A Data Parallel Java Client-Server Architecture for Data Field Computations over $\mathbb{Z}^n$

Jean-Louis Giavitto, Dominique De Vito, Jean-Paul Sansonnet

LRI u.r.a. 410 du CNRS, Bâtiment 490 – Université de Paris-Sud,
F-91405 Orsay Cedex, France.
email: {giavitto|devito}@lri.fr

Abstract. We describe `FieldBroker`, a software architecture, dedicated to data parallel computations on fields over $\mathbb{Z}^n$. Fields are a natural extension of the parallel array data structure. From the application point of view, field operations are processed by a field server, leading to a client/server architecture. Requests are translated successively in three languages corresponding to a tower of three virtual machines processing respectively mappings on $\mathbb{Z}^n$, sets of arrays and flat vectors in memory. The server is itself designed as a master/multithreaded-slaves program. The aim of `FieldBroker` is to mutually incorporate approaches found in distributed computing, functional programming and the data parallel paradigm. It provides a testbed for experiments with language constructs, evaluation mechanisms, on-the-fly optimizations, load-balancing strategies and data field implementations.

1 Introduction

1.1 Collections, Data Fields and Data Parallelism

The data parallel paradigm [19] relies on the concept of *collection* to offer an elegant and concise way to code many algorithms for data intensive computations. A collection is an aggregate of data *handled as a whole* [38]. A *data field* is a theoretically well founded abstract view of a collection as a function from a finite index set to a value domain [41, 24]. Higher order functions or intensional operations [33] on these mappings correspond to data parallel operations: point-wise applied operation (map), reduction (fold), parallel prefix (scan), rearranging operations (permutation), etc. One attractive advantage of the data field approach, in addition to its generality and abstraction, is that many ambiguities and semantical problems of "imperative" data parallelism can be avoided in the declarative framework of data fields [30, 25].

1.2 A Distributed Paradigm for Data Parallelism

Data parallelism was motivated to satisfy the increasing needs of computing power in scientific applications. As a consequence, the main target of data parallel languages has been supercomputers and the privileged linguistic framework was Fortran (cf. HPF [35]). Several factors urge to reconsider this traditional framework:

– Advances in network protocols and bandwidths have made practical the development of high performance applications whose processing is distributed over several supercomputers [39].

– The widening of parallel programming application domains (e.g. data mining, virtual reality applications, generalization of numerical simulations) urges to use cheaper computing resources, like NOWs and COWs (networks and clusters of workstations) [1].

– Development in parallel compilation and run-time environments have made possible the integration of data parallelism and control parallelism [18, 40, 4], e.g. to hide the communication latency with the multithreaded execution of independent computations.

– New algorithms exhibit more and more a dynamic behavior and perform on irregular data. Consequently, new applications depend more and more on the facilities provided by a run-time (dynamic management of time and space resources, localization, etc.).

– Challenging applications consist of multiple heterogeneous modules interacting with each other to solve an overall design problem. New software architectures are needed to support the development of such applications.

All these points require the development of portable, robust, high-performance, dynamically adaptable, architecture neutral applications on multiple platforms in heterogeneous, distributed networks.

Many of theses attributes can be cited as descriptive characteristics of distributed applications. So, it is not surprising that distributed computing concepts and tools, which precisely face this kind of problems, become an attractive framework for supporting data parallel applications. In this perspective, we propose `FieldBroker`, a client server architecture dedicated to data parallel computations on data field over $\mathbb{Z}^n$. Data field operations in an application are requests processed by the `FieldBroker` server.

`FieldBroker` has been developed to provide an underlying virtual machine to the $8_{1/2}$ language [11, 29] and to compute recursive definitions of group based fields [13]. However, `FieldBroker` aims also to investigate the viability of client server computing for data parallel numerical and scientific applications, and the extent to which this paradigm can integrate efficiently a functional approach of the data parallel programming model. This combination naturally leads to an environment for dynamic computation and collaborative computing. This environment provides and facilitates interaction and collaboration between users, processes and resources. It also provides a testbed for experiments with language constructs, evaluation mechanisms, on-the-fly optimizations, load-balancing strategies and data field implementations.

