A REAL-TIME PARALLEL IMAGE-PROCESSING MODEL

M. Fleury, H. Sava, A. C. Downton and A. F. Clark

University of Essex, UK

Introduction

Recently, a number of generalised parallel computation models have emerged, for example McColl (1). Efforts also persist (Rinard et al (2)) to fit the serial model of processing to parallel hardware. The demise of the transputer and the world-wide retrenchment in the computer industry (Pfister (3)) have added impetus to the search for one universal programming model. Image processing can be viewed as a data-reduction pyramid with pixel-based, low-level, processing at its base and semantically-based, high-level, processing at the apex. It is the base of the pyramid which represents the bottleneck. However, it is unclear whether the parallel computation models so far proposed address the needs of low-level image processing.

This paper therefore proposes a real-time parallel processing model for image processing. A Fast Fourier Transform (FFT) for batch-processing of images illustrates the model in a distributed workstation environment. The model has also been implemented on a dedicated modular parallel machine, a Transtech Paramid (Fleury et al (4)), a C40 parallel DSP network (Sava (5)), and the Unix-like real-time operating system VxWorks. The intention is to provide a common processing environment across accessible parallel architectures, including development support tools for the application design process from initial sequential simulation, through parallel decomposition to embedded parallel application.

A Common Processing Model

Main Features

The components of the common parallel processing model are as follows:

The Communicating Sequential Processes (CSP) model of parallelism (Hoare (6)) is selected as an efficient model of parallelism. CSP also enables abstract reasoning about parallelism, in particular about program correctness. CSP has wide dissemination in Europe in the wake of the transputer.

CSP provides a space in which event ordering may be nondeterministic. However, this is not a problem for the programmer as it can be made to provide efficient utilisation of the underlying hardware. There are two important features from the efficiency standpoint:

1. the ability to alternate responses in a non-deterministic fashion; and
2. low-overhead context shifting, by means of threads.

CSP as implemented in the programming language `occam 2` is not sufficient for image processing because extensive use of shared-memory is needed to avoid excessive memory-to-memory data movements. Unfortunately, on recent hardware, improvements in memory access speed significantly lag and hence may obviate enhancements in processor speed.

Buffers are employed at the user process level to mask communication latency and to increase bandwidth. Input buffers reduce the time spent waiting for work (or speed-up waiting) and output buffers smooth out access to the return channel. Buffer access contention is regulated by counting semaphores, which do not restrict access untowardly.

Demand-based data farming provides a flexible way of scheduling work in most cases. Where this form of scheduling is not possible the place of the central data-farmer is taken by a central data-manager. Centralized coordination is invariably required. Thread scheduling, as in CSP, is by a FIFO queue.

The chief practical impediment to demand-based data farming is that it may be impossible to extend the set of worker processes in a single farm because of competition for bandwidth. In other words, as it stands the data-farm is not a fully scalable solution because of practical difficulties of setting up the initial data distribution. Dynamic link switching between a set of small subordinate farms by the farmer processor is proposed to address the scaling problem (Fleury et al (7)).

Where global communication is needed, it can be supported within a set of worker processes by a combination of centralised message switching and inter-worker-module links Fleury et al (8). A generic set-up which will work in farming mode and in global mode is shown in Figure 1.

Message records are provided, the equivalent of `occam's` protocols. To enable reuse the communication structure should be transparent to the type of application messages.

Asynchronous multicasts from farmer processes are supported. A multicast enables efficient distribution of global data. A multicast does not initiate any reply messages, thereby restricting circular message paths.

The CSP model of parallelism is static. Low-level image processing routines, being generally deterministic, are unlikely to require unpredictable patterns of communication or computational needs. A static model also facilitates CSP's `channel` communication construct which provides an automatic name space, without the need for name-servers.

