Using Aspects for Supporting Procedural Modules in # Programming

Francisco Heron de Carvalho Junior[1] and Rafael Dueire Lins[2]

[1] Departamento de Computação, Universidade Federal do Ceará
Campus do Pici, Bloco 910, Fortaleza, Brazil
heron@lia.ufc.br
[2] Depart. de Eletrônica e Sistemas, Universidade Federal de Pernambuco
Av. Acadêmico Hélio Ramos s/n, Recife, Brazil
rdl@ufpe.br

Abstract. Parallel programming still demands for higher-level languages, models, and tools that do not incur in performance penalties. The # programming model aims to meet those claims in large-scale programs. This paper describes how the # programming model works with procedural languages by using techniques from AOP (Aspect Oriented Programming). Performance comparisons with MPI are presented.

1 Introduction

High performance computing (HPC) architectures of today may be split into three classes: *capability computing* (MPP's[1]), *cluster computing* [6] and *grid computing* [14] architectures. Deep memory and source hierarchies can be supported in all classes. Grids, for example, may have clusters and MPP's as processing nodes, which may be formed by multiprocessors. Individual processors may implement vector and super-scalar processing. The consolidation of distributed architectures for HPC have brought new challenges. Efficient parallel programming on these architectures is not a trivial task using the tools available today. Despite having to specify computations, like in sequential programming, programmers must partition the application functionality and/or data, according to the features of the target architecture, and implement process synchronization. There are no consensual models for programming parallel architectures.

The evolution of parallel programming technology may be divided into three phases. The first phase was marked by the use of low level architecture-specific message passing interfaces. The start of the second phase is marked by the creation of CRPC (Center for Research on Parallel Computation), in 1989. From that milestone on, research efforts started to be coordinated, culminating with the development of several *efficient* and *portable* tools, including libraries for message passing (MPI [17] and PVM [15]), parallel extensions of Fortran (HPF [12] and Fortran M [13]) and specific-purpose scientific computing libraries (PETSc [3], ScaLAPACK [5], and many others [11]). The third phase searches

[1] Massively parallel processors, the supercomputers.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 730–739, 2005.

for models and languages for programming distributed high performance architectures, reconciling requirements of *generality* (**G**), high level of *abstraction* (**A**), *portability* (**P**) and *efficiency* (**E**), allowing to apply advanced software engineering concepts into the development of HPC software. Despite the efforts promoted in the second phase, and also due to the expansion in scale of HPC applications caused by *cluster* and *grid* computing, reaching the aims of the latter phase is still one of the most important challenges in parallel computing [11, 18].

The # parallel programming model provides a structured way to work with explicit message passing programming. The # parallel programming environment supports the analysis of large scale parallel programs by using Petri nets [9], including "debugging" and simulation facilities, proof of formal properties, and performance evaluation. The idea behind the # environment is to offer a "glue" for integrating existing high performance computing programming technologies in a common **component-based** framework, where advanced software engineering techniques may be successfully applied. The current prototype implementation of the # model is Haskell$_{\#}$ [8]. Haskell$_{\#}$ is a coordination language for distributing functional computations in clusters. Computations are described in Haskell, a pure lazy functional language. Haskell was initially adopted because it provides a clean orthogonal interface between coordination and computation media through lazy streams, besides allowing the analysis of formal properties of programs at computation level.

This work presents an approach based on AOP (Aspect Oriented Programming) [16] for incorporating computations written in *procedural languages* into the # programming environment. Procedural languages may either be *imperative* (such as C and Fortran), or *object oriented* (such as C++, Java, and C#). They are widely used for high performance programming, as they provide good time and space performance for scientific computations. Thus, it is possible to think about multi-lingual implementations of the # programming environment. This feature is highly desirable in large scale programming for grids.

This paper comprises three more sections. Section 2 presents an overview of the # programming model. Section 3 shows how procedural modules were introduced to the # programming environment. Section 4 benchmarks the proposed approach. Conclusions and lines for further works are presented in Section 5.