The subsequent sections describe `FieldBroker`. Section 2 presents the software architecture of the server. Section 3 describes the translation of a request

through a tower of three languages corresponding to a succession of three virtual machines. Section 4 reviews related works and the final section discusses the rationales of using Java in a preliminary implementation.

2 A Distributed Software Architecture for Scientific Computation

The software architecture of the data field server is illustrated by Fig. 1 right. Three layers are distinguished. They correspond to three virtual machines:

- The **server** handles requests on *functions over* $\mathbb{Z}^n$. It is responsible for parallelization and synchronization between requests from one client and between different clients.
- The **master** handles operations between sets of arrays. This layer is responsible for various high-level optimizations on data field expressions. It also decides the load balancing strategy and synchronizes the computations of the slaves.
- The **slaves** implement sequential computations over contiguous data in memory (vectors). They are driven by the master requests. Master requests are of two kinds: computations to perform on the slave's own data or communications (send data to other slaves; receives are implicit). Computations and communications are multithreaded in order to hide communication latency.

The communications between two levels of the architecture are specified by a language describing the data field representation and the data field operations. Three languages are used, going from the more abstract $\mathcal{L}_0$ (client view on a field) to $\mathcal{L}_1$ and to the more concrete $\mathcal{L}_2$ (in core memory view on a field). They are described in the next section. The server-master and the slave programs are implemented in Java. The rationale of this design decision is to support portability and dynamic extensibility (cf. section 5). The expected benefits of this software architecture are the following:

- **Accessibility and client independence:** requests for the data field computation are issued by a client through an API. However, because the slave is a Java program, Java applets can be easily used to communicate with the server. This means that an interactive access could be provided through a web client at no further cost. In this case, the server appears as a data field desk calculator.
- **Autonomous services:** the server lifetime is not linked to the client lifetime. Thus, implementing persistence, sharing and checkpointing will be much easier with this architecture than with a monolithic SPMD program.
- **Multi-client interactions:** this architecture enables applications composition by pipelining, data sharing, etc.

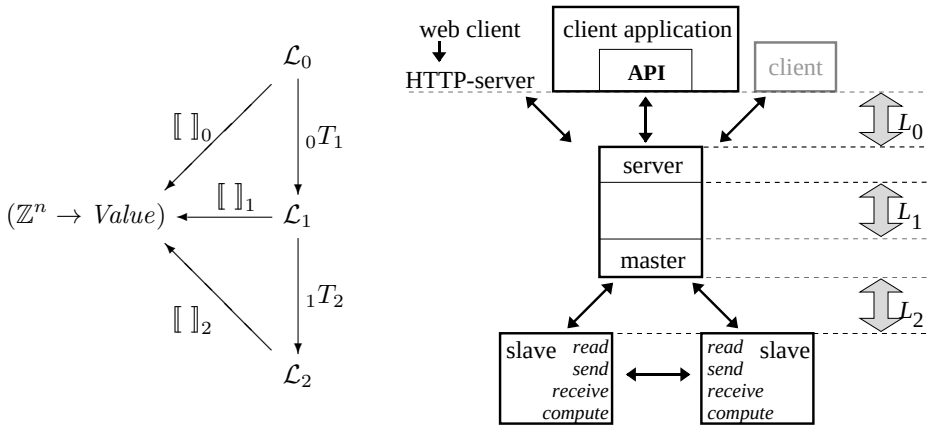

Fig. 1. ***Left:*** *Relationships between field algebras* $\mathcal{L}_0, \mathcal{L}_1$ *and* $\mathcal{L}_2$. ***Right:*** *A client/server-master/multithreaded-slaves architecture for the data parallel evaluation of data field requests.* The software architecture described on the right implements the field algebras sketched on the left. Functions ${}_iT_{i+1}$ are phases of the evaluation. The functions $[\![\ ]\!]_i$ are the semantic functions [31] that map an expression to the denoted element of $\mathbb{Z}^n \to Value$. They are defined such that the diagram commutes, that is $[\![e_i]\!]_i = [\![{}_iT_{i+1}(e_i)]\!]_{i+1}$ is true for $i \in \{0, 1\}$ and $e_i \in \mathcal{L}_i$. This property ensures the soundness of the evaluation process.

