Multi-Paradigm Framework for Parallel Image Processing

David Johnston, Martin Fleury, Andy Downton
University of Essex
Multimedia Architectures Laboratory
Colchester, Essex, CO4 3SQ, UK
djjohn@essex.ac.uk

Abstract

A software framework for the parallel execution of sequential programs using C++ classes is presented. The functional language Concurrent ML is used to implement the underlying harness and to design the programming interfaces. The hardware-independent harness promises a composable multi-paradigm, unified approach to parallelism across different technologies: PowerPCs, DSPs and FPGAs. Performance results for an image processing case study are given.

1 A Parallel Framework

This paper presents a software framework currently under development for the parallel execution of applications. The source of the application remains in a sequential form such as C++ with no annotations, language extensions, or library calls for parallelism. A framework in this context is defined to be the the totality of the software infrastructure that maps this sequential expression to parallel execution. The framework consists of application class hierarchies, a pre-compiler and an underlying "harness" which is the executable code for controlling the parallel application. Only the bottom layers of the harness are hardware dependent.

The resulting application code can run on any conventional serial machine, or on any parallel machine for which a suitable harness has been written, because the form is uncommitted to any type of parallel hardware. The multi-paradigm nature of the harness enables the simultaneous exploitation of various common forms of parallelism (*e.g.* data, pipeline and algorithmic parallelism) as well as less regular parallelism via inherent extensibility. The programmer is thus unrestricted and may use different composable parallel paradigms.

The harness dynamically monitors performance and optimises parallelisation parameters such as granularity during execution, and thus may be described as "self-tuning".

Such an approach is suitable for embedded real-time high-bandwidth systems in continuous operation where this self-tuning phase is short compared to the lifetime of the execution run *e.g* radar target tracking. However, the approach is also applicable to programs with a shorter run where data from previous runs are available.

The sequence of events runs as follows: the classes used indicate which types of parallelism can be exploited; a "pre-compiler" examines the type signatures (or templates) of function calls using these classes to identify exploitation opportunities; the profiler shows where this has been worthwhile; and the harness iteratively retunes its parallelisation parameters accordingly. To exploit parallelism the software will need to be (re-)written to use framework application classes. However, only as much or as little of an existing application as necessary need be changed in order to obtain the performance benefits required.

The remainder of this paper describes how different parallel paradigms can be captured in a software framework and how these are mapped to hardware. The commonest technique "data decomposition" serves as illustration, showing that applications that do not fit the traditional mould can still be captured (see Section 6). The use of a functional language for developing Application Programmer Interfaces (APIs) and concurrent harnesses is discussed, and the results of parallelising an image processing application following this model are presented. Finally, future work for the RaPPID[1] project is described.

2 Hardware

The software framework addresses a set of applications requiring critical real-time responses, but given the cost of developing software also requires a lifetime of up to twenty years. Similar software lifetimes [2] are seen in the avionics industry, where the software must adapt over its lifetime

---

to successive generations of hardware, and indeed should support software synthesis, *i.e.* the transfer from specialist hardware to general-purpose hardware running at increased rates. Radar systems [3] are illustrative of large-scale, embedded applications with sample rates now approaching 500 MHz. Multidimensional and non-linear signal-processing techniques are being used for 2D and 3D imaging of terrain or objects within Synthetic Aperture Radar (SAR), and target-tracking radar is now required to give estimates of the velocity of moving targets. Algorithm changes within an application are endemic, though there are a few staples such as the FFT. I/O bandwidth may be critical and airborne systems will require compact, low-power solutions. This type of application is perhaps the antithesis of consumer applications, being large-scale with few systems, but without the transient lifetime of many applications designed for the mass market.

