An Efficient Parallel Algorithm for Graph-Based Image Segmentation

Jan Wassenberg[1], Wolfgang Middelmann[1], and Peter Sanders[2]

[1] FGAN-FOM, 76275 Ettlingen, Germany `jan.wassenberg@fom.fgan.de`
[2] Institute for Theoretical Informatics, Universität Karlsruhe, Germany

Abstract. Automatically partitioning images into regions ('segmentation') is challenging in terms of quality and performance. We propose a Minimum Spanning Tree-based algorithm with a novel graph-cutting heuristic, the usefulness of which is demonstrated by promising results obtained on standard images. In contrast to data-parallel schemes that divide images into independently processed tiles, the algorithm is designed to allow parallelisation without truncating objects at tile boundaries. A fast parallel implementation for shared-memory machines is shown to significantly outperform existing algorithms. It utilises a new microarchitecture-aware single-pass sort algorithm that is likely to be of independent interest.

1 Introduction and Related Work

Segmentation (automatically partitioning an image into regions) is an important early stage of some image processing pipelines, e.g. object-based change detection. The final results of such applications are often strongly dependent on the quality of the initial segmentation. Since subsequent processing steps can use higher-level region information instead of having to examine all pixels, the segmentation may also be the limiting factor in terms of performance. Many algorithms have been proposed, but good quality results often come at the price of high computational cost.

One extreme example of this is a multi-scale watershed approach (MSHLK) [1]. Repeated applications of anisotropic diffusion smooth the image and reduce the oversegmentation caused by the watershed transform. The resulting subjective quality is very good, but its computational cost (1 second per kPixel) is unacceptable.

An alternative approach uses the Mean-Shift (MS) [2] procedure to locate clusters within a higher-dimensional representation of the image. This is guaranteed to converge on the densest regions in this space and yields good results in practice, but the processing rate (0.1 MPixel/s) is still inadequate.

Recent work has shown that Maximally Stable Extremal Regions (MSER) [3] within a gradient image are also suitable for image segmentation. While more efficient (2 MPixel/s), this scheme only detects high-contrast segments and does not provide full coverage of the image. It also seems ill-suited for parallelisation since the stability criterion depends on a global ordering of pixels.

Graph-based segmentation (GBS) [4] increases the amount of data to be handled (multiple edges per pixel) but has several attractive properties. Viewing pixels as nodes of a graph allows the reduction of segmentation to finding cuts in a Minimum Spanning Tree (MST). Defining edge weights as some function of the pixels' per-band intensity differences enables the use of colour information without having to compute gradients. Finally, an MST can be assembled from partial sub-trees, which provides the possibility of parallelisation.

The remainder of this article is structured as follows: Section 2 develops a new online graph-cutting heuristic for MST-based segmentation. Section 3 shows the promising results obtained on well-known images. Section 4 introduces 'PHMSF' (Parallel Heuristic for Minimum Spanning Forests), which we believe to be the first non-trivially-parallel segmentation algorithm. Perhaps most importantly, Section 5 shows it to significantly outperform existing segmentation techniques.

2 Segmentation Algorithm

Segmentation algorithms require (often application-dependent) definitions of 'image region'. We consider 'homogeneity' and high contrast to surrounding pixels to be reasonable criteria [5]. Homogeneity can be computed as distances between (vector-valued) pixels; we find the L2 norm to yield better results than L1 or pseudo-norms. Note that [4] advocates separate segmentation of the R/G/B component images and intersecting the results. Since object edges are not always visible in all multi-spectral bands [6], it is safer (and certainly faster) to segment once using all bands. Recalling the graph segmentation framework, the above homogeneity measure defines the weight of edges. It remains to be seen how an online graph-cutting heuristic should partition the MST depending on edge weight. A mere threshold is insufficient because it fails to account for noise or the overall homogeneity of a region. [4] suggests an adaptive threshold that is incremented by a linearly decreasing function of the region size[3]. The function's slope is a user-defined parameter that must be determined by experimentation since it has no physical explanation. This scheme also underestimates a region's homogeneity by defining it as the maximum weight in its MST, thus tending towards oversegmentation. We suggest the adoption of an idea from Canny's edge detection algorithm [7]. In the context of edge detection, pixels with large gradient magnitudes are likely to correspond to edges, but there is no single level at which this ceases to be the case. Applying a rather high limit finds likely candidates, which can be augmented by nearby pixels that lie above a second, lower threshold. Returning to segmentation terminology, regions connected by low-weight edges represent likely candidates that can subsequently be expanded by following adjoining edges with somewhat higher weights. To avoid potentially unbounded growth, we institute a 'credit' limit on the sum of edge weights that may be added to a candidate region. Since no shape can be more compact than a circle, the region's perimeter is bounded from below by the circumference

[3] This unduly penalizes the growth of large segments; we saw slightly better results when dividing by the logarithm of the region size.

