MVA '96 IAPR Workshop on Machine Vision Applications, November. 12-14, 1996, Tokyo, Japan

Parallel Image Processing on Heterogeneous SIMD-MIMD Machines

Pieter Jonker, Eddy Olk, Koen Volker

Pattern Recognition Section, Faculty of Applied Physics,
Delft University of Technology,
P.O. Box 5046, 2600 GA Delft, The Netherlands
pieter@ph.tn.tudelft.nl

Abstract

From a hardware standpoint a heterogeneous architecture such as an SIMD array coupled onto a MIMD system, will yield a powerful solution for real time embedded vision tasks. From a software standpoint there is a need to approach this heterogeneous system in a homogeneous way. We investigated the feasibility of a uniform parallel programming approach on a heterogeneous machine. Using the parallel language CC++ we can express functional parallelism with constructs like **par** and **parfor** and hide the data parallelism, like images distributed over SIMD PEs, in CC++ classes. We investigated a way of parallel programming using arrays of buckets. These bucket-arrays can be distributed over MIMD Processing Units, SIMD Processing Elements or both. A bucket array is hidden in a CC++ class and can be approached by independent producer and consumer threads of the parallel program. We analysed the feasibility and efficiency of this approach with the constrained Euclidean Distance Transform while varying the number of processors from 2 to 128.

1 Introduction

In real-time embedded vision task such as in robot vision or autonomous car driving, computing power is necessary on all levels of operation. These tasks start with a plain image and while processing, the type of operations moves from arithmetic to symbolic and the amount of data to process reduces, until eventually some decision based on the analysed data can be made. Parallel computing power is useful on all levels of operation: Low level or image processing, Intermediate level or object understanding, High level or scene understanding and finally Mission control.

From a hardware standpoint a heterogeneous architecture, e.g. an SIMD array coupled onto a MIMD system, will yield a powerful solution. Generally an SIMD architecture [1], [2], [3] is suitable to exploit the fine grain parallelism of the low-level processing operation. Such an architecture often has also proper interfaces with the sensor and the host system. E.g. the Smart Camera [2] couples a 2D sensor array on chip with a single bit Linear Processor Array, and the IMAP [1] series of systems is based on a dual ported memory coupled on chip with an 8 bit Linear Processor Array and a shift register section for shifting in video lines during operation.

Shared memory multi-processor systems supporting multi-threading, or distributed memory multi-processor systems (MIMD), coupled onto the SIMD systems [12] are quite able to exploit their coarse grain parallelism for high level operations. Although for numerous applications either SIMD or MIMD systems alone will be sufficient for the task, in many hierarchical applications with a video speed data-processing character, a heterogeneous SIMD-MIMD architecture may provide a more powerful solution.

From a software standpoint there becomes a need to approach this heterogeneous system in a homogeneous way, making the parallel programming of such a system a agreeable activity. We investigated the feasibility of a uniform parallel programming approach on such a heterogeneous machine [4]. This approach should be generic, usable on many heterogeneous architectures.

As our interest is in architecture design and smooth programming of applications, we adopted a parallel language from which we believed it could suit our needs. In the parallel language Compositional C++ [5] we can express functional parallelism with constructs like **par** and **parfor** blocks and at the same time hide the data parallelism like images distributed over SIMD PEs in CC++ classes.

2 Benchmark algorithm implementations

As example we use the constrained Euclidean Distance Transform. Assume that we have a linear array [6] MIMD or SIMD, in which the image is column-wise mapped over the P processors.