A two-level computer model has been proposed [11] that will consist of an invariant network-level structure and a variant node-level structure. An embodiment of the two-level computer in given in Figure 1. The first level consists of a Sparc host with a Myrinet [4] NIC (Network Interface Card), itself controlled by a LANai communication coprocessor. The host is responsible for initialization of its computation coprocessor and subsequent message handling. The computation coprocessor consists of five FPGAs with on-board NIC. The FPGAs are connected to the host by an 8-way crossbar to form what Myrinet call a System Area Network. (The FPGAs can also be connected locally in a ring topology for exchange of global results.) Higher-level connectivity is via Myrinet LAN, which replicates, for commodity processors, a form of interconnect that has been common on supercomputers such as the Cray T3D.
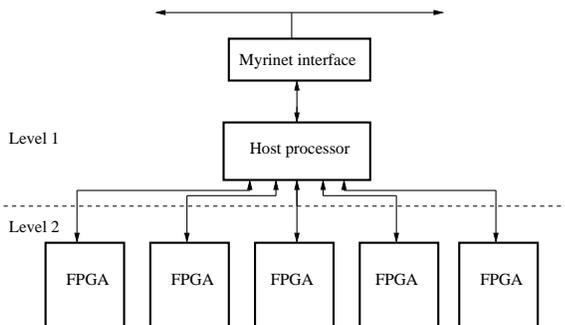
Figure 1. Two-level computer example.

From Figure 2, it can be seen how this architecture is applied to radar target tracking. After data-capture and pre-processing of images by SAR, stage two, Focus of Attention (FOA), is responsible for extracting regions of interest within images with potential targets, *e.g.* moving vehicles. Second-level detection (SLD), performs simple time-domain template matching, which is embarrassingly parallel. The same processing structure can be applied to sonar beam-forming. However, in detailed studies [7] for time-delay, and frequency-space beam-forming, it was established that there are non-obvious trade-offs between DSPs and FPGAs in terms of, respectively, memory access blockages caused by irregular addressing patterns, and a future bottleneck caused by a limit to the number of i/o pins available on an FPGA. These hardware vagaries illustrate the need for a flexible software structure.
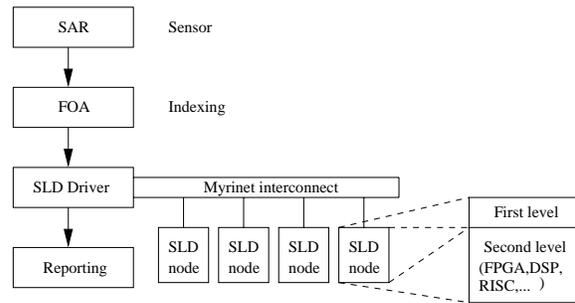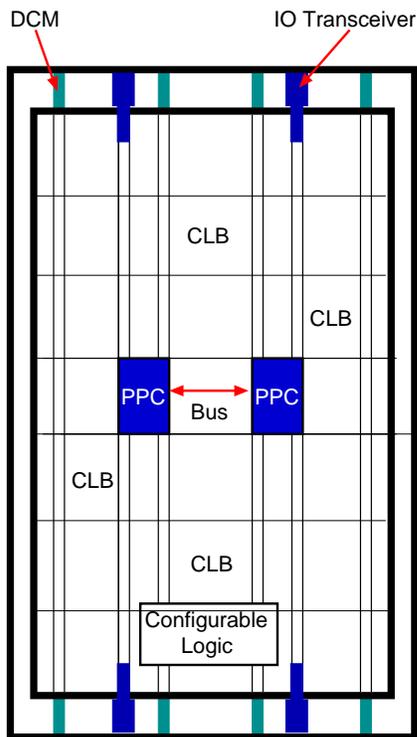

Figure 2. ATR pipeline.

Interestingly from the point of view of the need for software synthesis, an SMP (Symmetric Multiprocessor) is an alternative architecture [1] with fewer beams but interpolation of samples. A 12-processor Sun Ultra Enterprise achieved a speed-up of eleven with a throughput of 4.8 GFLOP/s. Word-level SIMD parallelism within the SPARC VIS instruction set was used, together with multithreading.

Turning to the design of individual nodes, there has been an observable shift towards soft architectures (which are runtime configurable), and away from a fixed hardware. Microprocessors with embedded FPGA coprocessors, such as the Chameleon CS112 [12] are now available. Xilinx's Virtex-II Pro platform FPGA [13] shown in Figure 3 is at the time of writing available on evaluation boards, and is the most recent embodiment of the architecture targeted by this framework. The Virtex-II Pro incorporates up to four embedded PowerPC 405 cores connected by an on-chip IBM CoreConnect bus; up to eight RocketIO 3.125 Gb/s transceivers; up to 12 Digital Clock Managers (DCMs) and other sundry logic items all on the one chip. These are to be found within a sea of Configurable Logic Blocks (CLBs) that constitute the atoms of the FPGA. The Virtex-II Pro clearly displays the same two-level computer model.