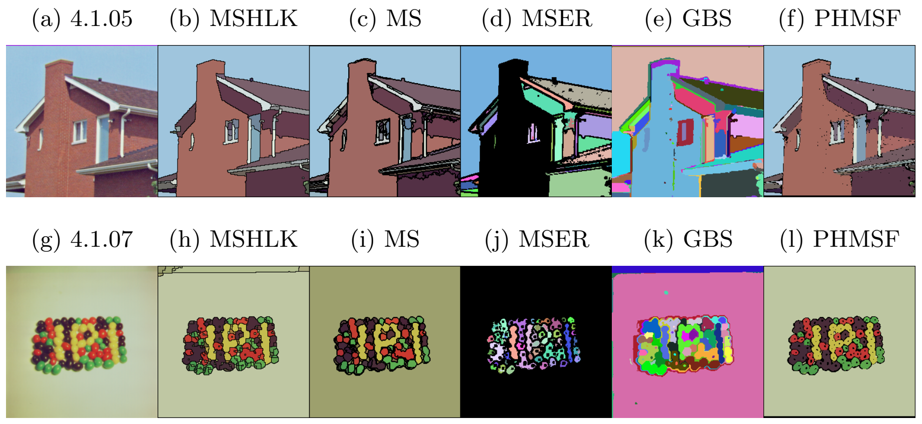
$\sqrt{4\pi \cdot \text{regionSize}}$. Let us also assume additive white Gaussian noise with variance σ_n^2, for which several estimators have been proposed [8, 9].s Defining contrast as the smallest edge weight along the border of any 'interesting' region minus $2 \cdot \sigma_n$ thus makes it likely that edges of total weight $\leq$ contrast $\cdot$ minPerimeter can be added to a region without inadvertently expanding beyond its bounds. This property is important because subsequent region merges are trivial, whereas splitting requires re-examination of the pixels or edges. However, the resulting regions are not necessarily too fine because pixels connected by low-weight edges are always merged. We have therefore averted global under- and oversegmentation of the image while using only local information. The algorithm first forms candidate regions by merging the endpoints of low-weight edges and then calls the following simple heuristic in increasing order of the remaining edges' weights:

```
Procedure EdgeHeuristic($edge$)
  $\text{region}_1, \text{region}_2 := \texttt{Find}(edge.\text{endpoints})$;
  if $\text{region}_1 \neq \text{region}_2$ then
      $\text{credit} := \min\{\text{region}_1.\text{credit}, \text{region}_2.\text{credit}\}$;
      if $\text{credit} > edge.\text{weight}$ then
          $\text{survivor} := \texttt{Union}(\text{region}_1, \text{region}_2)$;
          $\text{survivor}.\text{credit} := \text{credit} - edge.\text{weight}$;
```

3 Results

To demonstrate the usefulness of the new segmentation results, we compare them to the outputs of existing algorithms on standard images [10], the results of which are shown in Fig. 1:

Fig. 1: Segmentation results of the new PHMSF algorithm and others on USC SIPI [10] images 4.1.05 ('House') and 4.1.07 ('Jelly beans').



MSHLK [1] is known for high-quality results and provides excellent smoothing of the walls (b) but merges the eaves into the sky segment. We also call attention to the oversegmentation of the second image and shock effects [11] in the background (h). MS [2] is more successful at merging the individual objects (i) but also splits some of them (e.g. below the P); spurious segments near edges (c) are its only visible flaws. As with MSHLK, segment borders are delineated by black pixels. MSER [3] produces mostly adequate label images, though the wall is not considered to be a stable region (d); the effects of the gradient filter are clearly visible (j). GBS [4] is satisfactory but results in undersegmentation near the roof lines and oversegmentation of the sky and wall (e). It also merges different-coloured objects (k) but fails to return a uniform background. Our new PHMSF algorithm provides results comparable to MSHLK and MS and requires only 1/4000 and 1/50 the computation time, respectively (cf. Sect. 5). The black pixels (f) indicate surface irregularities that resulted in regions smaller than the minimum size. The segmentation in (l) is quite accurate, correctly separating different-coloured objects without introducing spurious boundaries.