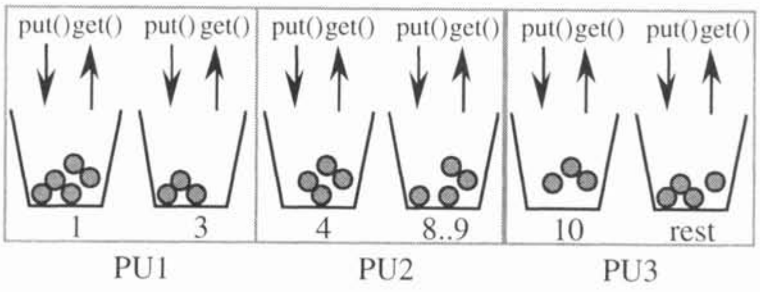

Figure 1. A bucket-array distributed over processing units

Various implementations of the cEDT exist:

1. The pure SIMD method, using the bit-wise sum of successive dilations of the objects in the image. This uses a local 3x3 neighbourhood around each pixel.
2. The Borgefors method which is an algorithm based on a two pass scan over the image which uses a recursive 3 x 3 neighbourhood around each pixel [7].
3. A method based on using only the pixels that are likely to change [8].

Although the latter method of processing pixels in queues is known for a while [9], we adopted this sequential approach to a parallel approach using distributed arrays of buckets [10]. These bucket arrays can be distributed over MIMD Processing Units, SIMD Processing Elements or both. A bucket array is hidden in a CC++ class and can be approached by independent producer and consumer threads of the parallel program.

A *bucket* is defined as a data structure (C++: class) with two access functions (C++: public functions): a `put()` function to put data elements with certain characteristics into a bucket and a `get()` function to retrieve an arbitrary data element from this bucket. This definition allows different implementations of the bucket data structure, like FIFO, LIFO or Linked List. Important is, however, that the user may not use this implementation knowledge, as the bucket data structure may in our case be distributed over a number of processors, which may destroy the implementation (e.g. the FIFO) character.

A *bucket-array* is defined as an array of buckets where each bucket has a label, a numeric value or range of values attached to it. The bucket-array as a whole is considered as a single data structure with two access-functions, similar to the single bucket: a `put()` function to put an item in a specific bucket and a `get()` function to retrieve an arbitrary item from a specific bucket from the array. The buckets in the array should have unique non overlapping labels. Consequently in the bucket-array we can store data with similar properties in a single bucket. When the array is distributed over processors we assume that a bucket of the array entirely resides on a single processor. See figure 1 where a bucket-array with 6 buckets, labelled {1, 3, 4, 8..9, rest} is distributed over three MIMD Processing Units.

Usually such a bucket-array is emptied and processed, whereupon the processed data items are put into the array again, possibly in other buckets. This means that processors only read in their own buckets, but may send data items over a network to buckets in another processor. More than one array can be involved in the processing and arrays may even reside on other parts of the heterogeneous architecture. By way of example, figure 2 shows an image crinkle-wise stored onto Processing Elements of an SIMD array.

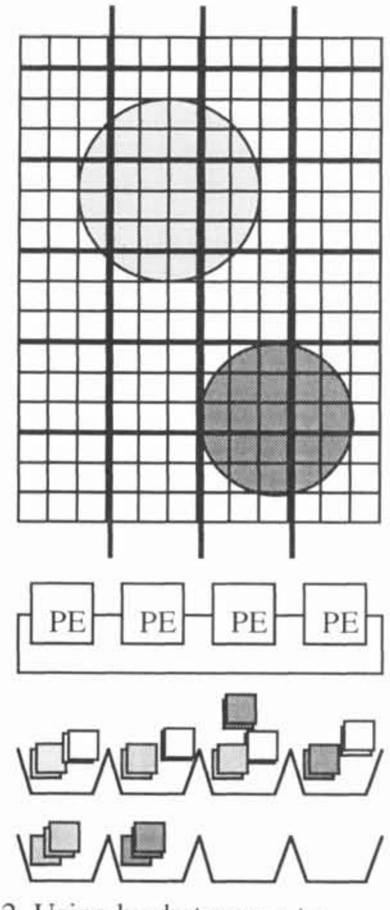

Figure 2. Using bucket arrays to redistribute data items over processors.

If the SIMD array allows indirect addressing, buckets can be implemented on them too, and even if the array has no long distance network to route data items to buckets on PEs further away, research showed that a Nearest Neighbour network can be programmed to do so [11].

