Partial Evaluation Scheme for Concurrent Languages and Its Correctness

Haruo Hosoya, Naoki Kobayashi and Akinori Yonezawa

Department of Information Science, University of Tokyo,
7–3–1 Hongo, Bunkyo–ku, Tokyo 113, Japan

Abstract. A simple, general, and well-formalized partial evaluation method for concurrent languages is proposed. In spite of many potential benefits, there are few partial evaluation techniques for concurrent languages. We choose a process calculus for the target language because it has theoretical clarity, and yet has expressive power enough to represent various high-level constructs in concurrent object-oriented languages. We realize effective optimization by allowing elimination of even nondeterministic interprocess communications. Furthermore, we prove correctness of our method with respect to *barb-agreed simulation.*

1 Introduction

Partial evaluation is a program transformation scheme to improve efficiency by specializing a program with respect to its known part of inputs. Many partial evaluation techniques have been proposed for sequential languages [6, 14]. They have been applied to programs in many fields such as compilers for reflective languages [10], logic circuit simulation [14], and numerical computation [2], speeding up programs by orders of magnitude. In concurrent languages, partial evaluation may optimize code for data distribution or load balancing if it depends heavily on the known inputs, or partial evaluation may enhance low-level optimizations by expanding sequences of message sending explicitly in output code. In spite of such potential benefits, there are few proposals of partial evaluation methods for concurrent languages.

Our goal is to develop a partial evaluation method that (1) is powerful enough to handle multiple processes and their communications, which are not treated in conventional partial evaluation methods for sequential languages, (2) is general enough to easily apply to various constructs in high-level concurrent (especially object-oriented) languages, and (3) is clear enough to formalize and prove its correctness since partial evaluation is non-trivial global program transformation. For these purposes, we make the following two fundamental decisions.

First, rather than doing complicated and specific partial evaluation directly on high-level languages, we first translate a program in a high-level language into a simple language based on *process calculi* [12, 9] and then apply partial evaluation to the translated program. It is because process calculi have theoretical clarity, and yet significant expressive power enough to represent high-level concurrent (object-oriented) languages, so that they have been foundations of static analyses and optimization techniques for concurrent languages [7, 8]. It is therefore easy to formalize our partial evaluation method and discuss its correctness. In this paper, we chose a subset of HACL [9], one of such process calculi, as our target language.

The second decision is that we allow our partial evaluation to eliminate communications between processes. This approach can not only reduce high-cost communications themselves, but can also make it easy to propagate values sent by communications for further optimization of the subsequent code. Even when there are several *nondeterministic* choices in a communication, our method chooses one and discards the rest. In this sense, our approach *resolves nondeterminism.* Some previous work already takes similar approaches. For example, some existing optimizing compilers achieve high performance by generating code for statically fixed scheduling order [15]. In practice, results of most of concurrent applications do not depend on nondeterminism.

We formalize our partial evaluation method and prove its correctness. Because our approach resolves nondeterminism, a resultant program is not always equivalent to the original program with respect to usual process equivalence relations. We therefore introduce ***barb-agreed simulation***, which is a relaxed version of barbed bisimulation [11]. We show that our method is correct with respect to barb-agreed simulation. We believe that it sufficiently expresses correctness criteria for most program transformation schemes as well as partial evaluation. As far as the authors know, this is the first study that develops a partial evaluation method for a concurrent language with enough expressive power, and also proves correctness of the method.

According to our preliminary experiments using a prototype system for our partial evaluation scheme with an application of logic expression interpreter, our method has power to eliminate a considerable number of communications in practice and improve efficiency dramatically.

The rest of this paper is organized as follows. Section 2 introduces the syntax and semantics of our target language. Section 3 gives our formal partial evaluation method. In section 4, we introduce barb-agreed simulation and prove our partial evaluation correct. In section 5, we discuss our approach to elimination of communications. In section 6, we remark on related work. Finally section 7 concludes this paper and touches upon future work. Full formalization, full proofs, and experiments, which are omitted here because of the lack of space, are found in our accompanying technical report [5].