4 Parallel Algorithm

Despite the efficiency of the new segmentation algorithm, a highly-tuned sequential implementation is still far slower than the collection rates of commercial imaging satellites (e.g. IKONOS with up to 90 km^2/s [12]). Since a significant reduction of the algorithm's constant factors appears unlikely and sequential programs have seen less benefit from recent CPU advances [13], it appears our self-set performance goal of 10 MPixel/s can only be reached by means of parallelisation. Note that embarrassingly-parallel schemes that simply split the input into independent tiles are not acceptable because they do not correctly handle objects straddling a border. Nor are overlapping tiles sufficient because there is no upper bound on the size of objects of interest (e.g. rivers or roads). Our first attempt at parallelisation addressed the MST computation. The new Filter-Kruskal scheme [14] combines ideas from Quicksort and Kruskal's algorithm and discards non-MST edges without having to sort them. This 'filter' operation, partitioning and sorting can all be parallelised. However, the total speedup on a quad-core system is only 1.5 – chiefly due to the sequential portion of the algorithm, but also because our eight-connected grid graphs are too sparse to derive much benefit from discarding edges. Our second approach is designed to allow independent processing of image tiles, but still ensures consistent results irrespective of the number of processors P.[4] The key observation is that Kruskal's MST algorithm can run in a data-parallel fashion until encountering an edge that crosses a tile border. From then on, MST components using such edges and in turn their incident edges must be 'delayed' until the partial MSTs of both tiles are available. We accomplish this with per-tile edge queues that are

[4] We ignore the (negligible) effects of unstable edge sorting.

processed in a subsequent sequential phase[5]. It remains to be seen how many edges are delayed – a long cross-border region of homogeneous pixels could affect a large proportion of a tile. However, high-weight edges at the boundary of such regions often serve as a 'firewall' because they can be discarded without affecting neighbouring regions. Only about 5 % of edges are delayed in practice, making Amdahl's argument less of a factor than real-world limits on memory bandwidth and P. The algorithm is described by the following pseudo-code:

Algorithm 2: Parallel Segmentation

```
parallel foreach tile do
    sort edges, merging those with weight < minWeight;
foreach borderEdge do  // connect and mark cross-border regions
    $\text{region}_1, \text{region}_2$ := Find(borderEdge.endpoints);
    survivor := Union($\text{region}_1, \text{region}_2$);
    Mark(survivor);
    tile.regions := tile.regions $\cup$ {survivor};
parallel foreach tile do
    foreach $r \in$ tile.regions do  r.credit := ComputeCredit(r.size);
parallel foreach tile do
    foreach edge in ascending order of weight do
        $\text{region}_1, \text{region}_2$ := Find(edge.endpoints);
        if edge crosses border then Mark($\text{region}_1$); Mark($\text{region}_2$);
        else if IsMarked($\text{region}_1, \text{region}_2$) then tile.delayQ.push (edge);
        else EdgeHeuristic(edge);
foreach tile do
    foreach edge $\in$ tile.delayQ do EdgeHeuristic(edge);
```

To avoid scheduling and locality issues, the (manually partitioned) loops reside in a single OpenMP parallel region. A novel variant of counting sort uses paged virtual memory to simulate bins of unlimited size and thus dispenses with a separate counting phase. An explicit buffering technique further increases performance by enabling write-combining without cache pollution. Details are given in App. A.

The algorithm outputs a Union-Find (UF) tree represented as an array of pointers to a parent pixel or region, as well as per-tile lists of regions, which each store size (number of pixels) and credit. Computing features for single-pixel regions would consume too much memory, so we only consider regions of size min..max. This requires relabeling the per-tile regions and replacing them with so-called 'accumulators' for the region features, which is accomplished by Alg. 3. Its separate and very efficient count phase seems preferable to updating the per-tile region count when cross-border merges are performed by the Kruskal algorithm. Since the desired output includes a label image, we 'collapse' the UF tree once all regions have been re-labeled. With all pieces in place, we can now compute the contribution of each pixel toward its region's features.

