Building BSP Programs Using the Refinement Calculus

D.B. Skillicorn*

Department of Computing and Information Science
Queen's University, Kingston, Canada
skill@qucis.queensu.ca

Abstract. We extend the refinement calculus to permit the derivation of programs in the Bulk Synchronous Parallelism (BSP) style. This demonstrates that formal approaches developed for sequential computing can be generalised to (some) parallel computing environments, and may be useful for managing some of the details of programming in such situations.

1 Introduction

An important problem in parallel software construction is *structure-directed refinement*, that is the ability to take account of properties of the execution environment during the development of a parallel program. Almost all existing parallel programming environments do not have ways of capturing this, which means that programs must be developed informally, and their properties determined only by executing them. This in unsatisfactory because it prevents proper engineering design, and limits portability.

General-purpose parallel computation aims to find an abstraction or model that conceals much of the complexity of parallel computation. Such an abstraction forms a target for software development independent of architecture specifics, and hence justifies investment in tools and techniques for software development.

One possible choice for a general-purpose model is *bulk synchronous parallelism* (BSP). Because it separates computation from communication, and communication from synchronisation, it is a particularly clean and simple approach. These separations allow us to add a handful of laws to the refinement calculus and derive BSP programs within it.

One attractive feature of applying formal software development techniques to parallel computing is that the application of a single transformation will often be a large, global step in the development of a program. Such transformations in sequential computing often only affect single statements, and so are small steps. Thus the overhead of formal techniques may be better amortised in a parallel setting.

2 Bulk Synchronous Parallelism

The *bulk synchronous parallelism* model [6] is a general-purpose model in which the properties of architectures are captured by four parameters. These are: the raw speed of

* This work was supported by the Natural Sciences and Engineering Research Council of Canada

the machine (which can be ignored by expressing the remaining parameters in its units), the number of processors, p, the time required to synchronise all processors, l, and the ability of the network to deliver messages under continuous load, g. Computations are expressed in *supersteps* which consist of local computations in each processor, using only local values, and global communication actions, whose results take effect locally only at the end of the superstep. The cost of a superstep is given by

$$cost \;=\; w + hg + l$$

where w is the maximum local computation in any processor during the step, h is the maximum number of global communications into or out of any processor during the step, and l is the cost of a barrier synchronisation to end the step.

Although it imposes a modest structuring discipline on parallel programming, there is growing evidence that BSP helps with program construction. Because a BSP program's cost is parameterised by properties of the target architecture, it is possible to decide for which architectures, that is for which range of the model parameters, a particular parallel program will perform well. In some cases it has been possible to construct programs that will be optimal for any target architecture.

The organisation of programs as sequences of supersteps reduces the complexity of arranging communication and synchronisation, and results in series-parallel computation graphs for programs. This makes it straightforward to extend techniques for constructing sequential programs to BSP programs.

3 Extending the Refinement Calculus

The *refinement calculus* [4] is a methodology for building programs using a set of refinement laws. An initial specification is transformed by small refinements into an implementation which is (a) executable, (b) efficient, (c) but harder to understand than the original specification. The correctness of each refinement law guarantees the correctness of the implementation with respect to the initial specification.

A specification consists of three parts: a frame, the list of variables that the specification may change; a precondition; and a postcondition.

$$frame \;:\; [precondition,\; postcondition]$$

that is, specifications are predicate transformers. Some specifications, called *code*, correspond to programs in the intended execution model. Non-code specifications cannot be directly executed, but instead represent abstract computations.

The intuitive meaning of a specification is an abstract computation which, if started from a state satisfying the precondition, terminates in a state satisfying the postcondition. Starting from any other state, the specification's behaviour is completely arbitrary, including perhaps failing to terminate. Formally, the meaning of a specification is given by *weakest precondition semantics* so that

$$wp(f : [pre\,, post], otherpred) \mathrel{\widehat{=}} pre \;\&\; (\forall f \cdot post \Rightarrow otherpred)$$

Programs are constructed by *refinement*, written

$$spec1 \sqsubseteq spec2$$

(*spec2* refines *spec1*), a relation on programs defined by

$$spec1 \sqsubseteq spec2 \mathrel{\widehat{=}} wp(spec1, pred) \Rightarrow wp(spec2, pred) \text{ for all predicates } pred$$