2 The # Component Model

The # programming model moves parallel programming from a *process-based perspective* to an orthogonal *concern-oriented perspective*. From the process-based perspective, a parallel program is a collection of processes synchronizing by means of communication primitives. For improving practice of parallel programming, it had been tried to lift level of abstraction for dealing with these primitives, resulting in efficiency losses. *Concerns* are scattered along implementation of processes, since they are orthogonal to processes. In fact, a process may be viewed as a set of *slices*, each one describing the role of the process with

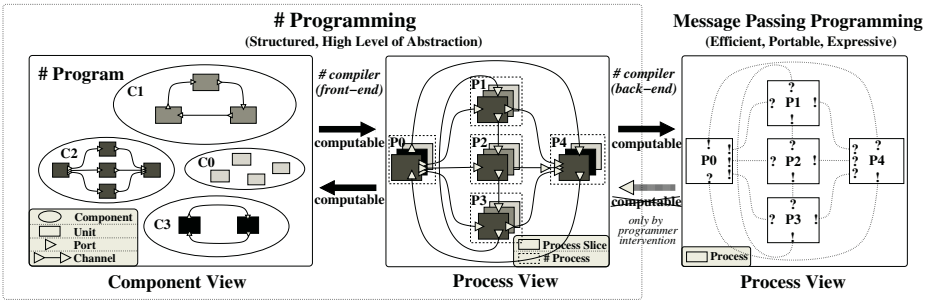

Fig. 1. Component Perspective versus Process Perspective

respect to a given concern. In this context, concerns are *decomposition criteria* for *slicing* processes [19]. Thus, they may be viewed as sets of related slices, probably from distinct processes. From the concern-oriented perspective of parallel programming, proposed by the # model, ***components*** are programming abstractions that address functional and non-functional concerns. We believe that a concern-oriented perspective of parallel programming fits contemporary advanced software engineering artifacts better than a process-based perspective.

In # programming, the slices that comprise a component are called *units*. They are connected in a communication topology, formed by one-direction, point-to-point, and typed channels. For that, a unit has a set of input and output ports, whose activation order is dictated by a protocol, specified using a formalism with expressiveness of labelled Petri nets. In # programming, concerns about parallelism and computations are separated in composed and simple components, respectively. Composed components comprise the ***coordination medium*** of # programs. They are specified in terms of units and channels, possibly by composition of existing components, by using some language that supports the coordination level abstractions of the # model. Today, there are a textual notation, called HCL (# configuration language), and a visual notation, called HVL (# visual language). Simple components are specified using Turing-computable languages, comprising the ***computation medium*** of # programs. They are the atoms of functionality in # programs. Simple components may be assigned to units of composed components in order to configure computations

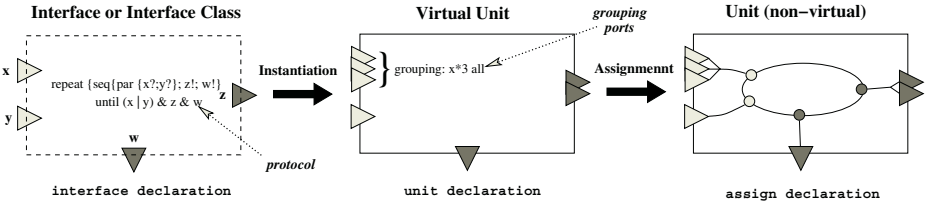

Fig. 2. Configuring a Unit

```
component CPipeLine <N> with

iterator i range [1,N]

interface ICPipe where
          ports: i* → o*
          protocol: repeat seq{o!; i?} until <o & i>

[/ unit pipe[i] where ports: ICPipe /]

connect pipe[i]→o to pipe[i+1]←i, buffered
```

```
component Torus <N> with

use Skeletons.Common.CPipeLine

iterator i, j range [1,N]

interface ITorus where
          ports: ICPipe @ n → s # ICPipe @ e → w
          protocol: repeat seq {par {s!; w!}; par {n?; e?}}
                    until <n & e & s & w>

[/ unit vpipe[i]; assign CPipeLine<N> to vpipe[i] /]
[/ unit hpipe[j]; assign CPipeLine<N> to hpipe[j] /]

[/ unify vpipe[i].pipe[j], hpile[j].pipe[i]
         to node[i][j] where ports: ITorus /]
```

```
component Farm<N> with

unit distributor where ports: () → job
unit worker where ports: job → result
                 protocol: seq {job?; result!}
unit collector where ports: result → ()

connect distributor.job to worker.job, synchronous
connect worker.result to collector.result, synchronous

replicate N: worker
```

```
component SqMatMult<N> where

iterator i, j range [1,N]

use Skeletons.Common.{Torus, Farm}
use MMShift, SPMD

interface ISqMatMult where
   ports: j → r # ITorus
   protocol: seq { j?; repeat seq {par {s!;e!};
                                   par {n?;w?}}
                       counter N;
                   r! }

unit mm_torus; assign Torus<N> to mm_torus
unit mm_farm; assign Farm<N> to mm_farm

[/ unify farm.worker[i + j × N], torus.node[i][j]
         to sqmm[i][j] where ports: ISqMatMult /]

unify farm.distributor, farm.collector, sqmm[0][0]
      to sqmm_root where
         ports: () → ab # c → () #
                ISqMatMult @ mm
         protocol: seq {ab!; do mm; c? }

unit spmd; assign SPMD<N × N> to spmd
supersede sqmm to spmd.peer

[/ assign MMShift to sqmm[i][j] /]
```

```
module MMShift(main) where

main :: Num t ⇒ t → t → [t] → [t] → ([t],[t],t)
main a b as_i bs_i = (as_o,bs_o,c)
     where
          c = matmult as_o bs_o
          (as_o, bs_o) = (a:as_i, b:bs_i)

matmult :: Num t ⇒ t → [t] → [t] → t
matmult [] [] = 0
matmult (a:as) (b:bs) = a*b + matmult as bs
```