3 A Three Levels Language Tower

A client issues two kinds of requests to the server: data field expressions and commands. Commands are used by clients to modify the operational behavior of the server, e.g., garbage collection and data distribution constraints, etc. We focus in this paper only on the evaluation of data field expressions.

The evaluation of a $\mathcal{L}_0$ term begins with its optimization into an equivalent $\mathcal{L}_0$ term and its translation into a $\mathcal{L}_1$ term. The same treatment happens for a $\mathcal{L}_1$ term which is translated, after optimization, into a *set* of $\mathcal{L}_2$ terms. Finally, these terms are dispatched to the slaves to achieve the data parallel final processing. This process is illustrated in Fig. 1.

In the rest of this section, we sketch the $\mathcal{L}_0, \mathcal{L}_1$ and $\mathcal{L}_2$ algebras. Technical details, such as the formal definition of each function appearing in Fig. 1 (left), and the diagram commutations, can be found in [7].

3.1 $\mathcal{L}_0$: functions on $\mathbb{Z}^n$

We do not accept any function over $\mathbb{Z}^n$ as a data field. Intuitively we will preserve the operational property of the array data-structure: the access of an element is done in constant time. Translated in the data field context, this means that applying a data field to an index gives a value in constant time. Thus, two kinds of functions are allowed as data fields: functions over a finite set (because

they can be tabulated to achieve the previous property) and functions given as constant time evaluation rules.

Extensional and symbolic constants. The first kind of functions are the *extensional* constants of $\mathcal{L}_0$ and the second one, the *intensional* or symbolic ones. The idea is that extensional constants are implemented as (set of) arrays and that symbolic constants parameterize some operations on arrays.

We give an example to make it more concrete. The correct evaluation of expression $A+1$ must assign a data field to the (overloaded) constant 1: typically, 1 must denote a data field with same shape as A and $+$ is interpreted as a binary operator on data fields. Our approach is to assign to 1 a data field defined over all $\mathbb{Z}^n$ and to interpret $+$ as a strict operator. The advantage is that there is no need to overload 1 with several shapes [12, 20] anymore. Furthermore, there is no need to really build an array full of 1: the operation $(\_ + c_s)$ where c_s is a symbolic constant can be recognized as a specialized unary operator.

Functional operators. $\mathcal{L}_0$ operators are classified into functional and geometrical ones. An example of a functional operator is `map`: $\texttt{map}[op](F_1, ..., F_q)$ where op is a strict q-ary operator. Formally, we write

$$[\![\texttt{map}[op](F_1, ..., F_q)]\!]_0 = \lambda z \in \mathbb{Z}^n.\ op([\![F_1]\!]_0(z), ..., [\![F_q]\!]_0(z))$$

where a lambda expression is used to denote an element of $\mathbb{Z}^n \to \mathit{Value}$. However, we may omit the brackets $[\![\ ]\!]_i$ because they can be recovered from the context, and we write more liberally this semantic equation as:

$$\texttt{map}[op](F_1, ..., F_q)(z) = op(F_1(z), ..., F_q(z))$$

We adopt this simplification in the rest of this paper.

A second example, is the restriction:

$$\texttt{restrict}(F_1, F_2)(z) = \text{if } F_2(z) \text{ then } F_1(z) \text{ else } \star$$

which enables the selection of parts of data fields for later operations. The value $\star$ is a "soft bottom" element meaning "undefined value" (this value is distinguished from $\perp$ which means "unterminating computation", cf. [25, 15]).