2.1 Project Hardware

The wider objective of the project is to move towards an uncommitted form of parallelism in order to provide unified

Figure 3. Virtex-II Pro FPGA

Single Point-to-Point Communication		
Communication Direction		Data Bandwidth
PPC	$\rightarrow$ PPC	44 MB/s
PPC	$\rightarrow$ x86	49 MB/s
x86	$\rightarrow$ PPC	44 MB/s

Table 1. achievable Myrinet performance

daughterboards conform to the PCI Mezzanine-Card (PCM) format. These hardware decisions proved wise in retrospect - see the analysis of parallel performance in Section 6.

Figure 4. Target Hardware Schematic

support for a wider range of parallel and indeed hybrid parallel architectures with such components as Digital Signal Processors (DSPs), conventional Central Processing Units (CPUs) and Field Programmable Gate Arrays (FPGAs - see Section 2.3). Support for the ubiquitous Networks of Workstations (NOWs) architecture is planned as part of this drive. However, the preliminary approach of the project is to use a conventional multi-processor network that is well-established in the embedded system domain.
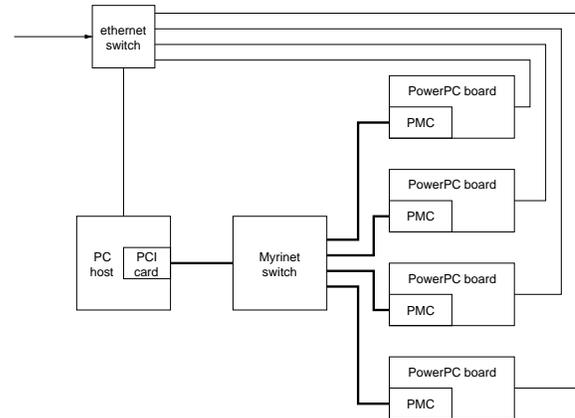
2.2 Multi-Processor Network

The initial target hardware, shown in Figures 4 and 5, consists of four PowerPC (PPC) processors connected by a high speed Myrinet network. Efficient parallelisation is often limited by the lack of inter-processor bandwidth and it is significant that our choice of hardware prioritised communication performance over node performance. Note that a Myrinet link is also provided to the host PC, to ensure that the bandwidth to the host (generally I/O) is not unbalanced compared to the rest of the system. This link also allows the host PC to become one of the processing nodes if required - albeit one with differing processing characteristics. The communication capability is supplied via a Myrinet PCI card within the PC, and four Myrinet PCI daughterboards, each housed on one of the four VME PowerPC cards. The

The nodes are 450 MHz PPC, while the host PC is 800 MHz. The Myrinet network is a switching one supporting point-to-point full-duplex 2+2 Gb/s links. Of course, the best realisable data rates are considerably less than the raw hardware ratings suggest, and test software provided by Myricom gives the results summarised in Table 1. These suggests that the current weak-point is the receiving end, as performance picks up when receiving is done on the higher specification processor. Figures 6 and 7 show the effect of varying message size on bandwidth and latency respectively.