Fig. 3. Configuration Code of Matrix Multiplication on a Torus

performed by slices. Skeletons [10] are supported by allowing units with no component assigned, called *virtual units*, giving support for high level of abstraction without loss in efficiency and portability. Nested composition of components is possible by allowing to assign composed components to units of other composed components. Besides to give support for non-functional concerns and skeletons, another important distinguishing feature of the # component model in relation to other component models [1, 4] is its ability to combine components by overlapping them. For that, it is possible to unify units from different composed components. Component models of today allow only nesting composition. Components are black-boxes addressing functional concerns. Whenever supported, non-functional concerns are introduced by means of orthogonal language extensions or by using tangling code cross-cutting component modules, like in sequential programming. However, cross-cutting concerns are not exceptions in parallel programming. The ability to overlap components makes possible to treat cross-cutting concerns as first-class citizens when parallelizing of applications.

Figure 3 presents a simple, yet illustrative, process topology of a composed component, named SqMatMult, that implements a parallel matrix multiplication strategy based on a systolic interaction pattern amongst processes organized in a torus. The code is written in HCL, the textual realization of the # coordination level abstraction. The component SqMatMult is composed by overlapping

skeletons TORUS and FARM. A $N \times N$ TORUS is defined by overlapping $N + N$ instances of CPIPELINE. The configuration code of components TORUS, FARM, CPIPELINE and SQMATMULT, in HCL, are also presented in Figure 3.

3 Procedural Modules as Simple Components

In Haskell# [8], simple components are *functional modules* written in the pure lazy functional language Haskell. Haskell provides the simplest technique for linking computation to coordination media without neither intermediate constructors nor extensions to the language Haskell. Functional modules neither make any reference to HCL constructors nor need to import libraries. They are standard Haskell modules, exporting the function *main*, whose arguments and elements of the returned tuple correspond to arguments and return points of the simple component. This is possible due to the Haskell support for *lazy lists*, which are associated to *streams* at coordination level [7]. However, using a language without lazy semantics, other approaches may be applied for keeping orthogonal the separation between coordination and computation media.

Procedural modules are simple components written in procedural languages, encompassing imperative and object oriented (OO) paradigms. They are implemented as *abstract data types* (imperative languages), or *objects* (OO languages). The routines, or methods, declared in procedural modules change the data structure state in the progress of computation. It is needed to define how procedural module routines (or methods) are invoked in response to events at coordination level and to define their arguments and return points. Techniques from Aspect Oriented Programming (AOP) [16] are used for the first purpose. For instance, a procedural module may be associated to aspect configurations, written in the # Aspect Language (HAL). In AOP, programmers may define *pointcut designators* that "identify particular *join points* by filtering out a subset of all the join points within the program flow". In the # terminology, the term program corresponds to the *protocol* of the *unit* for which the procedural module is *assigned*. *Join points* correspond to the actions in the protocol. Thus, pointcut designators stand for sub-sets of these actions. For defining them, *labels* and *pattern matching operators* may identify and filtering actions (joint points) in protocols. Labels extend HCL syntax for allowing to associate identifiers to actions. Pattern matching operators may be used for filtering sets of actions according to a given pattern. For example, the operator "_? | _!" stands for every communication action in a protocol, while the operator "**seq** {p!;_;_;...}" stands for any sequential action, encompassing at least three actions, that begins with the activation of output port p. A pointcut is enabled whenever one of its join points (actions) is reached when executing the protocol. Routines in the procedural module are associated to *pointcut designators*. They may execute before or after to enable the pointcut.

