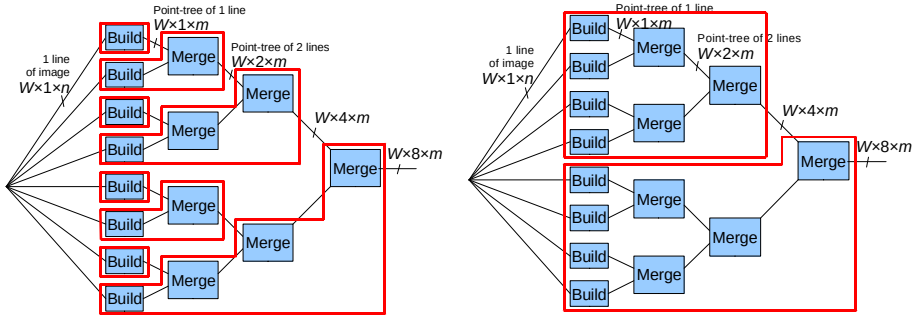

Fig. 3. Data dependency graph. a) Parallelism maximization strategy, b) Communication minimization strategy. The enclosing polygons show which operations are done by a single process

The algorithm is executed in $P \in \mathbf{N}$ (natural number) independent parallel processes (threads). Numbered lines in a queue are fetched by the individual processes. The queue is represented by a shared *next_line* counter. The counter has to be read first and then incremented on each fetch. This is done by the synchronized procedure `get_next_line`, where "Synchronized" means that if two processes attempt to enter the procedure at the same time, one of them has to wait until the other leaves the synchronized procedure. When a line is fetched by the process, the process performs the building operation. Then the process proceeds to the cascade of merging operations following the path from the build job to the right in the data dependency graph. The process performs as many merging operations as possible without waiting. Each merging job requires two inputs (two point-trees). One of the inputs is carried by the process which is about to perform the merging job. The other input has to be supplied by another process. The boolean array *block_ready* is used to keep track of merging jobs which are ready to be done. An element of *block_ready* is true if the corresponding merging job has at least one input ready. The process, which attempts the merging, calls the synchronized procedure `test_merge`. This procedure reads the previous value of the corresponding element of *block_ready*, which it will return, and sets it true. If the read value is true (the process is the second one to attempt this merging job), it means that the other input of the merging job is ready and the job is performed by the process. If the read value is false (the process is the first one to attempt the merging job), the other input is not ready yet, so the process leaves the cascade of merging operations and fetches next unassigned line to be built. We can see that each building job will be done exactly once, because each of them is fetched by one process. Also each merging job will be done exactly once, because exactly one process attempts each merging job as the second. If a process attempts to fetch next unassigned line, but there are no more lines left, the process exits. The algorithm ends when all processes exit.

Algorithm 3 Concurrent implementation

Input: image $f : V \rightarrow \mathbf{R}$
Output: point-tree $par : V \rightarrow V \cup \{\perp\}$

```
 1    var next_line : integer := 0 ;
 2    var block_ready[1..H] : boolean ; (elements initialized to false)
 3    synchronized procedure get_next_line() returns integer =
 4         if next_line < H then
 5              return next_line++ ;
 6         else
 7              return -1 ;
 8    end; end;

 9    synchronized procedure test_merge(border : integer)
      returns boolean =
10         var result : boolean := block_ready[border] ;
11         block_ready[border] := true ;
12         return result;
13    end;

14    process line_parallel_tree() =
15         while (var line := get_next_line()) != -1 do
16              build1D(line) ; (* Build partial tree*)
17              var block_no := line ;
18              var block_size := 1 ;
19              while block_size < H do (*Merge as deep as possible*)
20                   block_no := block_no div 2 ;
21                   block_size := block_size * 2 ;
22                   var border := (block_no + 1/2) * block_size;
23                   if border < H then
24                        if test_merge(border) = true then
25                             merge(border) ;
26                        else
27                             break ;
28    end; end; end; end; end;
```

Communication Minimization Strategy. The algorithm is executed in $P \in \mathbf{N}$ (natural number) independent parallel processes (threads). The entire image is divided into P roughly equal-sized blocks of approximately (H / P) lines (the numbers of lines per block have to be integers). The data dependency graph is the same as for the line parallel algorithm only if the block size is a power of 2 lines (Fig. 3b). If the block size is not a power of 2 then the data dependency graph is slightly different. Each block is assigned to one process. The process builds and merges all lines of the block into the point-tree of the block, just like the line parallel algorithm running in a single process does. It means that after building of a line, as many merging steps as possible are done. Synchronization is no more necessary, because the job execution order is known in advance. After the block was built and merged, it is merged with the other blocks. If we consider processing of the block as a single building operation, the data dependency graph applies to the merging of the blocks as for the line parallel algorithm. One of the processes, which reach a merging operation, terminates and the other process proceeds with merging. The final point-tree is done when last two blocks are merged.