2 Syntax and Semantics of the Target Language

This section introduces our target language, a subset of HACL. From the full HACL [9], we excluded static polymorphic types, choices, functions, and first-class processes.

In HACL, computation is performed by concurrent processes communicating each other *asynchronously* via channels. `m(`v`)` is a process that sends a value v to a channel `m`. `m(x)=>`P is a process that receives a values v from a channel `m`,

Syntax

$$\begin{array}{rrcl}
\text{(processes)} & \mathcal{P} \ni P & ::= & P_1 \,|\, P_2 \mid \texttt{\$x.}P \mid e_1\texttt{(}e_2\texttt{)} \mid e\texttt{(x)=>}P \mid \texttt{T} \\
 & & & \mid \texttt{if } e \texttt{ then } P_1 \texttt{ else } P_2 \\
\text{(procedure defs.)} & \Gamma & ::= & \{\texttt{f}_1\texttt{(x)=}P_1, \ldots, \texttt{f}_n\texttt{(x)=}P_n\} \\
\text{(programs)} & \mathcal{G} \ni \Pi & ::= & \Gamma \triangleright P \\
\text{(arithmetic exprs.)} & \mathcal{E} \ni e & ::= & \texttt{x} \mid \mathit{const} \mid \mathit{op}(e) \mid (e_1, \ldots, e_n) \mid \texttt{\#}i(e) \\
\text{(values)} & v & ::= & \texttt{x} \mid \mathit{const} \mid (v_1, \ldots, v_n)
\end{array}$$

Structural congruence

(1) $P_1 \,|\, P_2 \cong P_2 \,|\, P_1$ (2) $(P_1 \,|\, P_2) \,|\, P_3 \cong P_1 \,|\, (P_2 \,|\, P_3)$

(3) $\texttt{\$x.}(P_1 \,|\, P_2) \cong P_1 \,|\, \texttt{\$x.}P_2$ if $\texttt{x} \notin Fv(P_1)$.

Reduction Rules

$$\frac{P \cong Q \quad \Gamma \triangleright P \longrightarrow \Gamma \triangleright P' \quad P' \cong Q'}{\Gamma \triangleright Q \longrightarrow \Gamma \triangleright Q'} \text{(SCONG)} \qquad \frac{\Gamma \triangleright P \longrightarrow \Gamma \triangleright P'}{\Gamma \triangleright P \,|\, Q \longrightarrow \Gamma \triangleright P' \,|\, Q} \text{(PAR)}$$

$$\frac{\Gamma \triangleright P \longrightarrow \Gamma \triangleright P'}{\Gamma \triangleright \texttt{\$x.}P \longrightarrow \Gamma \triangleright \texttt{\$x.}P'} \text{(NEW)} \qquad \frac{}{\Gamma \triangleright \texttt{m(}v\texttt{)} \,|\, \texttt{m(x)=>}P \longrightarrow \Gamma \triangleright P[v/\texttt{x}]} \text{(COM)}$$

$$\frac{}{\Gamma \cup \{\texttt{f(x)=}P\} \triangleright \texttt{f(}v\texttt{)} \longrightarrow \Gamma \cup \{\texttt{f(x)=}P\} \triangleright P[v/\texttt{x}]} \text{(APP)}$$

Fig. 1. Syntax and operational semantics (selected)

and executes $P[v/\texttt{x}]$. Semantically, a channel is a bag of values, rather than a (FIFO) queue, and each of them is consumed by a receiver. If there are multiple senders and receivers, it is nondeterministic which pair of sender and receiver will communicate. Processes are spawned by parallel composition: $P_1 \,|\, P_2$. $\texttt{\$x.}P$ creates a channel $\texttt{x}$ and executes P. Channels can be carried as first-class data.

We give the syntax and operational semantics in Figure 1. *op* is a primitive operator such as integer addition. Note that $e_1\texttt{(}e_2\texttt{)}$ can be either a sender process or a procedure (described below) call according to whether e_1 is evaluated to a channel or a procedure. $\texttt{T}$ denotes a run-time error. A *context* $C[\cdot]$ is a process with a single occurrence of a hole $[\cdot]$ in it. $C[P]$ denotes a process obtained by replacing $[\cdot]$ in $C[\cdot]$ with P.