Although not truly processor to processor, an attempt was made to take x86 $\rightarrow$ x86 timing measurements by starting off the "send" and "receive" ends on the same x86 workstation. This did not work, so loopback software was used instead on both the x86 and PPC. In this case, messages do the full round trip, so bandwidth is down and latency is up. The loopback software unsurprisingly shows better performance on the faster processor.
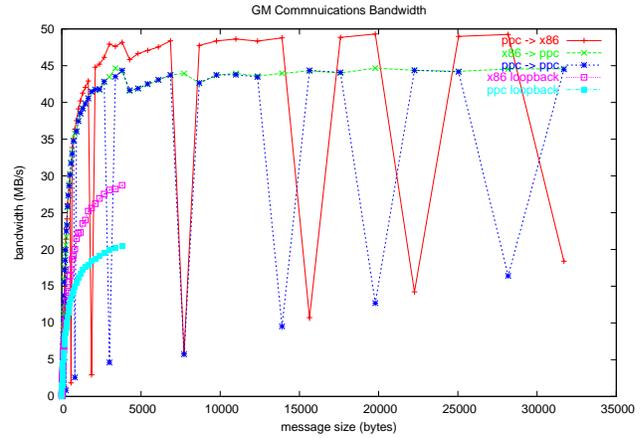
3
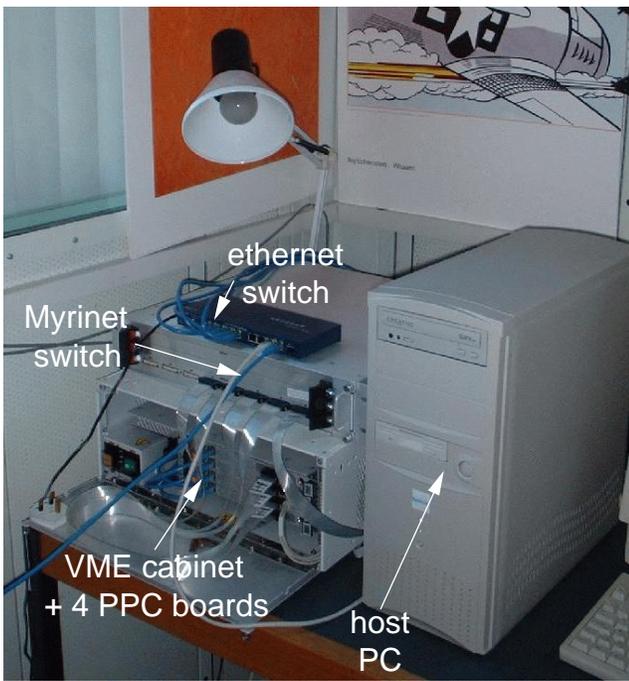
Figure 5. Target Hardware

Figure 6. Myrinet bandwidth

Figure 7. Myrinet latency

Where the PPC is the sending processor in a non-loopback communication, there are extreme anomalies in performance for certain message sizes. Figure 7 shows the effect is caused by the latency jumping by multiples of around 600 μS. There is perhaps a problem with the PPC software as no such effect occurs on the x86, although it may be an artifact of the scheduling quanta of the operating system on the PPC. Otherwise, latency increases linearly with message size, showing that intermediate memory stores are involved.
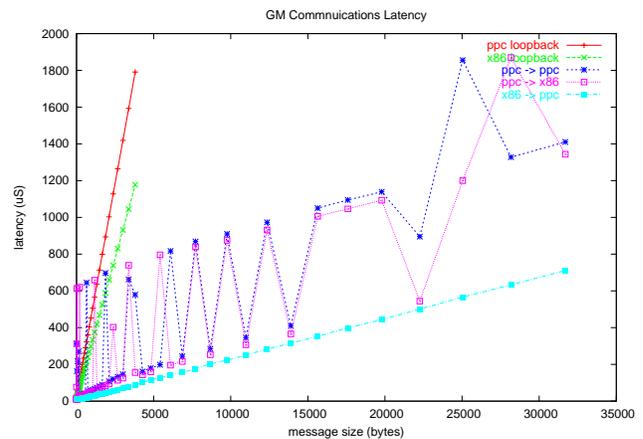
Message sizes have to be about 4 kbytes for the maximum point-to-point bandwidth to be achieved. The best performance observed was around 50 Mbytes/s ≈ 0.4 Gb/s which is "of the order" of theoretical GM bandwidth (which is either 1 or 2 Gb/s) but still falls rather short.

2.3   FPGA

A particularly interesting emerging target architecture is the FPGA. Given the increasing gate count of such devices it is becoming possible to directly compile down increasing amounts of a high level language (such as 'C') into silicon. Hardware is inherently parallel and the potential to exploit massive parallelism on such devices is clear. The difficulties of doing so are of expression (representing suitable concurrency using a conventional software language which is generally inherently sequential) and of implementation (what

model to employ to map a line of software into a hardware structure).