When each PE has stored the object pixels that reside in his part of the distributed image in a first bucket array, than after e.g. a labelling procedure all pixels of one object can be stored into one bucket of a second bucket array. Where the first array resides on the SIMD system, the second array may reside on an MIMD system. This can be profitable for some applications, where the PUs of the MIMD system can proceed with the object and scene interpretation and the SIMD system can grab a new image and pre-process it. Concluding: Programming with distributed images and bucket arrays in combination with functional parallel constructs allow us to program hierarchical applications on heterogeneous architectures.

3 Analysis of the Euclidean Distance Transform

To show the feasibility and efficiency of the homogeneous parallel programming approach using bucket-arrays, this section describes in pseudo CC++ the analysis of the constrained Euclidean Distance Transform (cEDT) while varying the number of processors from 2 to 128. For this experiment we used CC++ with the Nexus runtime system [13] mapped onto the multi-threading facility of Solaris 2.5. The cEDT starts from the object edges and propagates contour by contour inward in a wave front way, writing the distance to the object border in an output image. We will discuss three methods of the cEDT: Synchronised, Unsynchronized and Enhanced Unsynchronized. In the first method, all processors perform one propagation step inward in one iteration. If the processing of the front in one processor invokes candidates residing in the neighbouring processor, these candidates are put into a bucket of the neighbouring processor. When all processors are ready with the processing of the current front, the wave is allowed to propagate one step inward.

The pseudo CC++ code of a synchronised cEDT program [8], [10] is:

```
// allocate distributed images
dist_bit_image source(256,256);
dist_grey_image result(256,256);

// allocate two distributed bucket-arrays per wind direction
// (so each processor has 2 times 8 buckets in his local memory)
dist_grey_bucket N[2],NE[2],E[2],SE[2],S[2],SW[2],W[2],NW[2];

// load source; column-wise map the data-types over processors
// initially fill the buckets by scanning the source
parfor(x=0; x<=255; x++) {
    for(y=0; y<=255; y++) {
        if (source.val(x,y) == OBJECT) {result.val(x,y) = 0}
        else {
        result.val(x,y) = MAXVAL;
        if (source.val(x+1,y) == OBJECT) { W[0].put(x,y,1) };
        if (source.val(x+1,y+1) == OBJECT) { SW[0].put(x,y,2) };
        //etc for all other wind directions of the neighbourhood
        }
    }
}

// perform the wave front propagation in a data parallel way
ready = FALSE; i = 0; HorVerCoeff = 3; DiagCoeff = 6; Diff = 10;
parwhile (!ready) {
    ready = TRUE;
    parwhile(!W[i].empty) {
        pixel a = W[i].get;
        if (result.val(a.x, a.y) > a.val {
            result.val(a.x, a.y) = a.val;
            W[i].put(a.x-1, a.y, a.val + HorVerCoeff);
            ready = FALSE
        }
    }
    parwhile(!SW[i].empty) {
        pixel a = SW[i].get;
        if (result.val(a.x, a.y) > a.val {
            result.val(a.x, a.y) = a.val;
            SW[i].put(a.x-1, a.y+1, a.val + DiagCoeff);
            S[i].put(a.x, a.y+1, a.val + HorVerCoeff);
            W[i].put(a.x-1, a.y, a.val + HorVerCoeff);
            ready = FALSE
        }
    }
    // etc. for all other wind directions

    // update coefficients, swap read and write bucket-arrays
    HorVerCoeff += 2; DiagCoeff += Diff; Diff += 4; i = mod(2,i+1);
}
```

The pseudo source code of the unsynchronised version is almost identical to the synchronised version. The synchronisation is realised by using separate read and write bucket-arrays. If, however, results are put back in the same (read_write) bucket-array, an unsynchronised version is obtained. Then all processors process the full cEDT over the pieces of object that reside in their own memory. The processors can proceed until they are blocked because no more work is generated in their own piece of the image. Again, if the processing of the front in one processor invokes candidates residing in the neighbouring processor, these candidates are also put into the read_write buckets of the neighbouring processor. As candidates generated by neighbours are delayed with respect to candidates generated by the own processor there will be a preference for own candidate points in each processor. To enlarge this effect, it can even be forced by putting candidates for neighbouring processors in a separate neighbour bucket array that is only emptied into the read-write bucket array when all processors are ready. These new candidates are processed until the read-write buckets are again empty, the neighbour bucket array is emptied into the read-write bucket array, etc...