We give a last example of a functional operator: the `merge` operator which recombines two data fields into one:

$$\texttt{merge}(F_1, F_2)(z) = \text{ if } F_1(z) \neq \star \text{ then } F_1(z) \text{ else } F_2(z)$$

`merge` implements the asymmetric union of data fields. It enables the representation of irregular data structures.

Geometric operations. A geometric operation g acts only on the index part of a data field, that is $g[F] = F \circ g$ where g is a function from $\mathbb{Z}^n$ to $\mathbb{Z}^m$. Examples of such functions allowed in $\mathcal{L}_0$ are: `transpose`, `shift` and `dilate` [7].

The optimization O_0 of $\mathcal{L}_0$ expressions is to convert any sequence of shift, transpose and dilate into a sequence of no more than five basic geometric operators. This simplification is very analog to the one performed in the *Infidel Virtual Machine* [37]. In our case, the computation of the canonical form is achieved as the normal form of a rewriting system [6], allowing the easy integration of additional optimizations as $\mathcal{L}_0$ rewriting rules.

Note that `restrict`, `merge`, `shift` (cf. below) and `map` are sufficient to implement an important class of numerical methods like red-black relaxations or explicit schemes for grid methods.

3.2 $\mathcal{L}_1$: expliciting iterations and lazy operations

The purpose of $\mathcal{L}_1$ is twofold. First, we will explicitly determine an "iteration domain" for each operator in $\mathcal{L}_0$, that is, to deduce the description of a region of $\mathbb{Z}^n$ where the data field is defined. Secondly, we will avoid to compute some operations by keeping them symbolic. This last goal is a generalization of the trick used to avoid the computation of a matrix transposition M^t: do not compute the transposition but remember to use $M(j,i)$ in place of $M^t(i,j)$ in the subsequent computations.

Avoiding `shift`, `restrict` and `merge`. Three kinds of $\mathcal{L}_0$ operations are subject to such a trick: `shift`, `restrict` and `merge`. To avoid the computation of shift operations, a constant of $\mathcal{L}_1$ includes the parameter of the translation. The purpose is similar to the one that motivates the `alignment` construct in HPF, but here, the alignment is assigned to each value (rather than to each variable), to enable a finer control over data movements. To avoid `restrict` operations, we adjoin a boolean data field that acts as a guard. And finally, to avoid merging, we represent the merge of a list of fields by a list. Thus, a $\mathcal{L}_1$ constant is described by:

$$\langle (s_1, b_1, s'_1, f_1) \; ; \; ... \; ; \; (s_p, b_p, s'_p, f_p) \rangle$$

where s_i, s'_i are translations, b_i are $\mathcal{L}_0$ boolean constants and f_i are $\mathcal{L}_0$ constant. The idea is that s_i is the translation associated to the boolean guard b_i while s'_i is the translation attached to the value field f_i, and the value of a point is the value defined by the first defined quadruple in the list. So, the meaning of such constants is defined inductively on the list structure: $\langle\,\rangle(z) = \star$ and $((s, b, s', f)\,; l)\,(z) =$ if $(b \circ s)(z)$ then $(f \circ s')(z)$ else $l(z)$, where l is a list of quadruples and ";" denotes the cons operation.

The translation ${}_0T_1$ of $\mathcal{L}_0$ terms in $\mathcal{L}_1$ is not detailed here, but we give some examples. Assuming that ${}_0T_1(F_i) = \langle s_i, b_i, s'_i, f_i \rangle$, then ${}_0T_1(\texttt{merge}(F_1, F_2)) = \langle (s_1, b_1, s'_1, f_1) \; ; \; (s_2, b_2, s'_2, f_2) \rangle$. For a translation t, ${}_0T_1(t[F_1]) = \langle s_1 \circ t, b_1, s'_1 \circ t, f_1 \rangle$. Finally, ${}_0T_1(\texttt{restrict}(F_1, F_2)) = \langle id, (b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2), s'_1, f_1 \rangle$ where id denotes the identity function. Note that this last expression is not a $\mathcal{L}_1$ constant but a $\mathcal{L}_1$ expression if $(b_1 \circ s_1) \wedge (b_2 \circ s_2) \wedge (f_2 \circ s'_2)$ cannot be simplified as a $\mathcal{L}_0$ constant (boolean expressions over symbolic constants are simplified in $\mathcal{L}_1$ expressions optimization).

