An Event-Based Execution Model for Efficient Image Processing on Workstation Clusters and the Grid

D. J. Johnston, M. Fleury, and A. C. Downton
Electronic Systems Engineering Department, University of Essex,
Wivenhoe Park, Colchester, CO4 3SQ, United Kingdom

Abstract

The event model of the functional language CML (Concurrent Meta Language) is used to capture concurrency, which normally remains unexploited within conventional parallel geometric harnesses running on (virtual) arrays of processors. This complexity is hidden from the application programmer, who merely supplies conventional geometric sequential code which is automatically executed in parallel. An example of a low-level image filtering operation is used, to show how execution efficiency can be maintained in spite of the communication delays and indeterminacies encountered in real networks.

1. Introduction

Pattern recognition applications demand two (currently) irreconcilable attributes: rapid code prototyping to explore algorithm space and the high performance that is sometimes only achievable through parallel execution. Parallel programming is a difficult activity and adds to the cognitive burden of the algorithm implementer. Moreover, the specialised nature of parallel application code forces a divergence of the application source code, when time pressure to prototype future versions causes the parallel version to become abandoned with the parallel hardware on which it ran.

Committing certain sections of application code to specific models of parallelism limits the concurrency that may be exploited, and restricts the nature of future algorithmic change. This paper presents a development method that allows source code to stay in a serial form, yet supports transparent parallel execution provided the signature of algorithms within that application match parallel paradigms for which *software frameworks* have been written. A common approach to parallelism, followed by this paper, is the use of such system software frameworks or *harnesses* which mediate the execution of conventional sequential segments of application code. In particular, implementation details for a generic parallel geometric harness are presented, which has been written to the machine-independent event model of the concurrent functional language CML (Concurrent Meta Language [3]). The power of the event model is to enable good execution performance across a range of different parallel architectures, especially when communication bandwidth between computational nodes is low or unpredictable. The example used to demonstrate this point is a 2D image filtering operation.

Workstation clusters and the Grid [4] suffer from unpredictable communication bandwidths and latencies, so harnesses expecting synchronous operation (such as the phase of edge-data exchange with adjacent processors encountered in parallel filtering) may exhibit sub-optimal performance. The event model is able to dynamically respond to the indeterminacy present within real networks, and to re-order execution so that no matter which data have not arrived, as much as possible is done with that which has arrived, in the hope that this execution time will mask the communication time of the delayed data.

The original CML system runs as a single process and uses pseudo-parallelism for concurrent operation. Concurrency simplifies considerably the expression of certain applications (such as windowing systems) and CML was invented specifically for this purpose. However, this work novelly examines the use of the CML model within a genuine multi-computer environment, and finds it to be the correct tool for the job as delayed messages need not impact execution efficiency. The authors have extended CML to work (with constraints) in a multi-computer environment to implement the ideas presented, though this work [2] is outside the scope of this paper.

The main technical content of this paper is the contrast between a conventional parallel geometric harness (Section 2) and a parallel geometric harness constructed using the event model (Section 3). The conclusions (Section 4) reflect briefly upon the value of the event model and the overall approach to parallel application development.

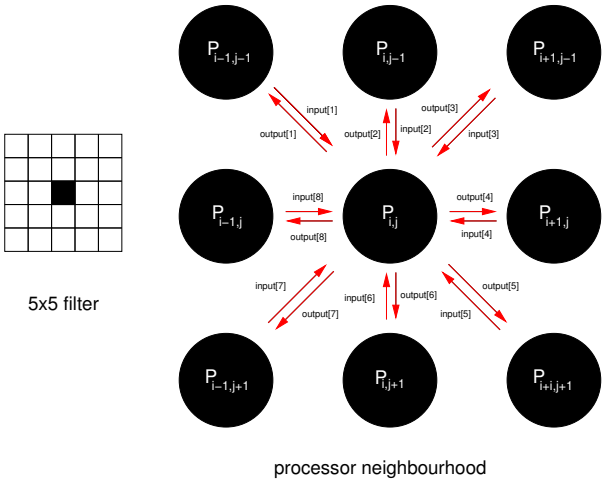

Figure 1. Filter and processor neighbourhood

2. A Traditional Geometric Harness

One of the most fundamental but compute intensive image-processing operations is the application of a spatial filter. The filter used for illustration has dimension 5×5 as shown in 1. Though the parallelism inherent in this operation is clear (the filter is applied to each pixel within the image), parallelising the filtering algorithm on a distributed memory parallel architecture is complex as an "adjacent pixel" may be on another processor. In other words, before filtering can occur boundary data needs to be swapped.

Figure 1 shows the adjacent compute nodes for each processor $P_{i,j}$ in a 2D array and the names of the interconnecting communication channels. Figure 2 illustrates the eight regions of boundary data ($S_i, 1 \leq i \leq 8$) which must be sent from each processor and the direction of the corresponding channel.