It will be clear that the synchronised method is more an SIMD approach and the unsynchronised method more an MIMD approach. The saving up of neighbour candidates in a separate bucket-array is also more in accordance with the DMA block data transfer between MIMD processors than the transfer of one candidate at a time, as can more easily be realised in SIMD systems. Concluding: In SIMD, with many simple processors, synchronisation and neighbour communication is easy. In MIMD, with less more powerful processors, it is more beneficial to let the processors go ahead as far as they can come, as synchronisation and communication is slower.

Figure 3 on the next page shows the average memory access overhead plotted against the number of processors { 2,4, 8 .. 128} for an enhanced unsynchronised version of the cEDT on a 256^2 image. The average memory access overhead is obtained by taking the total memory access overhead and divide it by the number of processors. The total memory access overhead is the number of excess reads/writes on all processors in comparison with a single processor version of the same algorithm. The figure shows that for this image the enhanced unsynchronised version of the algorithm is equally efficient for a modest number of processors (16) and for a large number of processors (128). This can be explained by the fact that for a modest number of processors most objects remain within the local memory of the processor and can hence be processed locally. When the number of processors increase, more objects cross the processor boundary and give rise to propagation seeds that invoke new free running wave fronts that partially overwrite the values calculated by older waves. In the high limit case with 256 (SIMD) PEs this algorithm is almost equal to the synchronised version of the algorithm, as all objects always cross the PE boundary.

4 Conclusions

Programming with CC++ and the introduction of distributed abstract data-types such as images and bucket-arrays is a method that theoretically allows to smoothly program heterogeneous SIMD-MIMD architectures in a single parallel program. However, research is still necessary on the port of runtime systems onto such a heterogeneous architecture.