The operational semantics is defined via three reduction relations: an (applicative-order) reduction relation $\longrightarrow_{\mathcal{E}}$ over arithmetic expressions $\mathcal{E}$; a structural congruence $\cong$ over processes $\mathcal{P}$; and a reduction relation $\longrightarrow$ over programs $\mathcal{G}$, i.e., pairs of a set of *procedure* (parameterized recursive process) definitions and a process. Structural congruence $\cong$ is the smallest reflexive and transitive congruence relation closed under the rules in Figure 1. We define it in order to identify "equal" processes in terms of the structure and simplify the reduction rules. In the figure, we present part of the inference rules for $\longrightarrow$. **SCong** rule allows structurally congruent processes to make the same reduction.

Example We give an example of simplified parallel "logic expression interpreter", which inputs a logic expression (`exp`) and the value for the variable `X` (`valofx`), and outputs the value for the expression. Arguments of each subexpression are evaluated in parallel. We describe below this example in our language, where a natural extension to continuation passing style (CPS) [1] is used. (ML-style case statement is a syntax sugar.) The procedure `logev` takes an extra argument `r`, which is a "reply" channel to which the procedure will pass a return value. On the other hand, each caller of the procedure first creates a new channel for reply, call the procedure with the new channel as an extra argument, and, in parallel to this, waits for a return value on the channel.

```
logev(exp,valofx,r) = case exp of
    TRUE or FALSE => r(exp)
  | X => r(valofx)
  | (AND,e1,e2) => $s1.$s2.( logev(e1,valofx,s1) | logev(e2,valofx,s2)
                            | s1(v1)=>s2(v2)=>r(v1 and v2))
```

3 Partial Evaluation Method

We represent each step of program transformation in partial evaluation by a reduction relation $\rightsquigarrow$ over programs. Its usage is as follows. Suppose we are given a program $\Pi_0 \equiv \Gamma \triangleright P_0$ and a context $C[\cdot] \equiv C_d[C_s[\cdot]]$ where $C_s[\cdot]$ and $C_d[\cdot]$ denote a statically known context and a dynamic (unknown) context, respectively. We first transform $\Gamma \triangleright C_s[P]$ in arbitrary number of steps by $\rightsquigarrow$: $\Gamma \triangleright C_s[P] \rightsquigarrow^* \Gamma \triangleright P_1$. We then execute $\Gamma \triangleright C_d[P_1]$ by $\rightarrow$.

In Figure 2, we present part of inference rules for $\rightsquigarrow_{\mathcal{E}}$ over $\mathcal{E}$ and $\rightsquigarrow$ over $\mathcal{G}$. Most of these rules are naturally derived from the operational semantics rules. In partial evaluation, any subexpression in an expression is allowed to reduce. For arithmetic expressions, we allow destructions of tuples containing even irreducible expressions (**PEArSel** rule), which is known as *partially static data structures* [6]. **PECom** rule eliminates communications between pairs of senders and receivers apparently in parallel; we say that P_1 and P_2 are *apparently in parallel* in Q if $Q \cong \$x_1.\ldots.\$x_n.(P_1 \,|\, P_2 \,|\, R)$ for some R, $x_1,\ldots,x_n$. **PEApp** rule inlines procedure calls. Our treatment of procedures makes it easy to allow specialization of procedures [5] (making specialized versions of recursive procedures w.r.t. particular arguments.)

We show some examples of reductions to illustrate our rules. Let Γ be the definition of the procedure `logev` shown in Section 2. $\Gamma \triangleright$ `$r.(logev(X,b,r)` $|R)$, where $R \equiv$ `r(y)=>s(y and TRUE)`, invokes `logev`, waits for a result on `r`, and replies a value to `s`. We have the reductions

$$\begin{array}{llr} \Gamma \triangleright \texttt{\$r.(logev(X,b,r)}\,|\,R) & \rightsquigarrow \Gamma \triangleright \texttt{\$r.(case X of \ldots}\ |R) & \textbf{(PEApp)} \\ & \rightsquigarrow \Gamma \triangleright \texttt{\$r.(r(b)}\,|\,R) & \textbf{(PEIf)} \\ & \rightsquigarrow \Gamma \triangleright \texttt{\$r.(s(b and TRUE))} & \textbf{(PECom)} \end{array}$$

where the right-most column shows main rules applied in each reduction step. We present below the inference of the last reduction by **PECom**.