For convenience, each local array of image data is provided with a halo of (initially unassigned) regions in which to receive adjacent boundary data ($R_i, 1 \leq i \leq 8$ as shown in Figure 3) so that the filtering code remains transparent to the distributed implementation *i.e.*filtering occurs purely within an array in local memory. The core of the local array is divided into a number of regions that can be computed independently ($C_i, 0 \leq i \leq 8$ as shown in Figure 3) . For an array z, the following notation is used to refer to the various sub-arrays, and is also used as "pseudocode" in the exemplar listings:

$z[R_i], 1 \leq i \leq 8$	receives data from direction i
$z[S_i], 1 \leq i \leq 8$	data sent to direction i
$z[C_i], 0 \leq i \leq 8$	separately computable region

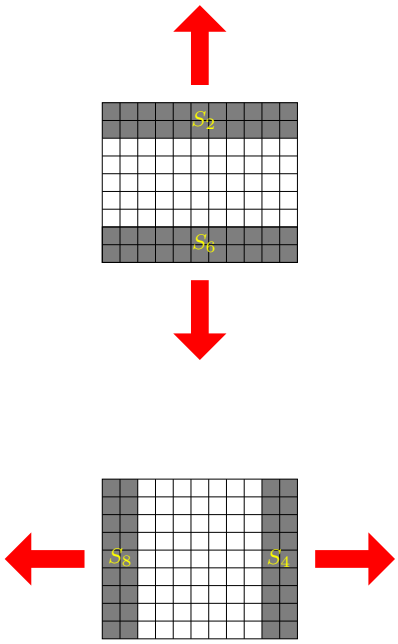

Figure 2. Sent regions

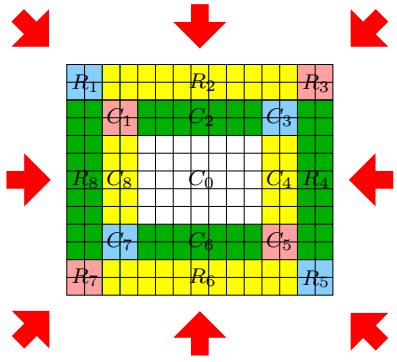

Figure 3. Received and calculated regions

```
SEQ
  PAR i = 1 for 8
    PAR
      input[i]  ? z[R[i]] -- receive data
      output[i] ! z[S[i]] --    send data
  -- implicit bulk synchronisation of 16 processes
  z' := filter ( z )
```

Figure 4. traditional occam harness

```
fun async_send(ch, z) =
        spawn ( fn () => send(ch, z ) );
fun async_recv(ch, z) =
        spawn ( fn () => z := recv(ch) );

val   send_tids =
tabulate (* eight asynchronous sends    0 <= i < 8 *)
(
    8 ,
    fn i => async_send( output[i+1],     z[S[i+1]] )
);
val recv_tids =
tabulate (* eight asynchronous receives 0 <= i < 8 *)
(
    8 ,
    fn i => async_recv(  input[i+1], ref z[R[i+1]] )
);
(* explicit synchronisation of 16 processes *)
val _ = map (sync o joinEvt) (send_tids @ recv_tids);
val z' = filter z;
```

Figure 5. traditional CML harness

A traditional parallel implementation swaps halo data and then performs the filtering operation. An `occam` [1] expression of the software on each computational node is given in Figure 4 for comparative purposes. `z` represents this node's segment of the input image and `z'` this node's segment of the filtered output image. Space within `z` and `z'` has been allowed for the reception of halo data from neighboring processors. The traditional CML equivalent is shown in Figure 5. It is not necessary to understand the functions `async_send` and `async_recv` which perform asynchronous output and input on a channel respectively, but the definitions are given for completeness. However, it is worthwhile noting that the `spawn` primitive is CML's way of starting a new process. `send_tids` holds a list of the Task IDentifiers (TIDs) of the eight sending processes; while `recv_tids` holds a list of the TIDs of the eight receiving processes. The `joinEvt` primitive takes a TID and turns it into a termination event for the associated process. The filtering does not proceed until the composite list of TIDs has first been converted into termination events which are then individually synchronised upon.

Note that both with `occam` and CML the various processes "spawned" share the array `z` between them (*i.e.*shared memory is used between processes on the local node). However, there is no overlap in the areas being written. The areas being read do overlap but these are not written. In short there are no race conditions.

Although these harnesses allows the various communications to occur in parallel, all communication must occur before any calculation begins. This is over-constraining the order of execution. For example as soon as $z[R_2]$ is received then $z'[C_2]$ may be computed. As soon as all of $z[R_2]$, $z[R_3]$ and $z[R_4]$ are received then $z'[C_3]$ may be computed. This capability for over-lapped communications and computation is not captured in the code.