Figure 4(a) presents a C version for MMSHIFT. The HAL code presented in (b) defines three *pointcut* designators: INITIAL, COMPUTATION, and TRACING. For instance, the pointcut COMPUTATION is enabled whenever the actions

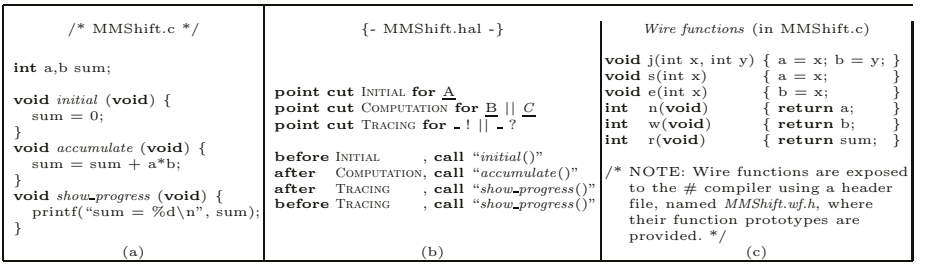

Fig. 4. C Version of the Functional Module MMShift

(join points) labelled by B and C are reached in the protocol of the unit *sqmm* of SqMatMult. A call to the subroutine *accumulate* is performed after Computation is enabled. The pointcut designator Initial has an analogous description. The pointcut designator Tracing is enabled in response to port activation. Before and after these events, the routine *show_progress* is invoked. No dynamic binding of routines to coordination events are needed, minimizing overheads. The # compiler is a *static weaver*, using the aspect configuration for generating code that calls specified routines at appropriate join points.

Arguments and return points of procedural modules are defined by means of *wire functions*. Essentially, wire functions compute the values to be transmitted through ports from the encapsulated state of the procedural module. Wire functions are declared in the procedural module and exposed by a header file listing their prototypes. Figure 4(a) exemplifies wire functions for unit *sqmm*.

4 Performance Evaluation Using NPB Kernels

A sub-set of the NPB kernels (*NAS Parallel Bechmarks*) [2] was implemented in # programming[2] by using AOP for linking imperative computations to # coordination medium: EP (Embarrassingly parallel), IS (Integer Sorting) and CG (Conjugate Gradient). They are used to compare the performance of # programs to their C/MPI (IS) and Fortran/MPI (EP and CG) counterparts. This experiment exemplifies how to design SPMD programs, a class where most of HPC programs fit, using the # approach. It also demonstrates how to translate MPI programs to the # model with minor performance penalties, despite gains in modularity and abstraction. The NPB kernels allow evaluating the use of collective communication skeletons for composing topologies and for automatically generating efficient code using lower level collective MPI primitives.

The composed components EP, IS, and CG address the functionality of the respective kernels, implementing the same strategies of parallelism adopted in the original versions. The differences lay on the separation of concerns between parallelism and computation in composed and simple components. The coor-

[2] Implementation codes of NPB kernels are available at `http://www.lia.ufc.br/ heron/npb_hash_code.html`.