[5] This could be parallelised if edges indicate which border they cross, but our implementation cannot spare any space within the 32-bit representation.

Algorithm 3: Parallel Relabeling

```
parallel foreach tile do                                  // compress regions
    foreach $r \in$ tile.regions do $r$.isValid := $r$.size $\in$ [min, max];
parallel foreach tile do                                  // count regions
    tile.numRegions := 0;
    foreach pixel do
        if IsRepresentative(pixel) and Find(pixel).isValid then
            tile.numRegions := tile.numRegions + 1;
for $i := 0$ to $|\text{tiles}| - 1$ do
    tiles $[i]$ .startIndex := $\sum_{0 \leq j < i}$ tiles $[j]$ .numRegions;
parallel foreach tile do                                  // re-label regions
    foreach pixel do
        if IsRepresentative(pixel) and Find(pixel).isValid then
            parents [pixel] := tile.startIndex;
            tile.startIndex := tile.startIndex + 1;
```

The per-band intensities B_i and $\sum B_i^2$ are required for computing the band averages and standard deviations. For pixel coordinates (Y, X), the six moments $\sum Y^p \cdot X^q$ $(p, q \in \mathbb{N}_0, p + q \leq 2)$ are sufficient for estimating an ellipse [15]. Finally, counting the number of neighbour pixels belonging to different regions allows computing the region perimeter. Using 64-bit floating-point accumulators mitigates precision issues while still enabling vectorization via SSE2 instructions.

5 Performance Analysis

We first examine the complexity of the proposed algorithm. Counting sort is $O(N)$. Region merges via Union-Find are effectively $O(1)$ for all practical input sizes[6] [17]. All other operations are also constant-time and reside in loops with trip counts in $O(N)$, so the complexity is (quasi-)linear in the input size. Since this also applies to the MSER and GBS algorithms, we must compare their implementations. Table 1 lists the performance[7] of each algorithm for a representative 8.19 MPixel subset of a 16-bit, 4-component (RGB + NIR) Quickbird image of Karlsruhe.

Our PHMSF algorithm does more work (computing region features and processing the original four-component 16-bit pixels rather than an 8-bit RGB version), yet significantly outperforms the other algorithms. In this test it is 138 times as fast as MS [19], 28 times as fast as GBS [20] and 5 times as fast as

[6] We view the inverse Ackermann function as a constant ≤ 5 for $N < 10^{80}$. Note that an attempt at replacing Union-Find with a 'true linear algorithm' [16] introduces a constant factor of 8.

[7] Measured on a X5365 CPU (3.0 GHz, 32 GiB FB-DDR2 RAM) running Windows XP x64. Our implementation is compiled with ICC 11.0.066 `/Ox /Og /Ob2 /Oi /Ot /fp:fast /GR- /Qopenmp /Qftz /QxSSSE3`.

Table 1: Performance comparison.

Algorithm	MPixel/s
MSHLK	N/A
MS	0.09
GBS	0.45
MSER	2.53
PHMSF	12.80

Table 2: Performance on large images.

Sensor	Preproc.[8]	Bits	MPixel	MPixel/s
IKONOS	PS	16×4	54	13.5
QuickBird	PS	16×4	219	14.3
JAS150s	BF	8×4	527	24.4

[8] PS stands for pan-sharpening by an as-yet unpublished algorithm (56 MPixel/s), while BF denotes approximated per-channel Bilateral Filtering [18] with $\sigma_r = 15$ and $\sigma_s = 32$ (80 MPixel/s).

our similarly optimised implementation of MSER. Note that (32-bit) MSHLK exhausted its address space after a single diffusion iteration. Our PHMSF implementation requires much less memory: the working set is about 7.1 GB for a 1.97 GB image, which equates to 13.5 bytes/pixel. Table 2 shows measurements from processing large images of up to 527 MPixel. Performance improves with size due to increased parallelism – tile interiors grow faster than their borders. The parallel speedup varies between 2 and 3.2 when using 4 cores. In the latter case, sequential processing only accounts for 2 % of processing time; the limiting factor is memory bandwidth. RightMark Memory Analyzer [21] measures read and write throughputs of roughly 3500 MB/s and 2500 MB/s on this system. Having analysed the elapsed times and minimum amounts of data that must be transferred to/from memory during the credit computation, region compression/counting/relabeling and feature computation phases, we can conclude that each is at least 85 % efficient. Improving their performance or scalability is therefore contigent on increased bandwidth (e.g. via NUMA architecture or by adding further memory channels).