$$\dfrac{\dfrac{}{\Gamma \triangleright \texttt{r(b)}\,|\,R \rightsquigarrow \Gamma \triangleright \texttt{s(b and TRUE)}}\ \textbf{(PECom)}}{\Gamma \triangleright \texttt{\$r.(r(b)}\,|\,R) \rightsquigarrow \Gamma \triangleright \texttt{\$r.(s(b and TRUE))}}\ \textbf{(PENew)}$$

$$\frac{e_i \rightsquigarrow_{\mathcal{E}} e_i'}{(\cdots, e_i, \cdots) \rightsquigarrow_{\mathcal{E}} (\cdots, e_i', \cdots)}\ (\textbf{PEAr1}) \qquad \frac{1 \le i \le n}{\texttt{\#}i((e_1, \cdots, e_n)) \rightsquigarrow_{\mathcal{E}} e_i}\ (\textbf{PEArSel})$$

$$\frac{\Gamma \triangleright P \rightsquigarrow \Gamma \triangleright P'}{\Gamma \triangleright P \,|\, Q \rightsquigarrow \Gamma \triangleright P' \,|\, Q}\ (\textbf{PEPar}) \qquad \frac{\Gamma \triangleright P \rightsquigarrow \Gamma \triangleright P'}{\Gamma \triangleright e\texttt{(x)=>}P \rightsquigarrow \Gamma \triangleright e\texttt{(x)=>}P'}\ (\textbf{PERecv})$$

$$\frac{}{\Gamma \triangleright \texttt{m(}e\texttt{)} \,|\, \texttt{m(x)=>}P \rightsquigarrow \Gamma \triangleright P[e/\texttt{x}]}\ (\textbf{PECom})$$

$$\frac{}{\Gamma \cup \{\texttt{f(x)=}P\} \triangleright \texttt{f(}e\texttt{)} \rightsquigarrow \Gamma \cup \{\texttt{f(x)=}P\} \triangleright P[e/\texttt{x}]}\ (\textbf{PEApp})$$

Fig. 2. The partial evaluation rules (selected)

Nondeterminism **PECom** rule can *resolve nondeterminism.* Specifically, when there are multiple pairs of sender and receiver apparently in parallel on a channel, the rule chooses a pair and eliminates the communication between them. For example, among two choices of communications in the following program Π, our rule chooses one and may transform the program as follows.

$$\begin{aligned} \Pi \ &\equiv\ \Gamma \triangleright \texttt{\$c.(c(a) | c(x)=>c(x+b1) | c(y)=>c(y+b2))} \\ &\rightsquigarrow\ \Gamma \triangleright \texttt{\$c.(c(a+b1) | c(y)=>c(y+b2))} \ \rightsquigarrow\ \Gamma \triangleright \texttt{\$c.(c((a+b1)+b2))} \end{aligned}$$

Nondeterminism in the original program is thus resolved by partial evaluation. As mentioned in the introduction, in our principle, any optimization scheme as well as partial evaluation can be allowed to resolve nondeterminism, for efficiency.

4 Correctness of Partial Evaluation

This section shows correctness of our partial evaluation method. For correctness criteria, we cannot use bisimulation equivalences because of our treatment of nondeterminism. On the other hand, the simulation relation [11] is too weak because it allows m(1) | n(2) to be replaced with m(1), for example. We therefore introduce *barb-agreed simulation.*

Barb-agreed simulation involves a binary relation $\mathcal{R}$ on programs. We require that for each $(\Pi, \Phi) \in \mathcal{R}$, (1) each possible one-step reduction from Φ corresponds to some possible multiple-step reduction from Π, and (2) Π and Φ have the same "actions observable from external processes." We require the latter condition because what *always* happen in Π should always happen in Φ. We allow to observe in Π sender processes via free channels, and the error, if any; we call them *barbs* of Π.