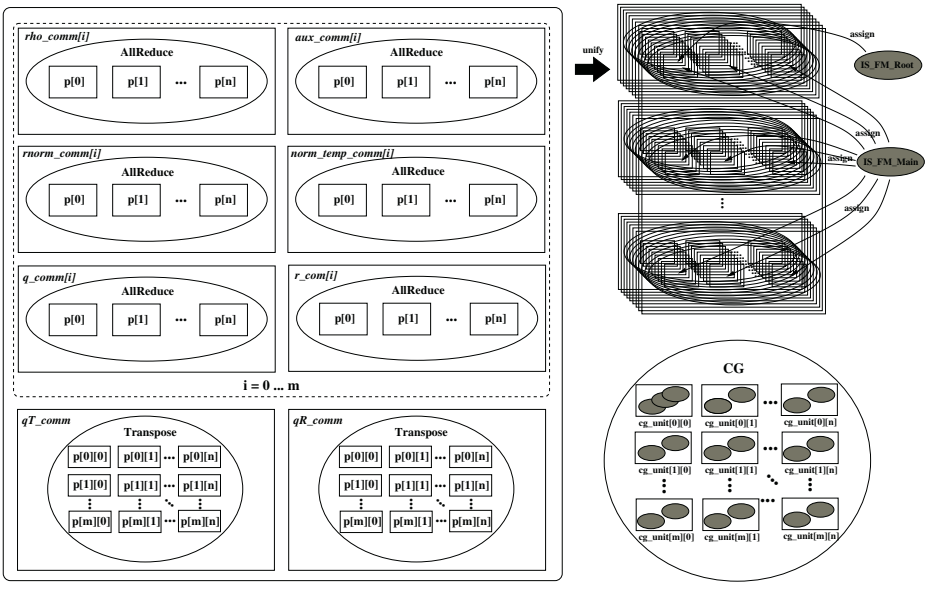

Fig. 5. The Topology of Component CG

dination medium specified is composed by overlapping composed components that implement collective communication skeletons (Figure 5). The resulting unit supersedes *peer* unit of a cluster for which component SPMD is assigned, informing the compiler about the "Single Program, Multiple Data" nature of the kernels. The procedural modules FM_EP, FM_IS, and FM_CG implement computations. Their routines are invoked according to events at coordination level, associated by means of aspect configurations (Section 3).

In the original versions of the NPB kernels, timing concern is implemented as calls to low-level timing routines intertwined with the code of computations. Using the # approach, a reusable component, called Timer, was designed for addressing the concern of *execution timing*. It was designed for synchronizing processes before timing begins, measuring duration of computation and communication/synchronization phases in a SPMD parallel program, and finally, providing timing summaries at the end of the execution. The component Timer is overlapped to the application components EP, IS and CG, yielding timed versions of them, called Timed_EP, Timed_IS and Timed_CG, by using unification. Using the same approach, it might be possible to design other reusable components to address cross-cutting concerns, such as *debugging*, placement and *load balancing* strategies, *security policies*, etc.

4.1 Performance Measures and Discussion

Figure 6 presents the performance figures for the NPB kernels EP, IS and CG. Standard problem sizes A, B, and C, defined in kernel documentation, is con-

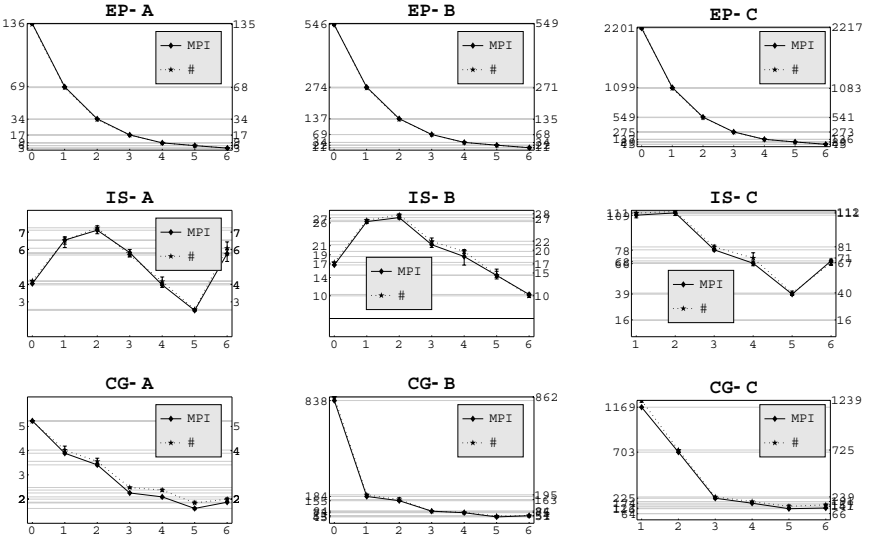

Fig. 6. Performance Figures for NPB (# vs. MPI)

sidered. For one process, IS and CG exhaust physical memory of cluster nodes ($> 1GB$). The architecture used was an Itautec cluster comprising 28 Intel Xeon nodes, each one with four processors, connected through a Gigabit Ethernet. It is installed at Computation Department of Federal University at Ceará, Brazil. MPICH 1.2.4 was used on top of TCP/IP. The data presented show no significative overheads for # versions in comparison to original ones, despite the gains in modularity advocated in the previous section. The minor differences are due to the partitioning of the monolithic original code in several routines scattered over distinct source files, affecting cache performance, causing larger number of function calls, and reducing some opportunities for compiler optimizations.

