MVA2005 IAPR Conference on Machine VIsion Applications, May 16-18, 2005 Tsukuba Science City, Japan
9-5

Skeletons and Asynchronous RPC for Embedded Data- and Task Parallel Image Processing

Wouter Caarls, Pieter Jonker
Delft University of Technology
Quantitative Imaging Group
Lorentzweg 1, Delft, The Netherlands
{w.caarls, p.p.jonker}@tnw.tudelft.nl

Henk Corporaal
Eindhoven University of Technology
Department of Electrical Engineering
Den Dolech 2, Eindhoven, The Netherlands
h.corporaal@tue.nl

Abstract

Developing embedded parallel image processing applications is usually a very hardware-dependent process, requiring deep knowledge of the processors used. Furthermore, if the chosen hardware does not meet the requirements, the application must be rewritten for a new platform. We wish to avoid these problems by encapsulating the parallelism.

We have proposed the use of algorithmic skeletons *[3] to express the data parallelism inherent to low-level image processing. However, since different operations run best on different kinds of processors, we need to exploit task parallelism as well.*

*This paper describes our asynchronous remote procedure call (*RPC*) system, optimized for low-memory and sparsely connected systems such as smart cameras. It uses a* futures*[10]-like model to present a normal imperative C-interface to the user in which the skeleton calls are implicitly parallelized and pipelined.*

1 Introduction

As processors are becoming faster, smaller, cheaper, and more efficient, new opportunities arise to integrate them into a wide range of devices. However, since there are so many different applications, there is no single processor that meets all the requirements of every one. The SMARTCAM [6] project investigates how an application-specific processor can be generated for the specific field of intelligent cameras, using design space exploration.

The processing done on an intelligent camera has very specific characteristics. On the one hand, low-level image processing operations such as interpolation, segmentation and edge enhancement are local, regular, and require vast amounts of bandwidth. On the other hand, high-level operations like classification, path planning, and control may be irregular while typically consuming less bandwidth. The architecture template on which the design space exploration is based therefore contains data-parallel (SIMD) as well as instruction-parallel (ILP) processors.

One of the main goals of the project is keeping the system easy to program. This means that one single program should map to a wide range of configurations of a wide range of processors. It also means that the application developer shouldn't have to learn a parallel programming language. The solution presented below is based on using algorithmic skeletons to exploit data parallelism within each operation, while a form of asynchronous RPC allows the operations to run concurrently.

The structure of this paper is as follows: section 2 reviews some related work. Sections 3 and 4 describe our programming environment and some optimizations. Section 5 presents some results from our prototype, and finally section 6 draws conclusions and points to future work.

2 Related work

Programming environments for image and signal processing applications are widely ranged. Tightly coupled systems usually have parallel extensions to a sequential language, like Celoxica's Handel-C [2] for FPGA programming, or NEC's 1DC [7] for their IMAP SIMD arrays. More loosely coupled systems usually work with the concept of a *task* or *kernel*, and differ in how these tasks are programmed and composed.

Process networks such as used by YAPI [4] allow much freedom in specifying the tasks, but require a static connection network between them. StreamC/KernelC [8], developed for Imagine, reduces the allowed syntax within a kernel, but makes the interconnections dynamic by using *streams*. Their current implementation doesn't support task parallelism, however. EASY-PIPE [9] does, but requires a batch of tasks to be explicitly compiled and dispatched by the user. Their main contribution is the use of algorithmic skeletons to make programming the tasks easier. Finally, Seinstra [11] allows no user specification of the tasks, instead relying on an existing image processing library. It is also limited to data parallelism, but these restrictions allow it to be more transparent to the user, presenting a purely sequential model.

Futures were introduced in the MultiLisp [5] language for shared-memory multiprocessors. Requesting a future spawns a thread to calculate the value, while immediately returning to the caller, which only blocks when it tries to access it. Once the calculation is complete, the future is overwritten by the calculated value. Batched futures [10] apply this concept to RPC, but with the intent to reduce the amount of RPC calls by sending them in batches that may reference each other's results.

3 Programming

Our programming environment is based on C, to provide an easy migration path. In principle, it is possible (although slow) to write a plain C program and run it on our system. In order to exploit concurrency, though, it is necessary to divide the program into a number of image processing operations, and to apply these using function calls. Parts of the program which cannot easily be converted can be left alone unless the speedup is absolutely necessary.