6 Conclusion

We have presented a new (quasi-)linear-time segmentation algorithm that provides useful results at previously unmatched speeds. Applications include automatic wide-area appraisal of the suitability of roofs for solar panels, object-based change detection, environmental monitoring and rapid updates of land-use maps. From an algorithm engineering standpoint, we believe this to be the first non-trivially-parallel segmentation algorithm. Its scalability is chiefly limited by the memory bandwidth of current SMP systems. Future work includes statistical estimation of the edge weight thresholds and efficiently computing a segment neighbourhood graph. We are also interested in applying this algorithm towards segment-based fusion of high-resolution electro-optical and hyperspectral imagery.

References

1. Vanhamel, I., et al.: Scale Space Segmentation of Color Images Using Watersheds and Fuzzy Region Merging. In: ICIP (1). (2001) 734–737
2. Comaniciu, D., Meer, P.: Mean Shift Analysis and Applications. In: ICCV. (1999) 1197–1203
3. Wassenberg, J., Bulatov, D., Middelmann, W., Sanders, P.: Determination of Maximally Stable Extremal Regions in Large Images. In: Signal Processing, Pattern Recognition, and Applications. (February 2008)
4. Felzenszwalb, P., Huttenlocher, D.: Efficient Graph-Based Image Segmentation. IJCV **59**(2) (September 2004) 167–181
5. Haralick, R., Shapiro, L.: Image Segmentation Techniques. CVGIP **29** (January 1985) 100–132
6. Thomas, C., Ranchin, T., Wald, L., Chanussot, J.: Synthesis of Multispectral Images to High Spatial Resolution: A Critical Review of Fusion Methods Based on Remote Sensing Physics. IEEE Trans. Geoscience and Remote Sensing **46**(5) (May 2008) 1301–1312
7. Canny, J.: A Computational Approach to Edge Detection. In: RCV87. (1987) 184–203
8. Shin, D.H., Park, R.H., Yang, S., Jung, J.H.: Block-Based Noise Estimation Using Adaptive Gaussian Filtering. IEEE Trans. Consum. Electron. **51** (2005) 218–226
9. Amer, A., Dubois, E.: Fast and Reliable Structure-Oriented Video Noise Estimation. IEEE Trans. Circuits Syst. Video Techn **15**(1) (2005) 113–118
10. Weber, A.: The USC-SIPI Image Database. `http://sipi.usc.edu/database/` Accessed 2008-10-06.
11. Buades, A., Coll, B., Morel, J.: The Staircasing Effect in Neighborhood Filters and its Solution. IEEE Trans. Image Processing **15**(6) (June 2006) 1499–1505
12. Dial, G., Bowen, H., Gerlach, F., Grodecki, J., Oleszczuk, R.: IKONOS satellite, imagery, and products. Remote Sensing of Environment **88**(1-2) (November 2003) 23–36
13. Sutter, H.: The Free Lunch is Over: A Fundamental Turn Toward Concurrency. Dr. Dobb's Journal (March 2005)
14. Osipov, V., Sanders, P., Singler, J.: The Filter-Kruskal Minimum Spanning Tree Algorithm. In Finocchi, I., Hershberger, J., eds.: ALENEX, SIAM (2009) 52–61
15. Zunic, J., Sladoje, N.: Efficiency of Characterizing Ellipses and Ellipsoids by Discrete Moments. IEEE Trans. Pattern Anal. Mach. Intell **22**(4) (2000) 407–414
16. Nister, D., Stewenius, H.: Linear Time Maximally Stable Extremal Regions. In: ECCV. (2008) II: 183–196
17. Harfst, G., Reingold, E.: A Potential-Based Amortized Analysis of the Union-Find Data Structure. SIGACT **31** (September 2000) 86–95
18. Durand, F., Paris, S.: A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach. In: ECCV. (2006) IV: 568–580
19. Robust Image Understanding Lab: EDISON System. `http://www.caip.rutgers.edu/riul/research/code/EDISON/doc/segm.html` Accessed 2008-09-23.
20. Felzenszwalb, P.: Efficient Graph-Based Image Segmentation. `http://people.cs.uchicago.edu/~pff/segment/` (March 2007) Accessed 2008-01-11.
21. Besedin, D.: RightMark Memory Analyzer. `http://cpu.rightmark.org` Accessed 2009-01-09.
22. Bender, M., Farach-Colton, M., Mosteiro, M.: Insertion Sort is O(n log n). Mathematical Systems Theory **39** (2006)