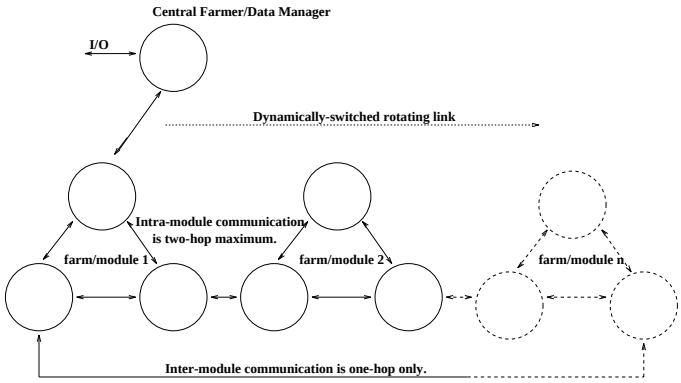

Figure 1: Generic Set-up

A Practical Extension

In practice, image-processing algorithms do not function in isolation but exist as part of a multi-algorithm application. The processing model is extended to cope with this requirement by replicating the farm structure in a Pipeline of Processor Farms (PPF). A suitable decomposition maps an existing sequential program onto the pipeline. Design rules for the decomposition which enable real-time constraints, such as throughput and pipeline traversal latency, to be met are already available in Downton et al (9), though static application behaviour is presently needed for an accurate decomposition.

The development cycle is:

- Time the components of the algorithm in a sequential setting.
- Decompose the sequential application into a pipeline of parallel components, each load-balanced using a processor farm.
- Account for dynamic costs such as communication and work-flow distributions.
- Test the concept in a distributed environment, with the benefit of the familiar Unix operating system. Unexpected message orderings will require debugging.
- Transfer the application to the target machine which has the same common processing structure. Performance debug the application, identifying hold-ups by means of execution traces.

A Pipelined Example

The design process initially consists in identifying suitable decompositions of sequential application code. Generally, PPF is appropriate to multi-algorithm applications but a single algorithm serves to outline the method.

The form of an FFT is well known. However, unfamiliar code can be analysed in a semi-automatic fashion, with the help of top-down profilers. Attention is concentrated on functions which take up a significant portion of the code runtime. The `Quantify` profiler (Pure Software (10)) is preferred to `gprof` as it counts machine cycles rather than relying on a statistical analysis of procedure calls (Pond and Fatemen (11)). Figure 2 is a screen shot of a call graph for a row-column (RC) 2D FFT size 360 (a mixed-radix Stockham autosort algorithm was used). In the centre, stemming from the `main` procedure, are the principal procedures ranked in time order. Subsidiary system calls are to the right. The three `Factor` functions are small-order FFTs, while `ReadMPgm` loads the image. Obviously, the RC transform can be split into input, 1D row transform, transpose, 1D column transform and output stages. At a lower-level of granularity the small-order transforms could form a 1D row (column) FFT pipeline but the communication overhead would be prohibitive on workstation-based hardware. Similarly, taking advantage of the linearity of the Fourier transform to decompose 1D row or column transforms by an overlap and add method is too expensive but may be appropriate for VLSI. The need for a design to adapt to differing architectures is a theme of PPF.

Figure 3 is an idealised timing diagram for a Fourier transform pipeline, showing that perfect steady-state overlap is achievable. The processing power,

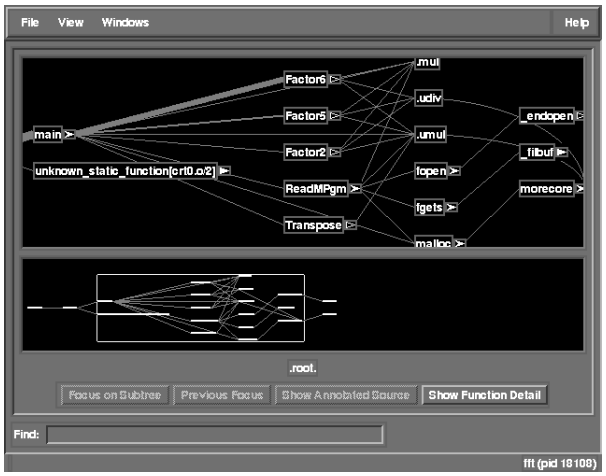

Figure 2: Call Graph of a 2D FFT

implemented in our case on Sun 4 architecture machines (through the socket application programmer's interface), must then be arranged to balance three stages of a pipeline, Figure 4.