One commercial solution is Handel C [5], which is a concurrent C-based programming language modelled upon CSP (Communicating Sequential Processes) [8]. It is compiled down to a hardware description language (VHDL or usually EDIF). There are two distinct ways of expressing concurrency in Handel C.

- a synchronous pipeline model

  Successive lines of code are directly pipelined in hardware.

- an asynchronous model

  Communication occurs between concurrent code modules via channels constructed in hardware.

The overall clocking of the system is dictated by the slowest line of code - a fundamental tenet of the Handel-

C programming model is that each line of code, however complex, takes one clock tick to execute and this in turn affects the maximum speed of the clocking possible.

Of course, this "rule" could be relaxed and compilers could translate a single line of user code into several lines of hardware code, but the effect of this would be rather too chaotic at today's level of technology! The asynchronous model decouples the need for different hardware modules to explicitly synchronise *i.e.* it would be very difficult to ensure that two modules executed exactly the same number of lines of code between successive communications, especially for any kind of large-scale code development with conditional branches. So while a tuned Handel-C program may look radically different from the starting point, the advantage of this approach for FPGA development is that this starting point of a high level language (in this case 'C') exists.

The product moves the increasingly massively parallel capability of an FPGA from the remit of the hardware engineer (in ever dwindling supply) into the domain of the software engineer. And while a software approach will not be optimal, it is an open question whether in the longer term the difference in performance will be regarded as insignificant as the difference between assembler and high level language usage, especially given the ever increasing levels of genuine hardware parallelism that FPGAs can support. The correspondence between this asynchronous model particularly and that of message passing architectures is clear, and provides the scope for unification.

3. Templates

Sub-divisions to achieve parallelism can be of data (divide and conquer); code (algorithmic parallelism) or time (pipelining). In fact pipeline parallelism is a sub-case of algorithmic parallelism, where the code is representable by a linear rather than a general graph. For illustration, let us formalise a generalised divide and conquer template in a functional language such as CML (Concurrent Meta-Language [10]). Just as values and variables have types in CML so do functions. `atan2` has type `real * real -> real` because it maps a pair of real numbers to another real number. Software modules, called "structures", in CML bundle together types, values and functions but **not** variables. There is no concept of a variable in a functional language, so one can think of a CML structure as a class without state. This, in a single sweep, automatically removes the source of many programming errors. A structure too has a type or "signature" which consists of the types of the components that are externally visible. Provided a user writes a CML structure to match the following signature, then the user's software can be executed in parallel:

```
signature SIG =
sig
    type input;
    type output;
    val process : input -> output;
    val combine : output * output -> output;
    val split   : input -> input * input;
end;
```

This argument extends to languages other than CML, but the Object-Oriented (OO) formulation for the target language (C++) would complicate the discussion. Once the user's software is in the correct form, the C++ precompiler intercepts the process function and replaces it with a call to the harness. The harness manages the parallel execution of the task, splitting the input recursively until the required granularity is achieved or until further splitting would become meaningless. How much of this software does the programmer have to supply? The input and output type variables and the process function are inherited from the original application, thus the only additional work for the programmer is to supply the split and combine functions. However with certain data types such as geometric grids in certain circumstances, the way such structures should be split and combined is obvious and can be pre-supplied by the system instead. If the split or combine or both are non-standard and have to be supplied by the user, then it is possible that the derived class can be reused as a new customised paradigm, giving extensibility. Similar abstractions are provided to capture geometric, algorithmic, and pipeline parallelism.