The main program, which calls the operations and includes the unconverted code, is run on a *control processor*, while the image processing operations themselves are run on the *coprocessors* that are available in the system (the control processor itself may also act as a coprocessor). Only this main program can make use of global variables; because of the distributed nature of the coprocessor memory, all data to and from the image processing operations must be passed using parameters.

3.1 Within-operation parallelism

The main source of parallelism in image processing is the locality of pixel-based operations. These low-level operations reference only a small neighborhood, and as such can be computed mostly in parallel. Another example is object-based parallelism, where a certain number of possible objects or regions-of-interest must be processed. Both cases refer to *data parallelism*, where the same operation is executed on different data (all pixels in one case, object pixels or objects in the other).

Data parallel image processing operations map particularly well on linear SIMD arrays (LPAs). However, since we don't want the application developer to write a parallel program, we need another way to allow him to specify the amount of parallelism present in his operations. For this purpose, we use *algorithmic skeletons*. These are *templates* of a certain computational flow that do not specify the actual operation, and can be thought of as higher-order functions, repeatedly calling an *instantiation function* for every computation. Take a very simple binarization:

```
for (y=0; y<HEIGHT; y++)
  for (x=0; x<WIDTH; x++)
    out[y][x] = (in[y][x]>128);
```

Using a higher-order function, **PixelToPixelOp**, we can separate the structure from the computation. **PixelToPixelOp** will implement the loops, calling **binarize** every iteration:

```
int binarize(int value)
  return (value>128);

void PixelToPixelOp(int (*op)(int),
    int in[HEIGHT][WIDTH], int out[HEIGHT][WIDTH])
  for (y=0; y<HEIGHT; y++)
    for (x=0; x<WIDTH; x++)
      out[y][x] = op(in[y][x]);

PixelToPixelOp(binarize, in, out);
```

Note that implementing **PixelToPixelOp** column-wise instead of row-wise – by interchanging the loops – does not change the result, because there is no way for **op** to reference earlier results (side effects are not allowed). It can be said that by specifying the inputs and outputs of the instantiation function, the skeleton characterizes the available parallelism. So, by choosing a skeleton, the programmer makes a statement about the parallelism in his operation, while not specifying how this should be exploited. This freedom will allow us to optimally map the operation to different architectures.

Another benefit is that the image processing library normally shipped with DSPs and other image processors is replaced by a skeleton library, which is more general and thus less in need of constant updates.

Of course, not all operations can be data-parallelized as easily as pixel operations. More irregular operations place increasing demands on the autonomy and interconnection of the processing elements. For example, for efficient implementation, local neighborhood operations are straightforward, recursive neighborhood operations require indirect addressing, run-length encoding requires non-local communication, and edge following is mostly sequential. However, even in the sequential case the skeleton approach can still be used, if only to facilitate programming instead of parallelization. If specialized hardware then becomes available, it is easier to make use of it.

3.2 Between-operation parallelism

An image processing application consists of a number of operations described above, surrounded by control flow constructs. Because our hardware platform is heterogeneous, it is important that multiple of these operations are run concurrently, as not all processors can be working on the same computation. We are therefore using asynchronous RPC calls as a method to exploit this task-level parallelism.

In RPC, the *client* program calls *stubs* which signal a *server* to perform the actual computation. In our case, the application is the client program running on the control processor, while the skeleton instantiations are run on the coprocessors. This alone does not imply parallelism, because the stub waits for the results of the server before returning. In asynchronous RPC, therefore, the stub returns immediately, and the client has to *block* on a certain operation before accessing the result. This allows the client program to run concurrent to the server program, as well as multiple server programs to run in parallel.

However, this still has the disadvantage of requiring the client program to wait on the completion of an operation before passing its result to another one, even though it never uses the results itself. To address this problem, we are using MultiLisp's concept of *futures*, placeholder objects which are only blocked upon when the value is needed for a computation. Since simple assignment is not a computation, passing the value to a stub doesn't require blocking; once the called function needs the information, it will block itself until the data is available, without blocking the client program:

```
while(1)
  Read(in);
  PixelToPixelOp(op1, in, inter1);
  PixelToPixelOp(op2, in, inter2);
  PixelReductionOp(op3, inter1, inter2, out);
  /* ... Concurrent client code ... */
  _block(out);
  /* ... Dependent client code ...  */
```

4 Optimizations

While our futures-like implementation is much less elaborate than MultiLisp's (requiring, for example, explicit blocks on results, although these could be inserted by the compiler), it does tackle two other problems: data distribution and memory usage. Both originate from our architecture template, which features distributed-memory processors with a relatively low amount of on-chip memory.