The refinement calculus takes the view that variables are global and hence may be read anywhere in a computation. Writes to these global variables are controlled by the *frame* of a specification, a list of those variables that it is allowed to change. Since we do not wish to model a global memory, we begin by extending the frame to include those variables readable by a specification as well. A specification becomes

$$(readframe; writeframe) : [precondition\ , postcondition]$$

The read frame is defined to be those variables readable by the computation being specified, which we will take to be equivalent to those variables whose values are local to the processor in which the computation is executing. The write frame is defined to be those variables that may be written by the executing processor. Since we are modelling distributed-memory execution it is possible for the same variable name to appear in the write frame of concurrent specifications, but some care must be taken to ensure that, at the end of a concurrent specification, the value of only one such variable can be used in subsequent computations. Note that the write frame need not be a subset of the read frame.

We now introduce location predicates, which are intended to model information about which processors hold each value. This could be done by, for example, partitioning frames. Instead we will use predicates in pre- and postconditions.

A *location predicate* (*lp*) relative to a read or write frame is a conjunction of simple predicates of the form $p_i(x)$, where $p_i(x)$ holds exactly when processor i contains the value denoted by variable name x. The predicate is written $lp(f)$ when the frame concerned is not obvious. In other words, a location predicate gives the location(s) of each variable mentioned in a frame. A *disjoint location predicate* (*dlp*) is a location predicate in which each variable name appears at most once (that is, no value is contained in more than one processor). A BSP program is one whose derivation does not alter location predicates except using the new laws given here.

Result: All laws of the refinement calculus continue to hold in this extension.

Proof: Add a read frame containing all variable names in the scope to each specification. The semantics for read frames given above *is* the semantics assumed by the refinement calculus.

We now introduce new operations to model data movement in a BSP architecture and new laws for manipulating specifications involving location predicates and producing these operations as code.

We use the operations given in Figure 1. The operations **distribute**, **collect**, and **redistribute** move values between processors using the distributions implied by the location predicates. Parallel composition models the concurrent, independent execution

$Syntax$	$Semantics$
$\mathbf{distribute}(rf, lp(wf))$	$wp(\mathbf{distribute}, pred \ \& \ lp) \ = \ pred$
$\mathbf{collect}(dlp(rf), wf)$	$wp(\mathbf{collect}, pred) \ = \ pred \ \& \ dlp$
$\mathbf{redistribute}(dlp_1(rf), lp_2(wf))$	$wp(\mathbf{redistribute}, pred \ \& \ lp_2) \ = \ pred \ \& \ dlp_1$
$\parallel (rf_i, wf_i) : [pre_i, post_i]$	$wp(\parallel (rf_i, wf_i) : [pre_i, post_i], pred) =$ $\&_i wp((rf_i, wf_i) : [pre_i, post_i], pred)$

Fig. 1. Semantics of New Code

of sequential code on distinct processors. The BSP cost of each of these new operations is straightforward to compute.

Definition [Local Knowledge]: For any location predicate lp of the form $p(x) \ \& \ p(y) \ldots$,

$$(rf \, ; wf) : [pre, post] \equiv (rf \, ; wf) : [pre \ \& \ lp(rf), post]$$

This follows from the definition of a read frame – the values that are local are those that can be read by the program.

The validity of the remaining laws follows from the semantics in Figure 1.

Law [Distribute]:

$$(rf \, ; wf) : [inv, inv \ \& \ lp(wf)] \sqsubseteq \ \mathbf{distribute}(rf, lp(wf))$$

where **distribute** is code that transmits the values of variables in rf to processors so that lp holds. inv is any predicate (which is invariant).

Law [Collect]:

$$(rf \, ; wf) : [inv \ \& \ dlp(rf), inv] \sqsubseteq \ \mathbf{collect}(dlp(rf), wf)$$

where **collect** is code that collects the values of variables whose names appear in wf into a (conceptually) single location. Note that a disjoint location predicate ensures that each variable can be given at most a single value.

Law [Redistribute]:

$$(rf \, ; wf) : [inv \ \& \ dlp_1, inv \ \& \ lp_2] \sqsubseteq \ \mathbf{redistribute}(dlp_1(rf), lp_2(wf))$$

where **redistribute** is code that communicates the values of variables whose locations are indicated by dlp_1 into locations satisfying lp_2.