4. Protocol Stack

This section compares the protocol stack of a traditional parallel execution harness (the left of Figure 8), with that proposed by the RaPPID project (the right of Figure 8).
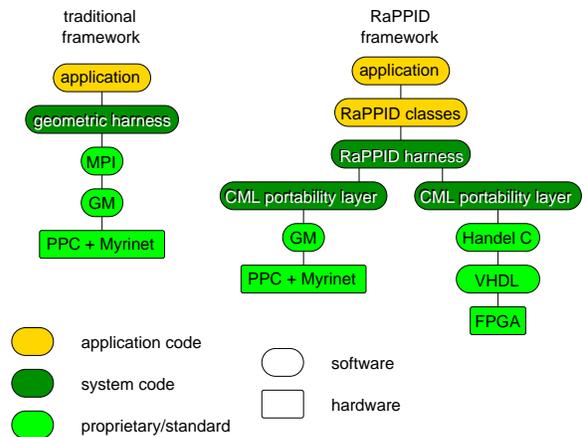

Figure 8. Protocol Stacks

4.1. Traditional

The traditional protocol stack model was used to port an Augmented Reality (AR) application [9] to the message-passing architecture of the PPC multi-processor network described in Section 2.2. This constitutes a baseline system for future comparison. The harness was written on top of the MPI (Message Passing Interconnect) message passing standard which in turn uses the Myrinet proprietary GM (Glenn's Messages) communication library. In fact, the Hardhat Linux running on each node was sufficiently minimal that it did not allow the use of an off-the-shelf MPI implementation, and the MPI layer had to be written from scratch.

4.2. RaPPID

Significantly, the alternative RaPPID harness is written in CML. The portability layer supports the small and elegant CML model which is based on CSP just as Handel-C. Adopting a clean CSP model allows the harness to be supported on a variety of architectures, and Figure 8 shows how this may unify the programming of FPGAs and message passing architectures.

5. Functional Prototyping

Functional languages [6] offer exceptionally rapid prototyping. Their power of expression leads to high level compact code, that can be altered at minimal cost. A harness is parameterised by an application, and suitably a functional language allows application code to be handled as any other data *e.g.* code and data can be sent down communication channels. The APIs and the harness themselves have been developed through the use of CML, which provides powerful constructs for concurrency and communication and removes concerns of memory management and clean process termination: there are garbage collectors for both! Several generations of software are necessary before an API is established - and functional rapid prototyping was used to shorten this process. In CML, communication occurs synchronously across channels. A channel is typed, and can only be used to transmit data of that type. The `send` and `recv` functions are actually not CML primitives, but are derived instead from typed channel event primitives `sendEvt` and `recvEvt` as follows:

```
val send = sync o sendEvt;
val recv = sync o recvEvt;

(* o indicates functional composition     *)
(* sync waits for the event to be enabled *)
```

These simple primitives and derivations contrast sharply with the plethora of buffered/unbuffered, synchronous/asynchronous communication functions in MPI.

A short but complete CML program provides an insight into the model. The main process spawns processes A and B. These are connected by two channels. B receives one of two communications from A - the first that (indeterminately) becomes possible. Note: the use of events does not commit to a communication (allowing a clean model of selection). Note also the exceptionally compact expression of a complete parallel operation.

```
fun A ch1 ch2 () =
sync( choose[ sendEvt(ch1, "hello 1"),
              sendEvt(ch2, "hello 2") ] );

fun B ch1 ch2 () =
let val msg = sync( choose[ recvEvt(ch1),
                            recvEvt(ch2) ] );
in print ( "received " ^ msg ^ "\n" ) end;

fun main () =
let val ch1 = channel(); val ch2 = channel();
in spawn (A ch1 ch2); spawn (B ch1 ch2) end;

RunCML.doit ( main,  NONE );
```

How far can CML be taken on the journey towards a final parallel target machine? The PPC system runs a primitive realtime version of Linux on its nodes and so any harness must interface to the GM 'C' library. Similarly for FPGAs, any harness must link to Handel-C code. Unexpectedly and fortuitously, past prototyping experience [9] has shown that it is possible to port CML code to 'C' at the last minute. Although the programming model is radically different, one can manually port CML to 'C' roughly on a line-to-line basis. Tools are also available to automatically translate CML to 'C', though the output is virtually indecipherable and thus impossible to interface to other 'C' code. Overall, this means that the harness should run on top of a 'C' layer, but should itself stay in CML, or at least a closely-allied form. The CML portability layer should be as small and clean as possible. This aim is well supported by the minimal elegance of CML itself.

6. Sample Application and Results

The following case study involves the region processing stages of an AR application. The approach is to initially parallelise the application using a traditional harness and manual tuning. Once the novel harnesses have been developed, parallel execution is re-attempted. The performance comparisons give a direct measure of the efficiency of the second approach.