Furthermore, although the skeletons are called as higher-order functions in order to provide an easy migration path, we avoid the function call overhead by using *source-to-source transformations*. Using transformations also allows us to translate between different target processor languages, and to provide an efficient way to specify skeletons that are polymorphic in the number and types of their arguments.

4.1 Data distribution

The data generated by most image processing operations is not accessed by the client program, but only by other operations. This data should therefore not be transported to the control processor. In order to achieve this, we make a distinction between images (which are streams of values) and other variables.

Images are never sent to the control processor unless the user explicitly asks for them, and as such no memory is allocated and no bandwidth is wasted. Rather, they are transported between coprocessors directly, thus avoiding the scatter-gather bottleneck present in some earlier work [9]. All other variables (thresholds, reduction results, etc.) are gathered to the control processor and distributed as necessary. These can be used by the programmer without an explicit request.

The knowledge about which data to send where, simply comes from the inputs and outputs to the skeleton operations, which are derived from the skeleton specification and are available at run time. Coprocessors are instructed to send the output of an operation to all peers that use it as an input.

4.2 Memory usage

Our concern about memory usage stems from the fact that especially SIMD LPAs for low-level image processing may not have enough memory to hold an entire frame, let alone multiple frames if independent tasks are mapped to it. These processors are usually programmed in a pipelined way, where each line of an image is successively led through a number of operations. We would like our system to conserve memory in the same way, and have therefore specified all our skeletons to read from and write to *FIFO buffers*.

The distribution mechanism allocates these buffers, and sets up transports as described above. The operations themselves read the needed information from the buffer, process it, and write the results to another buffer. A method is provided to signal that no more data will be forthcoming. This conserves memory, because even a series of buffers is generally much smaller than a frame. Simultaneously, it hides the origin of the data, making the operations independent of the producers of their input and the consumers of their output.

The price of all this is that operations must consume data in a certain order, and if the source operation doesn't generate it in the correct sequence, a *reordering operation* must be inserted, typically requiring a frame memory. Fortunately, many low-level operations can tolerate different orderings, while more irregular operations are generally run on processors with enough memory.

5 Results

We have implemented a double thresholding edge detection algorithm on a prototype architecture consisting of a XETAL [1] 16 MHz 320-PE SIMD processor and a TriMedia [12] 180 MHz 5-issue VLIW processor (figure 1). In this algorithm, the Bayer pattern sensor output is first interpolated, then the Sobel X and Sobel Y edge detection filters are run and combined, the output is binarized at two levels, and finally the high threshold is propagated using the low threshold as a mask image. This final propagation cannot be run on the XETAL, because it requires a frame memory.

Three situations were compared: one in which the entire algorithm was implemented in a single operation on the TriMedia, as a baseline for how a sequential application would be written. Next, the operation was split into tasks as described above, and all tasks were mapped to the TriMedia; this shows the overhead caused by the task switching and buffer interaction. Finally, all low-level operations were mapped to the XETAL, while the propagation and display were mapped to TriMedia; this resembles the final situation as it would run on our system. See table 1.

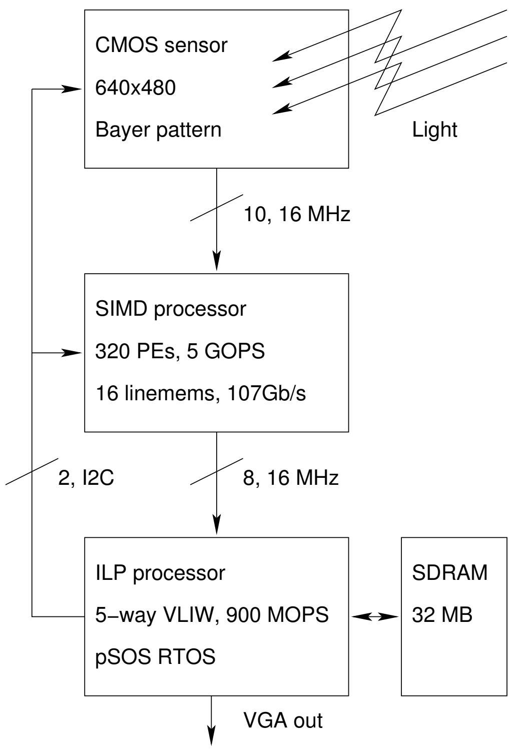

Figure 1: Inca+ prototype architecture (Philips CFT)