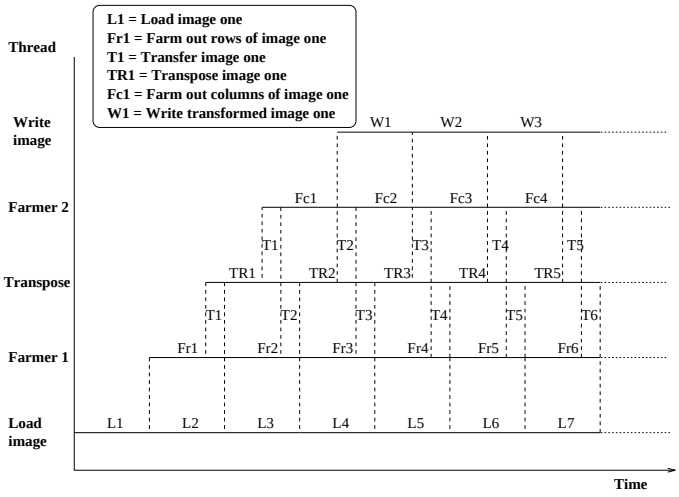

Figure 3: Ideal Timing of an FFT Pipeline

The number of processors needed in the farms depends critically on the the time to transpose the image matrix since the two farms being balanced can otherwise grow with the problem size (and are incrementally scalable). The transpose time varies with image size and the properties of the memory hierarchy (van Loan (12)). A variety of sequential transpose algorithms (Figure 5) are possible. If the block size is chosen appropriately to match the cache line size gains may be made. Cache lines vary from 4 to 128 bytes on recent processors. On the microSPARC II processor (50 MHz nominal clock speed) of the SPARCstation 5 employed in the tests the data cache has 16 byte lines. Figure 6 shows the mean

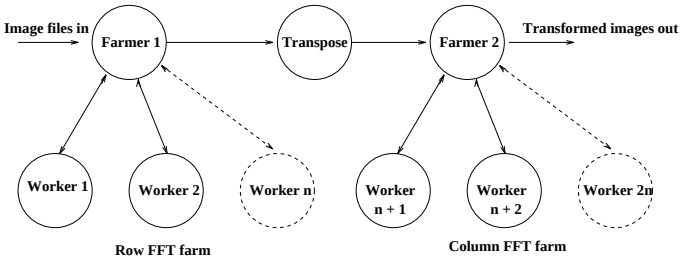

Figure 4: Logical Topology of an FFT Pipeline

of ten tests on complex-valued images to determine the optimum block size, when little effect is discernible. When the block-by-block method was compared to the diagonal method (appropriate to multiprocessor interleaved memories) and the obvious row-to-column method, it was the latter that performed best, Figure 7.