Guards annotations and $\mathcal{L}_1$ optimizations. Other $\mathcal{L}_1$ expressions are made of $\mathcal{L}_0$ operators annotated by an explicit iteration domain. This iteration domain is simply a $\mathcal{L}_0$ boolean expression b which denotes an approximation of the definition domain. The idea is that this boolean guard acts as an explicit `restrict` on each expression. So, the definition of $[\![\ ]\!]_1$ fulfills the following property: $[\![e^b]\!]_1 \sqsubseteq [\![e^{b'}]\!]_1$ if $[\![b]\!]_0 \sqsubseteq [\![b']\!]_0$ where $\sqsubseteq$ is the Scott order [15] on $(\mathbb{Z}^n \to \mathit{Value})$.

The optimization of $\mathcal{L}_1$ expressions is to replace a guard by a more restricted expression without changing the general meaning. Formally, we will replace e^b by $e^{b'}$ such that $[\![e^b]\!]_1 = [\![e^{b'}]\!]_1$ but $[\![b']\!]_0 \sqsubseteq [\![b]\!]_0$. Here is an example on vectors. The construct $\mathtt{R}[x, y]$ is a symbolic constant that is true inside the (hyper) rectangle specified by two extreme points x and y and false elsewhere. Then, the $\mathcal{L}_0$ expression

$$\mathtt{map}[+](1, \mathtt{restrict}(2, \mathtt{R}[0, 10]))$$

is a data field over $\mathbb{Z}$ that adds point-wise an infinite vector of 1 and a finite vector of 2 of domain $[0, 10]$. This is translated into the $\mathcal{L}_1$ expression:

$$\mathtt{map}^{true}[+](\langle id, true, id, 1\rangle, \mathtt{restrict}^{true}(\langle id, true, id, 2\rangle, \langle id, \mathtt{R}[0, 10], id, true\rangle))$$

which in turn is optimized as

$$\mathtt{map}^{\mathtt{R}[0,10]}[+](\langle id, \mathtt{R}[0, 10], id, 1\rangle, \langle id, \mathtt{R}[0, 10], id, 2\rangle)$$

Note that after guard propagation, there is no more field with infinite extension in this example. However, a symbolic constant remains symbolic and is not translated into an extensional constant.

3.3 $\mathcal{L}_2$: working on flat vectors

A $\mathcal{L}_2$ constant is composed of a vector and a data descriptor. Each vector corresponds to the flattening of a multidimensional array and the associated data descriptor describes how the array elements are packed into the vector. Currently, the data descriptor of the vector v is a couple (s, b) where s and b are respectively called *stride* and *base* and are such that if a is the array associated with v and z a multidimensional index, $a[z] = v[s.z + b]$ where $.$ is the scalar product. $\mathcal{L}_2$ operations are vector operations corresponding to $\mathcal{L}_1$ operations and curried form of such operations where the provided arguments are symbolic constants (cf. example in § 3.1).

Each $\mathcal{L}_2$ constant is owned by a slave and slaves are mainly $\mathcal{L}_2$ interpreters distributed over a network. They are implemented in Java. The distribution strategy is a parameter of the system. For the moment, we have developed a very simple heuristic that uniformizes the amount of memory used by the slaves: a new vector is allocated to the slave that uses a minimal amount of memory. Obviously, more refined approaches have to be studied, e.g. to minimize execution time and data communication [28, 27].

A slave basically waits for incoming master requests and spawns new threads in order to evaluate computing requests when the corresponding arguments are

available. If some arguments of a received request are not available, the request and the missing arguments identifiers are registered in a soft scoreboard. This scoreboard is updated each time a result is produced or a data is received from other slaves. New threads are started for requests that are ready for evaluation. The threads of a slave have different priorities upon the corresponding task, e.g. a communication thread (send or receive) has a higher priority than a computing thread.