The presented empirical study may not be extended to all # programs and problem sizes. Indeed, it is not possible to define an exhaustive set of benchmarks that prove it, for any programming technology or architecture. However, it is possible to enumerate some reasons that may strengthen the reliability of the presented results to predict # performance in other situations: **(1)** virtually, any parallel programming technology on top of message-passing may be encapsulated as components in the # programming; **(2)** # programmers may implement the same parallelization strategies that they would implement by using the underlying parallel programming technology. In NPB kernels, # versions were produced by refactoring the original MPI versions, reusing all computation code without either modifications or reimplementation; **(3)** The # compiler does not add any kind of run-time support to the one provided by the underlying parallel programming technology; **(4)** The # compiler may allow the use of several parallel programming technologies in the same application, on top of the same component abstraction. In fact, this is a realistic assumption in current parallel programming practice, where non-modular combinations of MPI, openMP, and

possibly, grid enabling tools, such as Globus Toolkit are used together in the development of applications.

5 Conclusions and Lines for Further Work

This paper demonstrates how imperative and object oriented languages may be bound to the # programming environment by applying Aspect Oriented Programming concepts. Performance figures are presented comparing the performance of # versions of some kernels of NPB (NAS Parallel Benchmarks), where computations are implemented as *procedural modules*, to their MPI counterparts. The results show no significative performance overheads due to the use of # programming approach, despite gains in modularity and abstraction.

The work with # programming model is on progress. The main goal is to develop a parallel programming environment based on # model for integrating existing parallel programming technology, where the proof and analysis of formal properties, the simulation and the performance evaluation of programs may become a reality on top of Petri net-based tools and of NS (Network Simulator).

References

1. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Towards a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.
2. D. H. Bailey, T. Harris, W. Shapir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995. *http://www.nas.nasa.org/NAS/NPB*.
3. S. Balay, K. Buschelman, W. Gropp, D. Kaushik, M. Knepley, L. C. McInnes, B. Smith, and H. Zhang. PETSc Users Manual. Technical Report ANL-95/11 Revision 2.1.3, Argonne National Laboratory, Argonne, Illinois, 1996. http://www.mcs.anl.gov/petsc.
4. F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*. Springer-Verlag, 2003.
5. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics (SIAM), 1997.
6. R. Buyya (ed.). *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall, 1999.
7. F. H. Carvalho Junior, R. M. F. Lima, and R. D. Lins. Coordinating Functional Processes with Haskell$_{\#}$. In ACM Press, editor, *ACM Symposium on Applied Computing, Track on Coordination Languages, Models and Applications*, pages 393–400, March 2002.
8. F. H. Carvalho Junior and R. D. Lins. Haskell$_{\#}$: Parallel Programming Made Simple and Efficient. *Journal of Universal Computer Science*, 9(8):776–794, August 2003.

9. F. H. Carvalho Junior, R. D. Lins, and R. M. F. Lima. Translating Haskell$_{\#}$ Programs into Petri Nets. *Lecture Notes in Computer Science (VECPAR'2002)*, 2565:635–649, 2002.
10. M. Cole. *Algorithm Skeletons: Structured Management of Paralell Computation*. Pitman, 1989.
11. J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White. *Sourcebook of Parallel Computing*. Morgan Kauffman Publishers, 2003.
12. High Performance Fortran Forum. *High Performance Fortran, Language Specification, Version 2.0*, January 1997.
13. I. Foster and K. M. Chandy. Fortran M: A Language for Modular Parallel Programming. Technical Report MCS-P327-0992, Argonne National Laboratory, June 1992.
14. I. Foster and C. Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. M. Kauffman, 2004.
15. G.A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing. *MIT Press, Cambridge*, 1994.
16. G. Kiczales, J. Lamping, Menhdhekar A., Maeda C., C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Lecture Notes in Computer Science (Object-Oriented Programming 11th European Conference – ECOOP '97)*, pages 220–242. Springer-Verlag, November 1997.
17. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3-4):169–416, 1994.
18. A. Skjellum, P. Bangalore, J. Gray, and Bryant B. Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software. In *International Workshop on Software Engineering for High Performance Computing System Applications*, pages 59–63. ACM, May 2004. Edinburgh.
19. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.