Definition 1 Barb. A program $\Pi \equiv \Gamma \triangleright P$ has a *barb* a, written $\Pi \Downarrow_a$, if (1) $P \cong \texttt{\$x}_1 . \ldots . \texttt{\$x}_n . (a(v) \,|\, Q)$ where $a \ne \texttt{x}_i$ and $a \notin \mathit{Dom}(\Gamma)$, for some Q; or (2) $a = \texttt{T}$ and $P \cong \texttt{T} \,|\, Q$, for some Q.

Definition 2 Barb-agreed Simulation. A relation $\mathcal{R}$ is a *barb-agreed simulation* if $(\Pi, \Phi) \in \mathcal{R}$ implies (1) $\Pi \Downarrow_a$ iff $\Phi \Downarrow_a$; and (2) if $\Phi \to \Phi'$, then $\Pi \to^* \Pi'$ and $(\Pi', \Phi') \in \mathcal{R}$, for some Π'. We write $\Pi \dot{>} \Phi$ if $(\Pi, \Phi) \in \mathcal{R}$, for some barb-agreed simulation $\mathcal{R}$.

Because the barb-agreed simulation itself is too weak a relation (for example, it allows `m(1)` to be replaced with `m(2)`), in order to obtain a reasonable relation, we should further require that the relation be closed under any context, just as barbed congruence is obtained from barbed bisimulation by closing it under any context. We call the resulting relation a *barb-agreed quasi-congruence.*

In addition, we have to take the following into consideration: (1) in some rules such as **PECom**, Π need not to have the same barbs of Φ at the starting point, but to reduce in some steps to get the barbs. (2) some rules such as **PECom** and **PEApp** assume that Π never cause errors at execution time. We then obtain the following relation.

Definition 3 Barb-agreed Quasi-congruence. $\Gamma \triangleright P \succ^c \Delta \triangleright Q$ if, for each context $C[\cdot]$, $\Gamma \triangleright C[P] \not\to^* \mathtt{T}$ implies $\Gamma \triangleright C[P] \to^* \Pi'$ and $\Pi' \dot{>} \Delta \triangleright C[Q]$ for some Π'.

We can finally prove soundness of partial evaluation by induction on the height of inferences of $\Pi \leadsto \Phi$, and the reflexivity and transitivity of $\succ^c$.

Theorem 4 Soundness of Partial Evaluation. *If $\Pi \leadsto^* \Phi$, then $\Pi \succ^c \Phi$.*

5 Discussions on Elimination of Communications

Power of a partial evaluation method mostly depends on how it can propagate constants to the whole program. Our approach propagates values to be sent by (even nondeterministic) communications by just eliminating them. It is analogous to conventional approaches that propagate argument values of function calls by inlining them. However, we have the following issues to discuss.

Which communications to eliminate? Although our **PECom** rule is sound as already shown, it is conservative in the sense that it eliminates a communication only if the sender and the receiver are apparently in parallel. It is far from trivial to judge whether the elimination is sound or not if they are not apparently in parallel. For example, in the process $P \equiv$ `(n(5) | m(x)=>n(y)=>f(x+y))`, we can show that it is not sound to eliminate the communication between the sender `n(5)` and the receiver `n(y)=>f(x+y)`, while it is sound if the channel `n` is bound as `$n.`P, intuitively because no external process can receive from `n`. Developing an analysis to detect such cases is one of our future research issues.

Possibility of loss of concurrency Eliminating communications has potential of decreasing concurrency. For example, consider the following transformation,

$$\Gamma \triangleright \mathtt{m1}(e_1) \ \mid\ \mathtt{m2}(e_2) \ \mid\ \mathtt{m1(x)=>m2(y)=>n(x+y)} \ \leadsto^* \Gamma \triangleright \mathtt{n}(e_1\mathtt{+}e_2)$$

At execution time, the expressions e_1 and e_2 will be computed concurrently in the original program, while they will be computed sequentially in the resultant program. It is a trade-off whether we should eliminate the communication and decrease concurrency; or leave it uneliminated and retain concurrency. A possible decision strategy may analyze the size of the expressions and whether they are serialized in the receivers. Another strategy may use annotations specifying location of each process, and leave remote communications uneliminated. Note that this issue exists even if we eliminate only deterministic communications.