A slave operation is generally implemented by using a single loop over vectors, whatever the dimension of the original data fields (there is no need of dimension dependent nesting of loops because the flatness of the representation). The control part of a loop is scalar if the (used or computed) elements of the involved vectors are contiguous. Otherwise, it corresponds to the emulation of a multidimensional index. In this last case, the scalar vector indexes are computed incrementally by using this multidimensional index and the data descriptors associated with each vector.

4 Related Works

`FieldBroker` integrates concepts and technics that have been developed separately. Relationships between the definition of functions and data fields are investigated in [24]. A proposal for an implementation is described in [16] but focuses mainly on the management of the definition domain of data fields. Guard simplification in $\mathcal{L}_1$ is a special case of the extent analysis studied in [26].

One specific feature of `FieldBroker` is the use of heterogeneous representations, i.e. extensional and symbolic constants, to simplify field expressions. Further investigations are needed to formalize and fully understand this approach. Clearly, the algebraic framework is the right one to reason about the mixing of multiple representations. These remarks hold also for the tricks used in $\mathcal{L}_1$ to avoid evaluation.

Implementation of regions of $\mathbb{Z}^n$ in $\mathcal{L}_2$ are inspired from the projects [37, 22] which develop a language and a library dedicated to non-uniform block structured algorithms. However, we do not distinguish between several kinds of region specifications, focusing on a uniform handling.

The flattening of arrays into vectors in $\mathcal{L}_2$ is inspired from [37] and the implementation of vector operation using a single loop was inspired by A++/P++ [34] (itself using an algorithm described in [32]).

Client server architecture for HPC applications have been proposed recently: [23, 14]. A lot of research efforts are now dedicated to the use of the emerging web technology in an HPC framework [10, 3, 2].

5 Discussion: What Cost of Java?

We have implemented a first prototype of the client-server/master-multithreaded-slaves architecture using Java as the implementation language. The advantages

associated with the Java programming language (portability of processes, transparent memory management, anonymous and dynamic accesses to remote resources, ...) come at some costs: the natural communication model relies on RMI and not on message passing, the language is interpreted, and the memory management is dynamic.

It is too early to conclude about the viability of using Java in the context of numerical and scientific applications, mainly because the purpose of the first implementation is only to give us some insights into the benefits and drawbacks of the system design and functionalities. Because performance is not of primary concern, we have always prefered the straightforward implementations over the optimized ones, making unfair any performance evaluation (actually they are poor). The purpose of this section is then to sketch some evidences against the *a priori* disqualification of Java and to propose some of the necessary optimizations.

RMI versus MPI. The client-server model fits well with the Remote Method Invocation (RMI) or the Remote Procedure Call (RPC) interface. This process interaction model avoids many of the pitfalls of asynchronous message passing programming. Furthermore, message passing requires a number of complex tasks to be explicitly handled by the user (process identification, message preparation, transmission and reception, ordering, etc.). However the message passing paradigm is actually a *de facto* standard in data parallel programming due to the effectiveness, robustness and implementation portability of communication libraries such as PVM or MPI. The problem is then to evaluate whether the use of a RMI communication model is a viable alternative for scientific applications, or not.

Recent studies [8, 23] show that "the client server model does not degrade either programmability or performance for physical applications" [23]. The difference between the two communication modes on a set of simple scientific programs is less that 10 percent. Moreover, there is a lot of room for performance improvement through the utilisation of multiprotocol communication like in the Nexus library [9].

Bytecode interpretation versus compilation. Compared to VCODE, a virtual machine dedicated to vector operations, Java achieves one sixth to half the performance [17]. That is to say, w.r.t. to the facilities provided by an interpreted approach, the performance degradation induced by the Java virtual machine are not redhibitory. However, this still compares poorly against compiled code. This drawback has led to the development of *just-in-time* Java compilers [5] that are able to translate portions of the Java bytecode into executable machine-dependent code. The performance obtained by these JIT is much better and could be still improved [21].