Law [Sequential Composition]: For any predicate *mid* and location predicate *lp*

$$(rf; wf) : [pre, post] \sqsubseteq (rf; wf) : [pre, mid \ \& \ lp] \ ; \ (rf \cup wf; wf) : [mid \ \& \ lp, post]$$

Only one more law is needed, to allow the sequential specifications within each superstep to be produced.

Law [Parallel Decomposition]: Whenever $\&_i \ post_i \Leftrightarrow post$

$$\begin{array}{l} (rf; wf) : [pre \ \& \ lp_1, post \ \& \ lp_2] \sqsubseteq \\ \quad \|_i \ (\{x : rf : p_i(x) \ in \ lp_1\}, (\{x : wf : p_i(x) \ in \ lp_2\}) : [pre, post_i] \end{array}$$

Note that

$$\begin{array}{l} \&_i wp((rf_i, wf_i) : [pre_i, post_i], pred) \\ \equiv \&_i (pre_i \ \& \ (\forall wf_i \cdot post_i \Rightarrow pred)) \\ \equiv pre \ \& \ (\forall wf \cdot post \Rightarrow pred) \end{array}$$

As a small example, it may happen that the data distribution at some point in the program is inappropriate for the desired next step. This requires the insertion of a redistribution like this:

$$\begin{array}{l} (rf; wf) : [pre \ \& \ dlp_1, post] \ \sqsubseteq \ \{sequential \ composition\} \\ \quad (rf; wf) : [pre \ \& \ dlp_1, pre \ \& \ lp_2] \ ; \ (rf; wf) : [pre \ \& \ lp_2, post] \\ \sqsubseteq \{redistribution\} \\ \quad \mathbf{redistribute}(dlp_1, lp_2) \ ; \ (rf; wf) : [pre \ \& \ lp_2, post] \end{array}$$

More substantial derivations can be found in [5].

Although derivations quickly become messy in terms of the symbols that are manipulated, they are conceptually straightforward, and it is clear that tools could (a) manage the manipulation of symbols, and (b) provide many of the transformations automatically or with minimal (say, choice from a menu) guidance by the developer.

4 Conclusions

We have shown how the refinement calculus can be extended to derive BSP programs. This goes some way towards fixing a weakness of the BSP approach, the lack of a formal way to derive programs and guarantee their correctness. As in all refinement approaches, the existence of a calculus does not provide explicit guidance for program construction. Rather it ensures that the details of an intuitive construction are handled correctly.

A more abstract approach to modelling BSP semantics can be found in work by He and Hoare [1, 3]. They model BSP supersteps as independent computations followed by a merge operator which appropriately resolves situations where the same variable

may have been written in multiple threads. This approach is semantically clean, but does not lead immediately to a software development methodology, although one is certainly possible. Our approach constrains and simplifies construction in precisely the same way that the refinement calculus constrains and simplifies reasoning using weakest preconditions.

The approach taken here allows programs containing nested parallelism to be generated. The present implementations of the BSP model do not allow this, but it is a natural extension to model clustered architectures. In any case, nested parallel constructs can be replaced by sequential ones without altering the semantics provided that distinct locations are provided for variables held in notionally distinct processors.

It is straightforward to extend the laws provided here to allow for other kinds of structured communication such as broadcasts and reductions. These are available within the Oxford BSP Toolset [2].

The refinement calculus can be similarly extended to model message passing. The essential idea is that both ends of an eventual communication must be marked before the threads involved are separated. While this works in a formal sense, it does not seem to be useful in a practical sense. As with all calculational approaches, some idea of the direction of the derivation must exist in the programmer's mind, and the sheer number of messages in a typical program would seem to make this kind of explicit management during derivation impractical.

References

1. J. He, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In *Proceedings of Europar '96*, number 1124 in Lecture Notes in Computer Science. Springer-Verlag, August 1996.
2. Jonathan M.D. Hill. The Oxford BSP toolset users' guide and reference manual. Oxford Parallel, December 1995.
3. C.A.R. Hoare and J. He. *Unified Theories of Programming*. Prentice-Hall International, to appear 1997.
4. C. Morgan. *Programming from Specifications*. Prentice-Hall International, 2nd edition, 1994.
5. D.B. Skillicorn. Building BSP programs using the Refinement Calculus. Technical Report 96-400, Queen's University, Department of Computing and Information Science, October 1996.
6. D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *Scientific Programming*, to appear. Also appears as Oxford University Computing Laboratory, Technical Report TR-15-96, November 1996.