6 Related Work

One of our motivations is to aggressively optimize high-cost communication and synchronization constructs in concurrent object-oriented languages. The Concert compiler [13] stands on a setting similar to ours and proposes several optimization techniques. However, their approach is rather ad-hoc and handles locks and communication constructs separately, while our framework can handle them uniformly.

In the reflective concurrent object-oriented language ABCL/R3 [10], partial evaluation is used for compiling away meta-level interpreter code. However, although their meta-level uses in principle concurrent objects, the target of their partial evaluation is restricted to functional part of the meta-level programs, and every inter-object communication is residualized as an I/O operation. It is one of our goals to develop a framework that can optimize such communications.

Some partial evaluation methods for concurrent logic languages have been proposed [4]. These methods, in contrast to ours, take approaches that preserve nondeterminism, though they have not clearly described it. It seems to us that such approaches are difficult to propagate values between processes effectively. Moreover, their correctness has not been proven yet to the best of our knowledge.

7 Conclusion and Future Work

In this paper, we have proposed a simple, general, and well-formalized partial evaluation method for a simple language based on process calculi. By taking an approach that eliminates even nondeterministic communications, we realized effective partial evaluation. We introduced barb-agreed simulation as correctness criteria for program transformation schemes that are allowed to resolve nondeterminism, and proved correctness of our partial evaluation method with respect to barb-agreed simulation.

Finally, to make our partial evaluation method more powerful, we are planning to utilize the following analyses: linear channel analysis [8], which analyzes channels used only "once", for elimination of more communications, and set-based analysis [3], which analyzes conservatively values to be sent, for specializing receivers.

Acknowledgements: We deeply appreciate Hidehiko Masuhara, Kenjiro Taura and Tatsurou Sekiguchi for helpful comments and advice.

References

1. A. W. Appel. *Compiling with Continuation.* Cambridge University Press, 1992.
2. R. Baier, R. Glük, and R. Zöchling. Partial evaluation of numerical programs in Fortran. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, 1994.
3. N. Heintze. Set-based analysis of ML programs. In *proceedings of the 1994 Conference on Lisp and Functional Programming*, pages 306–317, 1994.
4. H.Fujita, A.Okamura, and K.Furukawa. Partial evaluation of GHC programs based on the UR-set with constraints. In *proceedings of Logic Programming: Fifth International Conference and Symposium*, pages 924–941, 1988.
5. H. Hosoya, N. Kobayashi, and A. Yonezawa. Partial evaluation for concurrent languages and its correctness. Technical report of the Department of Information Science, the University of Tokyo, 1996. to appear.
6. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
7. N. Kobayashi, M. Nakade, and A. Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In *Second International Static Analysis Symposium (SAS'95)*, volume 983 of *Lecture Notes in Computer Science*, pages 225–242. Springer-Verlag, 1995.
8. N. Kobayashi, B. C. Pierce, and D. N. Turner. Linearity and the pi-calculus. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 358–371, 1996.
9. N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes in Computer Science*, pages 137–166. Springer-Verlag, 1995.
10. H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, pages 300–315, 1995.
11. R. Milner and D. Sangiorgi. Barbed bisimulation. In *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695, 1992.
12. B. C. Pierce and D. N. Turner. Concurrent objects in a process calculus. In *Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*, pages 187–215. Springer-Verlag, 1995.
13. J. Plevyak, X. Zhang, and A. A.Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 311–321, 1995.
14. E. Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, 1993. (Technical Reprt CSL-TR-93-563).
15. K. Taura, S. Matsuoka, and A. Yonezawa. *StackThreads*: An abstract machine for scheduling fine-grain threads on stock cpus. In *Proceedings of Workshop on Theory and Practice of Parallel Programming*, number 907 in Lecture Notes on Computer Science, pages 121–136. Springer Verlag, 1994.