Table 1: Timing results of the double thresholding edge detection algorithm

Trial	**Processing time**
Single operation (TriMedia)	115 ms
Split operations (TriMedia)	124 ms
Parallel (XETAL + TriMedia)	67 ms

Because XETAL has only 16 line memories, the buffers between the filters were 1 line. On the TriMedia, they were 16 lines, to avoid too much context switching. An allocate-and-release scheme was used on the TriMedia, so that no extra state memory was needed in the filters, and no unnecessary copies were made.

As can be seen, the overhead of running the RPC system is around 8% (with 16-line buffers; the overhead approaches zero if full-frame buffers are used, but that is unrealistic). This seems quite a reasonable tradeoff if we consider that it can now run transparently on the parallel platform, achieving a 42% processing time decrease. Actually, because the filtering and propagation are done concurrently in the parallel case, the processing time is bounded by the slowest operation, which is the propagation.

6 Conclusions and future work

We have presented a system in which an application developer can construct a parallel image processing application with minimal effort. Data parallelism is captured by specifying the way to process a single pixel or object, with the system handling distribution, border exchange, etc. Task parallelism of these data parallel operations is achieved through an RPC system, preserving the semantics of normal function calls as much as possible. Results from an actual prototype architecture have shown that the system works, and can achieve a significant speedup by using an SIMD processor for low-level vision processing.

The automatic skeleton instantiation is currently limited to ILP processors, and we wish to include XETAL and IMAP skeletons as well. Furthermore, we want to investigate dynamic image sizes and data types. Finally, an automatic mapping step should combine CPU-, memory-, and bandwidth usages to best determine buffer sizes and assign operations to processors.

This work is supported by the Dutch government in their PROGRESS research program under project EES.5411.

References

[1] A. Abbo, R. Kleihorst, L.Sevat, P. Wielage, R. van Veen, M. op de Beeck, and A. van der Avoird. A low-power parallel processor IC for digital video cameras. In *Proceedings of the 27th European Solid-State Circuits Conference, Villach, Austria*. Carinthia Tech Institute, September 18–20 2001.

[2] Celoxica Limited. *Handel-C Language Reference Manual*, 2003.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989. ISBN 0-273-08807-6.

[4] E. de Kock, G.Essink, W. Smits, P. van der Wolf, J.-Y. Brunel, W. Kruijtzer, P. Lieverse, and K. Vissers. YAPI: Application modeling for signal processing systems. In *Proceedings of the 37th Design Automation Conference (DAC2000)*, pages 402–405, June 5-9 2000.

[5] R. Halstead, jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[6] P. Jonker and W. Caarls. Application driven design of embedded real-time image processors. In *Proceedings of Acivs 2003 (Advanced Concepts for Intelligent Vision Systems)*. Ghent University, September 2-5 2003.

[7] S. Kyo, S. Okazaki, and I. Kuroda. An extended c language and compiler for efficient implementation of image filters on media extended micro-processors. In *Proceedings of ACIVS 2003*, pages 234–241. Ghent University, September 2-5 2003.

[8] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2001.

[9] C. Nicolescu and P. Jonker. EASY PIPE - an "EASY to use" Parallel Image Processing Environment based on algorithmic skeletons. In *Proceedings of the PDIVM Workshop*, 2001.

[10] P.Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 341–354. ACM Press, 1994. ISBN 0-89791-688-3.

[11] F. Seinstra and D. Koelma. Lazy parallelization: A finite state machine based optimization approach for data parallel image processing applications. In *Proceedings of the PDIVM Workshop*, 2003.

[12] G. Slavenburg. *TM1000 Databook*. TriMedia Division, Philips Semiconductors, 1997.