In addition, a just-in-time compiler enables a kind of optimization that is generally out of reach from a request server. We illustrate this on an example involving loop fusion. Suppose for instance that two successive expression evaluations imply two successive loops with body e_1 and e_2. If the two loops have

the same iteration domain, and satisfy some additional constraints on e_1 and e_2, then a smart compiler is able to factorize the two loops into only one with body $e_1; e_2$. This optimization is out of reach from an interpreter because the available primitive operations do not include the sequence $e_1; e_2$. However, with just-in-time compiler, it is possible to synthesize on the fly the bytecode corresponding to $e_1; e_2$ to achieve loop fusion. This approach is a possible answer to the usual criticisms made on the server approach (no global optimization over requests) and is a direction for future researches.

Dynamic versus static memory management. Finally, Java dynamic management is sometimes argued against its use in the context of scientific applications. But this is unavoidable in the case of irregular, dynamic and data dependent applications. Moreover, commands can be used by an application to fine-tune the memory management.

Acknowledgments. The authors thank O. Michel, E. Valencia and the *Meta-Computing* LaMI-LRI working group for stimulating discussions. This research has been supported by the GDR de Programmation, Pôle Parallélisme, CNRS.

References

1. T. E. Anderson, D. E. Culler, D. A. Patterson, and the NOW Team. A case for networks of workstations: Now. *IEEE Micro*, Feb. 1995.
2. B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: data parallel extensions to Java. In *Proceedings of The ACM Workshop on Java for High-Performance Network Computing*, Stanford University, Palo Alto, California, February/March 1998.
3. H. Casanova and J. Dongarra. The use of Java in the NetSolve project. In *Proceedings of the 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematic*, Berlin, 1997.
4. K. M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications*, 8(2):80–98, 1994.
5. T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java just in time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–43, May/June 1997.
6. D. De Vito. Simplification of sequence expressions of shift, inject, project and transpose applications on domains or grids of $\mathbb{Z}^n$. Technical Report 1043, Laboratoire de Recherche en Informatique, May 1996. 23 pages.
7. D. De Vito. *Implémentation portable d'un langage déclaratif data-parallèle.* PhD thesis, Université de Paris-Sud, centre d'Orsay, Mars 1998. (in preparation).
8. R. Fatoohi. Performance evaluation of communication software systems for distributed computing. Technical Report NAS-96-006, NASA Ames Research Center, June 1996.
9. I. Foster, J. Geisler, C. Kesselman, and S. Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40:35–48, 1997.

10. I. Foster and S. Tuecke. Enabling technologies for web-based ubiquitous supercomputing. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Syracuse, New York, Aug. 1996. IEEE Computer Society Press.

11. J.-L. Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.

12. J.-L. Giavitto. Typing geometries of homogeneous collection. In *2nd Int. workshop on array manipulation, (ATABLE)*, Montréal, 1992.

13. J.-L. Giavitto, O. Michel, and J.-P. Sansonnet. Group based fields. In I. Takayasu, R. H. J. Halstead, and C. Queinnec, editors, *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068 of *Lecture Notes in Computer Sciences*, pages 209–215, Beaune (France), 2-4 October 1995. Springer-Verlag.

14. P. A. Gray and V. S. Sunderam. IceT: Distributed computing and java. In *ACM Workshop on Java for Science and Engineering Computation*. ACM, 1997.

15. C. A. Gunter and D. S. Scott. *Handbook of Theoretical Computer Science*, volume 2, chapter Semantic Domains, pages 633–674. Elsevier Science, 1990.

16. J. Halén, P. Hammarlund, and B. Lisper. An experimental implementation of a higly abstract model of data parallel programming. Technical Report TRITA-IT 9702, Royal Institute of Technology, Sweden, March 1997.

17. J. C. Hardwick, G. J. Narlikar, and J. Sipelstein. Interactive simulations on the web: Compiling NESL into Java. *Concurrency: Practice and Experience*, 1997.