4.2 Time complexity and memory requirements

The time complexity of the building operation (procedure `build1D`) was already analyzed in [4] and equals $O(W)$.

For its memory requirements, let us consider that: no point can be inserted twice into the stack; no two points simultaneously present in the stack can have the same level; the last point is never inserted into the stack; the points with the highest level are never inserted into the stack. Thus, the maximum stack size is $\min(W-1, G-1)$.

The merging operation time complexity was already analyzed by Wilkinson [5] and is $O(WG \log N)$. The merging operation does not need any additional memory except input and output buffers. The recursive procedure `levroot` needs a stack to store the points for path compression. It is possible to avoid the need for stack by implementing this procedure without recursion, but the parent pointers have to be read twice.

The total time complexity of the algorithm is dominated by the merging operations and is $O(NG \log N)$ for the whole image with N pixels and G grey levels.

The total memory requirements are input buffer of N pixels, output buffer of N memory pointers and a stack of size $\min(W-1, G-1)$.

5 Experimental results

The algorithm described in the previous section was implemented in C and compiled using Visual C++ 2005 Express Edition. The benchmarks were run on two different computers with 2 and 4 CPU cores:

- 2-core machine: Portable computer with Intel Pentium Dual-Core T2330 @ 1.6 GHz, 1 GB RAM, Windows XP Professional SP2
- 4-core machine: 2× dual-core AMD Opteron 280 @ 2.4 GHz, 16 GB ccNUMA RAM, Windows Server 2003 R2

We measured the wall clock times (real time elapsed from start to end of the task) of the two parallelization strategies (parallelism maximization and communication minimization) for 1, 2, 4, 8, 16 threads on a test image of size 784×576 px (8 bits).

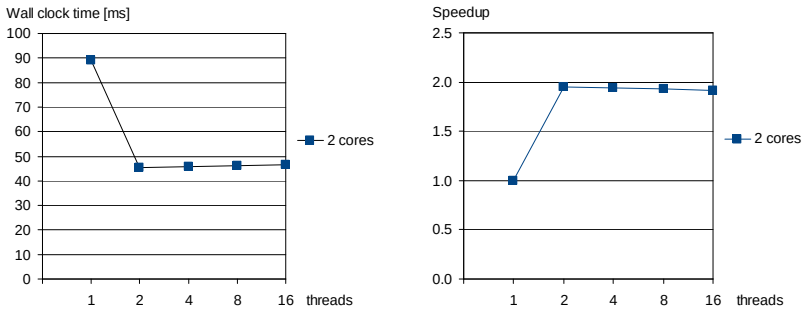

Fig. 4. Parallelism maximization strategy timings: a) Wall clock time versus number of threads, b) Speedup with respect to the sequential algorithm versus number of threads.

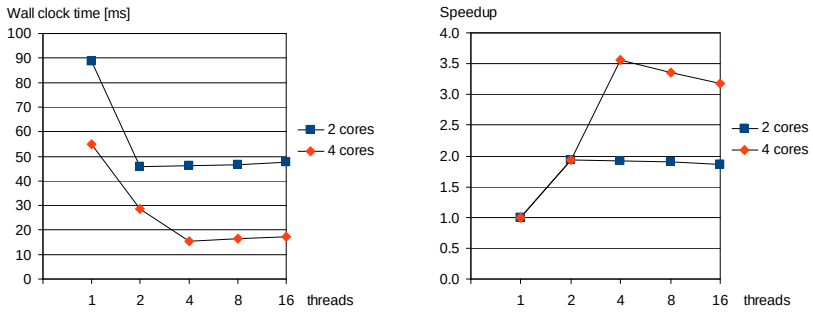

Fig. 5. Communication minimization strategy timings: a) Wall clock time versus number of threads, b) Speedup with respect to the sequential algorithm versus number of threads.

On the 2-core machine, we get maximal speedup 1.95 from 89.0 ms to 45.6 ms for 2 threads for the parallelism maximization strategy and the results for the communication minimization strategy are very similar. On the 4-core machine, we get maximal speedup 3.57 from 55.1 ms (0.122 μs/pixel) to 15.4 ms (0.034 μs/pixel) for 4 threads for the communication minimization strategy. Maximum speedup is obtained when number of cores and threads match. Theoretical maximal speedup is 2.0 and 4.0 for 2 and 4 cores respectively. The numbers measured are less than theoretical because of the workload imbalance and overhead.