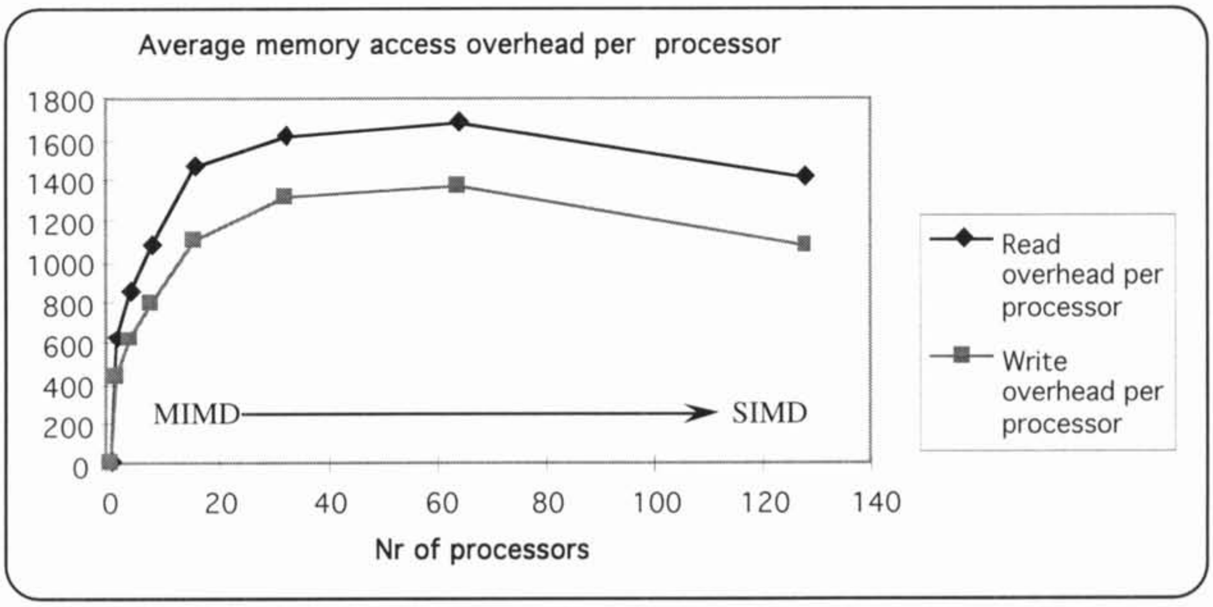

Figure 3 Average memory overhead per processor from 1 to 128 processors

Surprisingly enough, a single version of the cEDT, the enhanced unsynchronised version, can efficiently run on both an MIMD system with a modest number of PUs and on an SIMD system with the number of PEs in the order of the columns of the image.

5 Acknowledgement

This work was supported by the Commission of the European Union (LTR 8849)

6 Literature

[1] Fujita Y, Yamashita N, Okazaki S "A 64 Parallel Integrated Memory Array Processor and a 30 GIPS Real-Time Vision System" CAMP '95. Proceedings of the Workshop on Computer Architectures for Machine Perception (Como, I, Sept 18-20), IEEE Computer Society Press.

[2] Forschheimer R, Ingelhag P, Jansson C, "MAPP2200, a second generation smart sensor", SPIEW Vol 1659, 1992.

[3] D. W. Hammerstrom and D. P. Lulich, Image Processing using One-Dimensional Processor Arrays, Proceedings of the IEEE Vol. 84, N. 7, July 1996 pp.1005-1018.

[4] Olk JGE, Jonker PP, "A Programming and Simulation Model of a SIMD-MIMD Architecture for Image Processing" CAMP '95. Proceedings of the Workshop on Computer Architectures for Machine Perception (Como, I, Sept 18-20), IEEE Computer Society Press.

[5] Sivilotti PA.G, Carlin PA, "A Tutorial for CC++n" Compositional Systems Research Group. Dept. of Comp. Science. Caltech Mail stop 256-80, Pasedena, CA 91125, USA. www.compbio.caltech.edu/CCplusplus.html

[6] Jonker PP, "Why linear arrays are better image processors", Proceedings of the 12th IAPR International Conference on Pattern Recognition, Conference D (Jerusalem, Israel, October 9-13, 1994), IEEE Computer Society Press, Los Alamitos, CA, 1994, 334-338.

[7] Borgefors G "Distance Transformations in Digital Images" Computer Vision, Graphics and Image Processing 34: 344-371, 1986.

[8] Bouts E, "A fast, error free, squared Euclidean distance transform", Proceedings of VIP '93, International Conference on Volume Image Processing (Utrecht, The Netherlands, June 2-4, 1993), Stichting Computer Vision Research SCVR, Utrecht, 1993, 47-50.

[9] Groen FCA, Foster NJ "A fast algorithm for cellular logic operations on sequential machines" Pattern Recognition Letters, vol. 2, no. 5, 1984, 333-338.

[10] Olk JGE, Jonker PP, "Bucket Processing: a new paradigm for Image Processing" Accepted for the journal of Pattern Recognition and Image Analysis. ISSN 1054-6618

[11] Baglietto P. Maresca M, Migliardi M, Zingirian N, de Lescure B, Guérin B, Colaïtis MJ. "Eprit BRA Project 8849 SMIMP Deliverable Bn23: The Implementation of the Image Processing Layer Part I" January 1996. [pieter@ph.tn.tudelft.nl]

[12] "Parsytec CC series hardware documentation", "Embedded Parix software documentation" Parsytec GmbH, Auf der Huels 183, D-52068 Aachen, Germany. www.parsytec.de

[13] Foster I, Kesselman C, Tuecke S. "The Nexus parallel runtime system" Argonne National Laboratory, Argonne, Ill. 60439,1994. www.mcs.anl.gov/nexus