The serial operation of the region processing is shown in Figure 9. A 256-level greyscale image (i_{256}) is adaptively filtered to a 3-level greyscale image (i_3) to enable a subsequent region decomposition. The connectivity and geometry of the region image (i_r) are derived and respectively described by a graph (g) and a table (t) data structure. These

| Application Phase Communication	
Phase	Data Bandwidth
scatter	28 MB/s
gather	56 MB/s
accumulate	12 MB/s

Table 2. application bandwidths

are then used to identify known visual targets placed within the scene and thereby work out the camera position in 3D space.

The initial manual parallelisation provided good parallel efficiency. Indeed, it would be pointless to proceed to test a semi-automated system when a successful parallelisation is not possible. However, the application was chosen because the compute intensive final stage of composing pixels into distinct regions does not fit a purely traditional geometric approach. Figure 10 shows the parallelisation of the algorithm, where for the sake of simple illustration the image is processed in only two halves: a left-hand sub-image $(i_{256})_L$ and a right-hand sub-image $(i_{256})_R$. Each half-image can be processed independently, except that prior to filtering, edge data has to be exchanged and the graph and table results have to be eventually combined in a non-trivial accumulate operation. The sequence of communications is as follows:

1. host scatters input image to worker processors

2. worker processors swap edge data

3. accumulate processor gathers result image fragments

4. accumulate processor accumulates graphs and tables

It should be noted that it is not strictly necessary to gather the result image, but this is useful to check the correct operation of the code. The accumulate operations form a binary tree with associated processing at each branch node, whereas the gather operation is merely a many-to-one collation. However, in our simple example with two worker processors, the communication patterns are indistinguishable. The total data bandwidths measured during these communication phases are shown in Table 2. These are respectable compared to the "raw" figures in Table 1, especially as these are application figures with an additional layer of a home-brew MPI implementation underneath. Indeed the gather performance indicates that two simultaneous point-to-point links (the maximum possible in our configuration) must be in simultaneous operation as it exceeds the raw performance of a single link. The scatter performance is notably half the gather performance, while the accumulate figure is comparatively less useful, as it also involves a degree of computation.
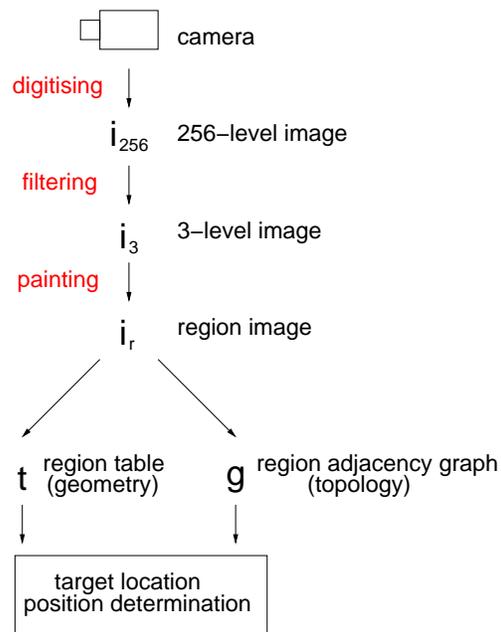
Figure 9. serial image processing

No	Parallelisation Type			
CPUs	Simple		+ Combine	
1	7.5 fps	(100 %)	8.7 fps	(100 %)
2	12.0 fps	(80 %)	16.4 fps	(94 %)
4	19.6 fps	(66 %)	31.5 fps	(90 %)

Table 3. performance (frames per second) and efficiency of AR application

Overall, there is a useful mix of computation in the application, because the initial image processing is simply geometrically parallel in contrast to the later accumulation. Moreover, the image-processing stages are themselves suitable for future pipelining. The parallel efficiency was first measured with simple geometric parallelism used on its own, and then additionally using the parallelism obtained through the accumulation operation. The results in Table 3 show that to obtain a significant speed-up, especially for larger numbers of processors, the more difficult parallelism also needs to be tackled.