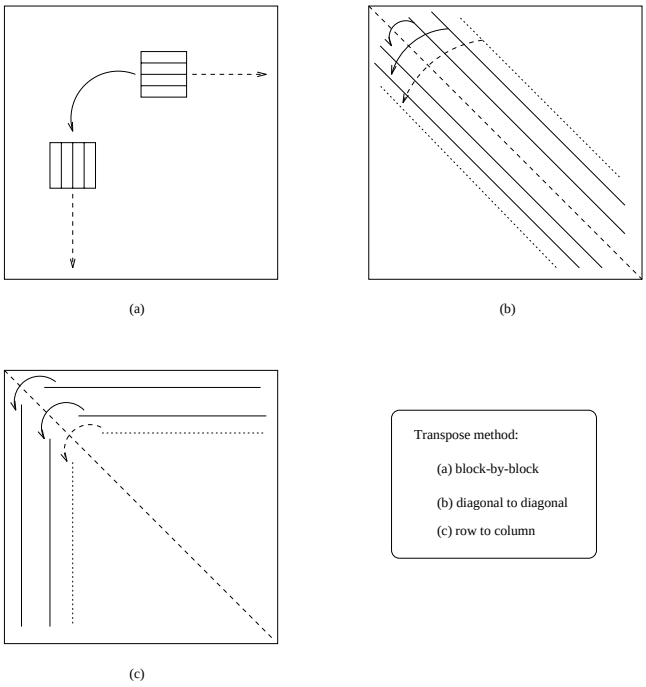

Figure 5: Image Transpose Algorithms

Having established the best transpose algorithm, timings for that algorithm were taken as a fixed sequential overhead. The computation time was also recorded for one of the 1D FFT phases across a number of processor types (in the SPARC family). This enables an estimate to be made of the number of processors needed in the FFT phases to balance a pipeline, if there is just one processor performing the transpose. The estimate is best case as no account of communication overhead has been taken. Figure 8 shows that the number of processors in each of

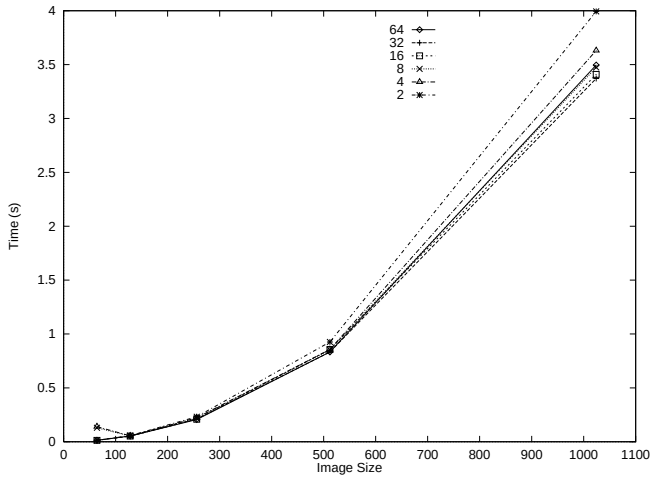

Figure 6: Block-by-block Transpose Timings

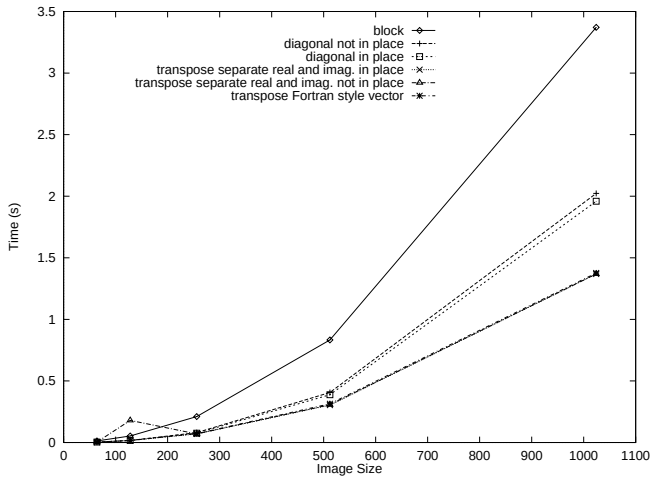

Figure 7: Other Transpose Algorithm Timings

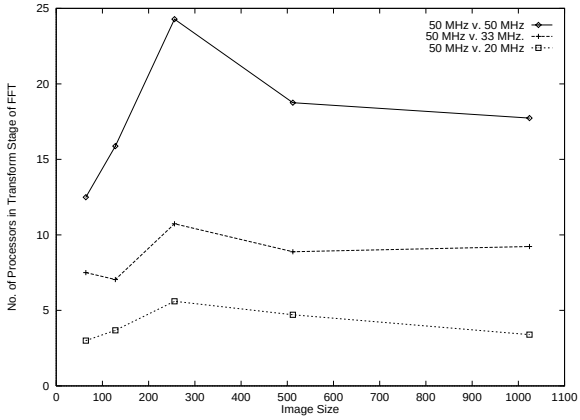

Figure 8: Pipeline Processor Numbers for Differing Processors in the Transform and Transpose Stages