3. An Event-Based Geometric Harness

The event model of CML provides a way of "triggering" a computation as a result of a successful receive event, using the `wrap` operator. The first stage is to form a communication event on a channel, say, `input[2]`:

```
val e2 = recvEvt(input[2]);
```

This does not actually perform any action, but can be thought of as creating a potential for receiving on that channel. After data has been received, they should ultimately be incorporated within array `z`, and let us suppose the function which does this is called `incorporate`.

```
fun incorporate array received_data =  ...
(* incorporates data received into array
       side effect not result significant  *)
```

It is possible using the `wrap` operator, to wrap the `incorporate` function around event `e2` to form a new event `a2`. The `wrap` primitive creates one event from another using a function (its second argument) to change the datatype of the input event (its first argument) into the datatype of the output event.

```
val a2 = wrap( e2, incorporate z[R[2]] );
```

As soon as the data received is packed into the array some filtering can proceed to create an event that signifies the completion of processing.

```
val c2 = wrap( a2, fn () => filter z[C[2]]);
```

By symmetry, cases `c4`, `c6` and `c8` are the similar to `c2`. The data in $z[C_3]$ can be processed once events `a2`, `a3` and `a4` have occurred. `AND` is the appropriate event operator that creates an event which fires once both its operand events have fired. The implementation of `AND` is trivial but outside the scope of this paper.

```
val c3 = wrap (
                  a2 AND a3 AND a4,
                  fn () => filter z[C[3]]
             );
```

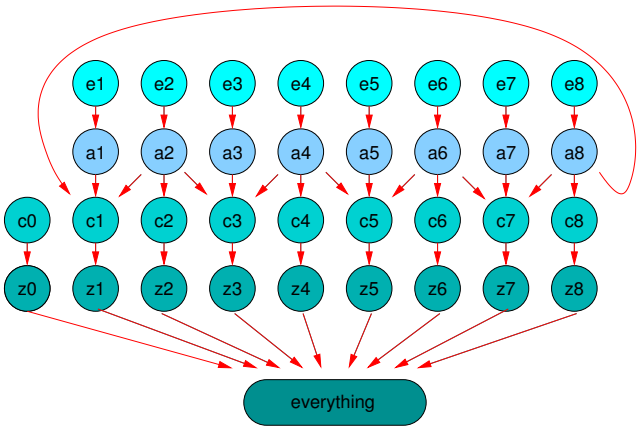

Figure 6. Event/execution graph

By symmetry cases `c1`, `c5` and `c7` are similar to `c3`. The data in $z[C_0]$ does not require any event to have happened before computation can proceed, so a processed event is created from wrapping the event which is always true by the filtering function.

```
val c0 = wrap (
                  AlwaysEvt (),
                  fn () => filter z[C[0]]
             );
```

More events are created to indicate the incorporation of each filtered subimage into the result image, for example:

```
val z3 = wrap (c3, incorporate z'[C[3]] );
```

Finally, an event in created for everything having been completed:

```
val everything = z0 AND z1 AND z2 AND z3 AND
                 z4 AND z5 AND z6 AND z7 AND z8;
```

`everything` is an event driven execution graph as shown in Figure 6 which represents the totality of the filtering computation and associated receiving communications. To actually enable execution a top-level synchronisation is required *i.e.*

```
val _ = sync(everything);
```

Certain events are instrumental in triggering more than one event. For example, `a2` is contributory to events `c1`, `c2` and `c3`. So instead of firing once, event `a2` has to fire three times, so the usual event mechanism of CML has to be extended somewhat so an event that is synchronised more than once returns what it did the first time. The `event` type is extended to a `hold_event` type which holds a reference to an instance of the event datatype if the event has fired already, or a reference to `NONE` if the event has not yet fired. An implementation is provided in Figure 7, including a function to create a `hold_event` from an `event` and a modified synchronisation operator.

```
(* 'a option shows if data of type 'a is present *)
datatype 'a option = NONE | SOME of 'a;

type 'a hold_event = 'a event * ('a option) ref;

fun event_to_hold_event e =
      (e, ref NONE) : 'a hold_event;

fun hold_sync
      ((e, ref (SOME(a))): ''a hold_event) = a
     (* event already fired case:
        use previously stored value *)

 |  hold_sync
      ((e, r as ref NONE): ''a hold_event) =
      let
         val result = sync(e);
      in
         ( r := SOME(result); result)
      end;
      (* event not fired case:
         sync and store value *)
```

Figure 7. Modified event type

4 Conclusion

The overall approach attempts to bring parallel execution, rather than parallel programming, transparently into the mainstream. The underlying event-based execution model, captures parallelism within harnesses that can accommodate indeterminacy in communication. If data is received on any one channel before any other, then any permitted associated computations will be performed as soon as it is possible instead of all computations being delayed until all communication is finished.

The significance of this capability becomes clear as we move from multi-processors on a dedicated switched network (high probability of synchronous operation) through multi-computers on an Ethernet (such as workstation clusters where there is a lower probability of synchronous operation) to a Grid-based system (almost no possibility of synchronous operation). The use of the event model decouples artificial dependencies in traditional imperative code and provides a dynamic response to network conditions to promote parallel execution efficiency of applications such as the image filtering example.

References

[1] H. C. Ed. *Occam 2 Reference Manual*. Prentice-Hall, 1988.

[2] D. J. Johnston, M. Fleury, and A. C. Downton. Event Based Model to Support Distributed Parallelism with CML and CSP, 2003. `http://privatewww.essex.ac.uk/~djjohn/rappid/`.

[3] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, UK, 1999.

[4] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. `http://citeseer.nj.nec.com/595640.html`.