7. Discussion

The current harness is static and needs to be enhanced to dynamically respond to the monitored performance as well as to be ported to native parallel hardware. Similarly, the calls to the harness have been manually placed into the application code, whereas these should be automatically edited in by a pre-compiler.
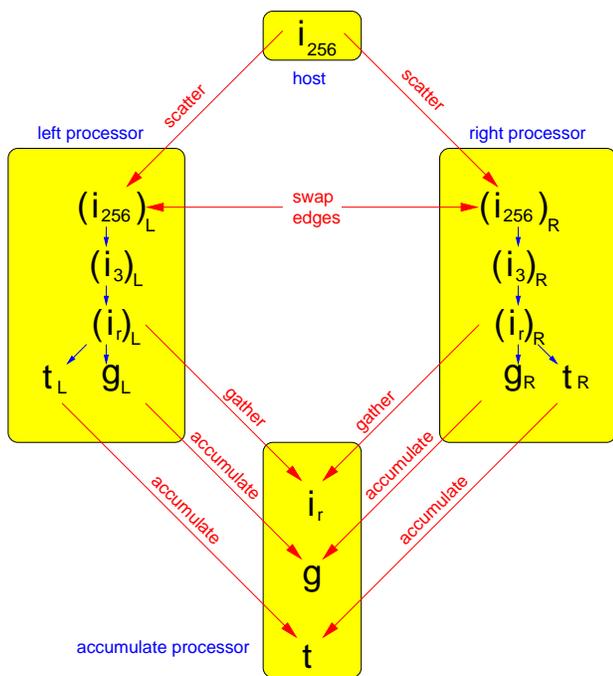
Figure 10. parallel image processing

8. Conclusion

The path to parallelism described in this paper has proven encouraging on two fronts:

- good parallel efficiency of tricky-to-parallelise applications written to fit the proposed API model

- positive software engineering experience of functional language use in the concurrent domain

For the manual parallelisation, the complication of producing an "accumulate" function was added to by the necessity of handling all communication and all concurrency as additional work loads. The aim of the project is not to spare the application programmer from the task of writing the accumulate operator, but of ensuring that this is the only burden and that the framework which accepts this operator is independent of hardware architecture. In short, the programmer's role is related to the application task at hand not the parallel system being used. The strength of the approach is to support multiple simultaneous paradigms of parallelism, both those which are standard and those which are semi-customised - certainly the irregularity of the test application described precludes the use of a conventional harness.

References

[1] G. E. Allen and B. L. Evans. Real-time sonar beam-forming on a unix workstation using process networks and PTHREADS. *IEEE Transactions on Signal Processing*, 48(3):921–926, 2000.

[2] N. Audsley, I. Bate, and A. Grigg. Portable code: Reducing the cost of obsolescence in embedded systems. *Computer Control Engineering*, 10(3):98–104, 1999.

[3] L. H. J. Bierens. Hardware design for modern radar processing. *Electronics & Communication Engineering*, pages 257–270, December 1997.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, pages 29–36, 1995.

[5] M. Fleury, R. P. Self, and A. C. Downton. Hardware compilation for software engineers: an ATM example. *IEE Proceedings Software*, 148(1):31–42, 2001.

[6] S. Gilmore. Programming in standard ml'97: A tutorial introduction, 1997.

[7] P. Graham and B. Nelson. Frequency-domain Sonar Processing in FPGAs and DSPs. In *IEEE Symposium on FPGAs for Custom Computing Machines, FCCM'98*, 1998.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, first edition, 1985. ASIN: 0131532715.

[9] D. J. Johnston. *Position Sensing and Augmented Reality*. PhD thesis, University of Essex, 2002.

[10] J. H. Reppy. Concurrent ML: Design, application and semantics. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, pages 165–198, 1993.

[11] C. Seitz. A prototype two-level multicomputer, 1996. Project information on DARPA/ITO project available from www.myri.com/research/darpa/96summary.html.

[12] R. J. X. Tang, M. Aalsma. A Compiler Directed Approach to Hiding Configuration Latency in Chameleon Processors. *Field-Programmable Logic and Applications*, 20(3):29–37, 2000.

[13] Xilinx Inc., San Jose, CA. *Virtex-II Pro Platform FPGA User Guide*, 2002.