the row and column data-farms will be rather large if the pipeline stages are homogeneous. The size of the image does not have an important effect on the scaling of the pipeline stages. Any variation is caused by the advantage of those image sizes which are an exact power of four, since a small order size four transform is more efficient than a size two transform, though the timing for image size 256 appears anomalous. If a lower speed processor is placed in the transpose stage fewer processors are needed in the FFT stages.

A simplified thread layout for the model is shown in Figure 9. I/O threads take up slack as the farmers may otherwise be under-utilized. Once a proposed decomposition has been determined, based upon sequential profiling, the application programmer is responsible for adapting the existing sequential transpose code and FFT, but all other code, implementing parallel communication and support functions, is part of a generic PPF template.

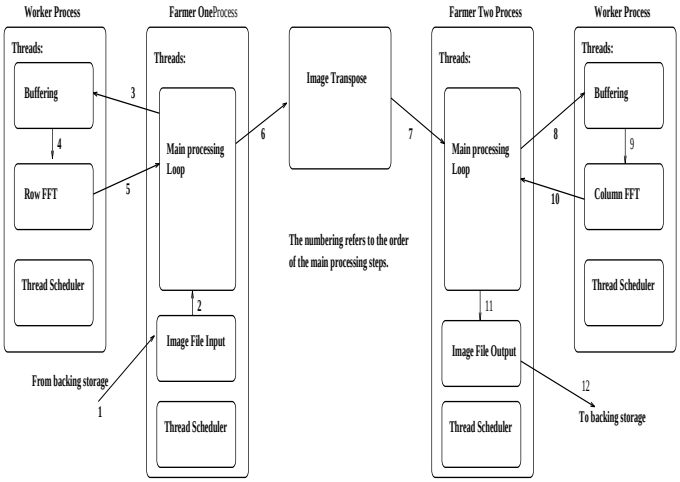

Figure 9: Threads in an FFT Pipeline

An additional feature of the template is a built-in trace mechanism. A logical clock system (Raynal and Singhal (13)), recording message ordering, has been implemented in order to event-stamp traces. (Uncertain propagation delay in a local network will restrict the resolution of a real-time clock if it relies on synchronisation messages in a distributed setting.) Figure 10 is a snapshot from the implementation. At this stage in development, the display is on the post-mortem visualizer, ParaGraph (Heath and Etheridge (14)). The plotted lines represent messages from one process to another. Processor 0 is farmer 1. For clarity, there is one worker in farm 1, processor 1. Processor 2 is the transpose and processor 3 is the second farmer. To the right of Figure 10, the transpose is receiving the next image from farmer 1, while farmer 1 also

handles work from its farm. Meanwhile, farmer 2 is passing work back and forth to its worker (processor 4). Figure 10 confirms through message ordering the possibility of completely overlapped processing. The actual degree of overlap can be judged in the distributed setting only approximately by elapsed wall-clock time. For the final stage of performance debugging, a trace based on a global real-time clock on the target machine has been implemented to pinpoint inefficiencies (Fleury et al (15)).