23. Navarro, J., Iyer, S., Druschel, P., Cox, A.: Practical, Transparent Operating System Support for Superpages. In: OSDI-02. Operating Systems Review, New York, ACM Press (December 2002) 89–104
24. Mehlhorn, K., Sanders, P.: Scanning Multiple Sequences via Cache Memory. Algorithmica **35** (2003)
25. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual. (November 2007)

A Microarchitecture-aware Sort

The Kruskal algorithm requires visiting edges in increasing order of their weights. A straightforward implementation sorts (i.e. permutes into sorted order) all edges, which is possible in $O(N)$ time because only edge weights $\leq M$ need be considered, thus enabling counting sort. This typically involves two passes through the data, first building a histogram of weights and then writing edges to the correct output position. However, a weaker post-condition suffices for iteration – 'bins' (the set of edges with a given weight) need not be immediately adjacent[9]. This suggests a simplified algorithm where bins of capacity N are pre-allocated and the initial scan is skipped. Edges are simply written to the next free position in the corresponding bin:

Algorithm 4: Two-pass algorithm for iterating over edges in sorted order

```
storage := ReserveAddressSpace($N \cdot M$);
for $i := 0$ to $M$ do next[$i$] := $i \cdot N$;
foreach edge do
    storage[next[edge.weight]] := edge;
    next[edge.weight] := next[edge.weight] + 1;
for $w := 0$ to $M$ do
    for $i := w \cdot N$ to next[$w$] do EdgeHeuristic(storage[$i$])
```

This algorithm only writes and reads each edge once, a feat that comes at the price of $N \cdot M$ space. While this appears problematic under the Random-Access-Machine model, it is easily handled by 64-bit CPUs with paged virtual memory. Physical memory is mapped to pages of size S when they are first accessed,[10] thus reducing the actual memory requirements to $N + M \cdot S$. The remainder of the initial allocation only occupies address space, a plentiful resource on 64-bit systems. Note that using large pages ($S = 2$ MiB on x86-64 architectures)

[9] Library Sort [22] is based on a similar idea, viz. using gaps between elements to speed up insertion sort.

[10] Accesses to non-present pages result in a page fault exception. The application receives such events via signals (POSIX) or Vectored Exception Handling (Microsoft Windows) and reacts by committing memory, after which the faulting instruction is repeated.

increases TLB coverage and also reduces the number of page faults at the cost of internal fragmentation [23].

A further microarchitectural consideration concerns the manner in which edges are written to a bin. An ideal cache with M lines could exploit the writes' spatial locality and entirely avoid noncompulsory misses. However, perfect hit rates are not achievable in practice due to limited set associativity [24]. The common (pseudo-)LRU eviction policy also causes the displacement of previously cached data instead of write-only lines that are not accessed after having been filled. This 'cache pollution' can be avoided by writing directly to memory via SSE2 non-temporal streaming stores[11]. Since 'partial writes' (non-burst transfers) involve significant bus overhead, the architecture attempts to combine neighbouring writes. However, M is likely to exceed the six write-combine (WC) buffers provided by current microarchitectures, so WC degenerates to partial writes because WC buffers are often flushed to memory before being filled. Having established the drawbacks of cached and non-temporal writes, we now show how to avoid them by means of software write-combining [25]. Instead of writing to each bin directly, edges are first placed into cache-line-sized temporary buffers. When one of these is full, it is copied to the actual destination via non-temporal writes, which can be combined into a single burst transfer. This scheme avoids the excessive cache pollution caused by normal writes and works around the limited number of WC buffers by using L1 cache lines for that purpose. The effect is a further 7 % speedup of the entire edge creation phase and an increase in scalability due to less memory traffic.

While this section has covered highly system-specific details, we believe the basic principles (taking advantage of paged virtual memory and reducing partial bus transfers) to have widespread application. Since these issues can determine the feasibility of algorithms, they must be considered during the design phase and cannot be relegated to a separate optimization phase.

[11] These also reduce pressure on the memory interface by avoiding the relatively expensive load of the destination cache line via Read-For-Ownership transaction.