The time increases roughly linearly with increasing width of the image.

Results obtained by Wilkinson on the 2× dual-core Opteron machine with 8 GB RAM operating on a 512^3 8-bit volumetric data set are as follows: Wall clock time for 1 thread was 73 s (0.54 μs/voxel), wall clock time for 64 threads was 13.7 s (0.102 μs/voxel), that gives the speedup 5.3.

6 Conclusions

We have designed a new parallel algorithm for CCT construction. The algorithm design was focused on maximal parallelism. Both parallelization strategies minimize the traffic between the execution units and the memory. The communication minimization strategy also minimizes the traffic among the execution units, but its disadvantage is the fixed process workload assignment. The parallelism maximization strategy has advantage when the data stream is of serial form from the sensor, because it takes the lines of the image from top to bottom, one by one. The parallelism maximization strategy provides a usable workload balancing, but depends on a significant amount of synchronization among processes.

The Salembier-based Wilkinson's algorithm is slow when the difference of neighbor levels is large. The new algorithm does not depend on absolute levels, so it is able to process any pixel data type (including floating point representation) without large performance loss, thanks to not using a hierarchical queue.

Wilkinson obtains the biggest speedup for number of threads far higher than number of physical CPUs. He explains that by improvement of data locality due to

dividing of the image into smaller blocks. This does not apply to our algorithm, because it proceeds by lines of size independent of number of threads, so best speedup is obtained when number of threads equals number of physical cores. Additional increase of thread count extends the time due to thread management overhead.

The new algorithm takes approximately 3 times less time to process one image element compared to Wilkinson's algorithm.

In the future we will evaluate a tradeoff between the presented strategies and optimize it for the Cell platform, a powerful multiprocessor on a chip.

References

1. P. Salembier, A. Oliveras, and L. Garrido. Anti-extensive connected operators for image and sequence processing. *IEEE Trans. on Image Proc.*, 7(4):555–570, April 1998
2. L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Transactions on Image Processing*, 15/11: 3531–3539, 2006
3. C. Berger, Th. Géraud, R. Levillain, N. Widynski, A. Baillard, E. Bertin. Effective component tree computation with application to pattern recognition in astronomical imaging. ICIP 2007
4. D. Menotti, L. Najman and A. de Albuquerque Araújo. 1D Component Tree in Linear Time and Space and its Application to Gray-Level Image Multithresholding. *Proceedings of the 8th International Symposium on Mathematical Morphology*, Rio de Janeiro, Brazil, Oct. 10–13, 2007, MCT/INPE, v. 1, p. 437–448
5. M. H. F. Wilkinson, Hui Gao, W. H. Hesselink, Jan-Eppo Jonker and Arnold Meijster. Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines. Submitted for *Transactions on Pattern Analysis and Machine Intelligence*, 2007
6. M. Couprie, L. Najman and G. Bertrand. Quasi-linear algorithms for the topological watershed. *Journal of Mathematical Imaging and Vision*, Volume 22, Issue 2 - 3, May 2005, Pages 231 - 249
7. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22: 215–225, 1975
8. N. Ngan, F. Contou-Carrère, B. Marcon, S. Guérin, E. Dokládalová, M. Akil. Efficient hardware implementation of connected component tree algorithm. Workshop on Design and Architectures for Signal and Image Processing, DASIP 2007, Grenoble, France.
9. C. Berger, N. Widynsky. Using connected operators to manipulate image components. *Report, LRDE Seminar*, July 2005
10. A. Meijster. Efficient Sequential and Parallel Algorithms for Morphological Image Processing. PhD thesis, Rijksuniversiteit Groningen.
11. Berger, Ch.; Geraud, T.; Levillain, R.; Widynski, N.; Baillard, A.; Bertin, E. Image Processing, 2007. ICIP 2007. IEEE International Conference on Volume 4, Issue , Sept. 16 2007-Oct. 19 2007 Page(s):IV - 41 - IV – 44
12. B. Deloison. Recherche et développement en traitement d'image: Utilisation de l'arbre des composantes pour la fusion d'images. Report from graduate project, ESIEE Paris, June 2007.
13. Chiang, Y.-J., Lenz, T., Lu, X., and Rote, G., Simple and optimal output sensitive construction of contour trees using monotone paths. *Comp.Geometry: Theory and Applications*, 30(2):165–195, 2005.
14. Mattes, J., Demongeot, J., Efficient algorithms to implement the confinement tree. *In LNCS:*1953, pages 392-405. Springer, 2000.