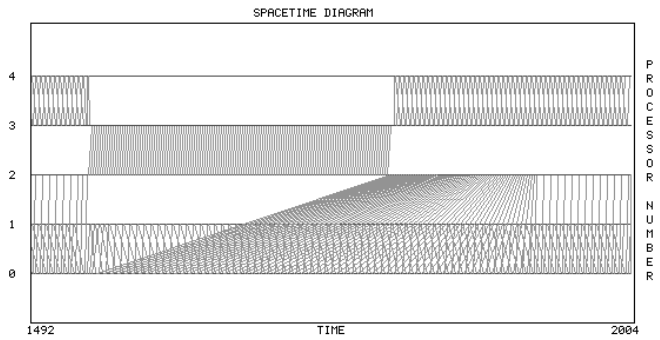

Figure 10: Trace of the FFT Pipeline

Concluding Remarks

A common processing model is proposed for real-time image processing. The model may be seen as a relaxation of an existing static model of parallellism, CSP. The justification for the model is based on providing an efficient software structure for embedded real-time applications. In a distributed workstation environment the processing model is envisaged in a prototyping role, providing a full range of development support facilities. The distributed environment represents a low-cost entry point. In a dedicated parallel machine, performance tuning facilities would be available. Timings reveal less than 5% overhead from event tracing in typical application regimes, in which case, provided real-time constraints are met, the monitoring structure might be left *in situ*. Appropriate account should be taken of the differences in architecture between the machines on which an algorithm is simulated and the target machine. An FFT exemplar shows that the transpose stage's performance may vary through difficult to predict interactions between a particular transpose algorithm and the memory hierarchy and this would effect the balance of the pipeline. Heterogeneous processors across stages of the FFT pipeline will decrease processor costs.

Acknowledgement

This work is being carried out under EPSRC research contract GR/K40277 'Portable software tools for embedded signal processing applications' as part of the EPSRC Portable Software Tools for Parallel Architectures directed programme.

References

1. McColl, W. F., "Scalable Computing", LNCS, 1000, 46–61, 1995.

2. Rinard, M. C., Scales D. J., Lam M. S., "Jade: A high-level machine-independent language for parallel programming", IEEE Computer, 26, 6, 28–38, 1993.

3. Pfister, G. F., "In Search of Clusters: the Coming Battle of Lowly Parallel Computing Systems", Prentice-Hall, Upper Saddle River, NJ, 1995.

4. Fleury, M., Sava, H., Downton, A. C., Clark, A. F., "Designing and instrumenting a software template for embedded parallel systems", in "UK Parallel '96", 163–180. Springer, London, 1996.

5. Sava, H., Fleury, M., Downton, A. C., Clark, A. F., "Fast implementation of discrete wavelet transform based on pipeline processor farming", elsewhere in IPA'97.

6. Hoare, C. A. R., "Communicating Sequential Processes", Prentice-Hall, Englewood Cliffs, NJ, 1989.

7. Fleury, M., Hayat, L., Clark A. F., "Performance Estimation for a Dynamically Reconfigurable Multiprocessor System Applied to Low-level Image Processing", in "Proceedings of the Conference on the Performance Evaluation of Parallel Systems (PEPS '93)", 209–213. Univ. of Warwick, 1993.

8. Fleury, M., Hayat, L., Clark A. F., "Parallelizing Grey-scale Coordinate Transforms", IEE Proc. Pt. I VISP, 142, 4, 207–212, 1995.

9. Downton, A. C., Tregidgo, R. W. S., Çuhadar, A., "Top-down structured parallelisation of embedded image processing applications", IEE Proc. Pt. I VISP, 141, 6, 431–437, 1994.

10. Pure Software Inc., 1309 Sth. Mary Ave. Sunnyval. CA, "Quantify User's Guide", 1992.

11. Ponder, C., Fateman, R., "Inaccuracies in Program Profilers", Software P. & E., 18, 5, 459-467, 1988.

12. van Loan, C., "Computational Frameworks for the Fast Fourier Transform", SIAM, Philadelphia, 1992.

13. Raynal, M., Singhal, M., "Capturing Causality in Distributed Systems", IEEE Computer, 29, 2, 49–56, 1996.

14. Heath, M., Etheridge, J., "Visualizing the performance of parallel programs", IEEE Software, 8, 5, 29–39, 1991.

15. Fleury, M., Downton, A. C., Clark, A. F., Sava, H., "The Design of a Clock Synchronization Sub-system for Parallel Embedded Systems", IEE Proc. Pt. I CDT, accepted for publication, 1997.