18. P. J. Hatcher and M. J. Quinn. *Data-parallel programming on MIMD computers*. Scientific and engineering computation series. MIT Press, 1991.

19. W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, Dec. 1986.

20. C. Jay, D. Clarke, and J. Edwards. Exploiting shape in parallel programming. In *1996 IEEE Second International Conference on Algorithms and Architectures for Parallel Processing: Proceedings*, pages 295–302. IEEE, 1996.

21. R. Kadel. The importance of getting on the right road to a fast Java. *JavaWorld*, May 1997.

22. S. R. Kohn and S. B. Baden. A robust parallel programming model for dynamic non-uniform scientific computation. Technical Report TR-CS-94-354, U. of California at San-Diego, Mar. 1994.

23. A. T. Krantz and V. S. Sunderam. Client server computing on message passing systems: Experiences with PVM-RPC. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 110–117, Passau, Germany, August 1997. Springer-Verlag.

24. B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*. ACM, ACM Press, June 1993.

25. B. Lisper. Data parallelism and functional programming. In *Proc. ParaDigme Spring School on Data Parallelism*. Springer-Verlag, Mar. 1996. Les Ménuires, France.
26. B. Lisper and J.-F. Collard. Extent analysis of data fields. Technical Report TRITA-IT R 94:03, Royal Institute of Technology, Sweden, January 1994.
27. A. Mahiout. Integrating the automatic mapping and scheduling for data-parallel dataflow applications on MIMD parallel architectures. In *Parallel Computing: Trends and Applications*, 19-22 September, Gent, Belgium, 1995. Elsevier. poster session.
28. A. Mahiout, J.-L. Giavitto, and J.-P. Sansonnet. Distribution and scheduling data-parallel dataflow programs on massively parallel architectures. In *SMS-TPE'94: Software for Multiprocessors and Supercomputers*, Moscow, September, 1994. Office of Naval Research USA & Russian Basic Research Foundation.
29. O. Michel. Introducing dynamicity in the data-parallel language $8^1/_2$. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 678–686. Springer-Verlag, Aug. 1996.
30. O. Michel, D. De Vito, and J.-P. Sansonnet. $8^1/_2$: data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II:Proc. of the 9th Int. Symp. on Lucid and Intensional Programming*. World Scientific, May 1996.
31. P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.
32. I. Oliver. *Programming Classics : Implementing the World's Best Algorithms*. Prentice Hall, Dec. 1994.
33. M. A. Orgun and E. A. Ashcroft, editors. *Intensional Programming I*, Macquarie University, Sydney Australia, May 1995. World Scientific.
34. R. Parsons and D. Quinlan. A++/P++ array classes for architecture independent finite difference computations. In *Conference on Object-Oriented Numerics*, 1994.
35. Rice University, Houston, Texas. *High Performance Fortran Language Specification*, 1.1 edition, May 93.
36. M. Sato, H. Nakada, S. Sekiguchi, S. Matsuoka, U. Nagashima, and H. Takagi. Ninf: A network based information library for a global world-wide computing infrastracture. In *HPCN'97 Proceedings*, volume 1225 of *Lecture Notes in Computer Sciences*, pages 491–502, Vienna, Austria, Apr. 1997.
37. L. Semenzato. The infidel virtual machine. Technical Report UCB/CSD 93-761, Computer Science Division, University of California, Berkeley, 25 July 1993.
38. J. M. Sipelstein and G. Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, Apr. 1991.
39. L. Smarr and C. E. Catlett. Metacomputing. *Communications of ACM*, 35(6):45–52, June 1992.
40. G. Steele. Making asynchronous parallelism safe for the world. In *Seventeenth annual Symposium on Principles of programming languages*, pages 218–231, San Francisco, Jan. 1990. ACM, ACM Press.
41. J. A. Yang and Y.-i. Choo. Data fields as parallel programs. In *Proceedings of the Second International Workshop on Array Structure*, Montreal, Canada, June/July 1992.

This article was processed using the LaTeX macro package with LLNCS style