Building BSP Programs Using the Refinement Calculus

D.B. Skillicorn
skill@qucis.queensu.ca

October 1996

External Technical Report
ISSN-0836-0227-
96-400

Department of Computing and Information Science

Queen's University

Kingston, Ontario K7L 3N6

Document prepared October 2, 1996

Abstract

We extend the Refinement Calculus to permit the derivation of programs in the Bulk Synchronous Parallelism (BSP) style. This provides a mechanism for constructing correct programs in this portable and efficient style.

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. There is widespread agreement that general-purpose parallel computation requires an abstraction or model that conceals much of the complexity of parallel computation. Taking this route requires targeting software development for this abstraction.

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

2 Bulk Synchronous Parallelism

The *bulk synchronous parallelism* model [2,3,5,6] is a general-purpose model in which the properties of architectures are captured by four parameters. These are: the raw speed of 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. A specification consists of three parts: a frame, the list of variables that the specification may change; a precondition; and a postcondition.

$$\mathit{frame} \; : \; [\mathit{precondition}, \; \mathit{postcondition}]$$

Specifications are regarded as programs. Certain specifications are called *code* and may be directly executed. Non-code specifications cannot be directly executed but represent abstract computations. Starting from a state satisfying the precondition, a specification terminates in a state satisfying the postcondition. Starting from any other state, the specification's behaviour is completely arbitrary, including perhaps failing to terminate.

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

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

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

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

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. Thus we are modelling execution on a distributed-memory architecture. 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 we will take some care to ensure that at the end of a concurrent specification only one such 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.

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

Figure 1: Semantics of New Code

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

Definition 2 (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 – those values that are local are those that can be read by the program.

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

Law 3 (Distribute)

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

where **distribute** *is code that transmits the values of variables in rf to processors so that lp holds.*

Law 4 (Collect)

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

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 5 (Redistribute)

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

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

Law 6 (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 7 (Parallel Decomposition) *Whenever* $\&_i\, post_i \Leftrightarrow post$

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

Note that

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

4 Examples

We illustrate the use of these laws with some small examples.

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:

$$\begin{aligned}(rf;wf):[pre\ \&\ dlp_1, post] \sqsubseteq\ & \{\textit{sequential composition}\}\\ & (rf;wf):[pre\ \&\ dlp_1, pre\ \&\ lp_2]\ ;\ (rf;wf):[pre\ \&\ lp_2, post]\\ \sqsubseteq\ & \{\textit{redistribution}\}\\ & \textbf{redistribute}(dlp_1, lp_2)\ ;\ (rf;wf):[pre\ \&\ lp_2, post]\end{aligned}$$

Now we illustrate the derivation of a complete BSP program. To reduce clutter we will use X_i for $\{x_i : i : 1 \le i \le n\}$, X_L for $\{x_i : i : 1 \le i \le n/2\}$, and X_R for $\{x_j : j : n/2{+}1 \le j \le n\}$.

$$(X_i, s; s) : [true, s = \sum_{i=1}^{n} x_i]$$

$$\sqsubseteq\ \{\textit{introduce local variable}\}$$

$$(X_i, s, s_1, s_2; s, s_1, s_2) : [true, s = \sum_{i=1}^{n} x_i]$$

$$\sqsubseteq\ \{\textit{algebra}\}$$

$$(X_i, s, s_1, s_2; s, s_1, s_2) : [true, s = s_1 + s_2\ \&\ s_1 = \sum_{i=1}^{n/2} x_i\ \&\ s_2 = \sum_{j=n/2-1}^{n} x_j]$$

$$\sqsubseteq\ \{\textit{sequential composition}\}$$

$$(X_i, s, s_1, s_2; s, s_1, s_2) : [true, s_1 = \sum_{i=1}^{n/2} x_i\ \&\ s_2 = \sum_{j=n/2-1}^{n} x_j]\ ;\ \textsf{(A)}$$

$$(X_i, s, s_1, s_2; s, s_1, s_2) : [s_1 = \sum_{i=1}^{n/2} x_i\ \&\ s_2 = \sum_{j=n/2-1}^{n} x_j, s = s_1 + s_2\ \&\ s_1 = \sum_{i=1}^{n/2} x_i\ \&\ s_2 = \sum_{j=n/2-1}^{n} x_j]\textsf{(B)}$$

Now using P_1 for $p_1(X_L)$, P_2 for $p_2(X_R)$, and S for

$$s_1 = \sum_{i=1}^{n/2} x_i\ \&\ s_2 = \sum_{j=n/2-1}^{n} x_j$$

we continue:

$$\begin{aligned}\textsf{(A)}\ \sqsubseteq\ & \{\textit{sequential composition}\}\\ & (X_i, s, s_1, s_2; s, s_1, s_2) : [true, P_1\ \&\ P_2]\ ;\ \textsf{(C)}\\ & (X_i, s, s_1, s_2; s, s_1, s_2) : [P_1\ \&\ P_2, P_1\ \&\ P_2\ \&\ S]\ ;\ \textsf{(D)}\\ & (X_i, s, s_1, s_2; s, s_1, s_2) : [P_1\ \&\ P_2\ \&\ S, S]\ ;\ \textsf{(E)}\end{aligned}$$

(C) $\sqsubseteq$ $\textbf{distribute}(X_i, s, s_1, s_2; P_1 \mathbin{\&} P_2)$

(D) $\sqsubseteq$ $\{\textit{parallel decomposition}\}$

$$(X_L; s_1) : [true, s_1 = \sum_{i=1}^{n/2} x_i] \parallel (X_R; s_2) : [true, s_2 = \sum_{j=n/2-1}^{n} x_j]$$

$\sqsubseteq$ $\|_2$ *code for sequential summation*

(E) $\sqsubseteq$ $\{\textit{collect}\}$

$\textbf{collect}(p_1(s_1) \mathbin{\&} p_2(s_2), s)$

(B) $\sqsubseteq$ $\{\textit{assignment}\}$

$s := s_1 + s_2$

It is easy to see how such derivations can be extended to partition the list over a larger number of processors, and how the final gather step can be rewritten as a reduction with addition of the results of the individual processors.

5 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.

The approach taken here allows programs containing nested parallelism to be generated. The present version of the BSP model does 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 [1].

References

[1] Jonathan M.D. Hill. The Oxford BSP toolset users' guide and reference manual. Oxford

Parallel, December 1995.

[2] W. F. McColl. An architecture independent programming model for scalable parallel computing. In J. Ferrante and A. J. G. Hey, editors, *Portability and Performance for Parallel Processors.* Wiley, 1994.

[3] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*, Cambridge International Series on Parallel Computation, pages 337–391. Cambridge University Press, Cambridge, 1993.

[4] C. Morgan. *Programming from Specifications.* Prentice-Hall International, 2nd edition, 1994.

[5] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. Technical Report TR-15-96, Oxford University Computing Laboratory, August 1996.

[6] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.