A Higher-Level Environment for Parallel Programming

Shakil Ahmed and David Gelernter

YALEU/DCS/RR-877
November 1991

A Higher-Level Environment for Parallel Programming*

Shakil Ahmed and David Gelernter
Department of Computer Science
Yale University
P.O. Box 2158, Yale Station
New Haven, CT 06520-2158
U.S.A.
Phone: (203) 432-1227, Fax: (203) 432-0593
Internet: ahmed-shakil@cs.yale.edu
Bitnet: ahmed-shakil@yalecs.bitnet

November 12, 1991

Keywords: Parallel programming paradigms, methodologies, parallel language extensions, programming environments for parallel languages

Abstract

We present the Linda Program Builder - a higher-level programming environment that supports the design and development of parallel software. It isolates much of the administrative effort in constructing parallel programs, and maintains a program-describing database. This database feeds information to the compiler for optimization, to a visualizer for enhanced program visualization, and to other tools in the environment. The LPB is a window-oriented, menu-based, user-friendly system which provides coordination frameworks for program construction. Most importantly, it represents an alternative approach to high-level programming languages.

*This research is supported by National Science Foundation grant CCR-8657615, by the Air Force Office of Scientific Research under grant number AFOSR-91-0098, and by Scientific Computing Associates, New Haven.

1 Introduction

The Linda Program Builder (LPB) [ACG91] is a higher-level programming environment that aids in the design and development of parallel software. The LPB is an Epoch-based[1], menu-driven, putatively user-friendly system that supports incremental development of explicitly parallel C-Linda programs.

This paper will discuss three major topics: CASE aspects of the LPB, the LPB's relationship to the compiler and visualizer, and (potentially most important) the LPB approach as an alternative to very high level programming languages.

Linda itself has been extensively discussed in previous work, so we will not describe it here. [CG89] is a representative paper.

2 CASE aspects of the LPB

The LPB is built on top of Epoch running under X-windows. The LPB environment is menu-driven, but allows the full flexibility of Epoch (Emacs) in editing all files. Several windows are open at all times. Some offer command menus. Another may offer a selection of tuples known to the LPB, while an additional window displays known information on the current tuple selected. Yet another menu lists all open Linda files and allows point-and-click switching between files. An experienced programmer may choose to bypass many of these point-and-click facilities. There have been many template-editor predecessors to the LPB, most notably the Cornell Program Synthesizer[RT89a], but on the whole, they impose rigid frameworks which the programmer is forced to follow. This insistence on following an imposed template offers guaranteed syntactic correctness, but it limits the flexibility that a creative programmer needs. The LPB offers similar features, but doesn't impose them.

The most important features of the LPB are its support of *templates* and *high-level operations,* and its construction of a *program database.* The LPB's goal is to capture organizing strategies for parallel programs and basic coordination frameworks. The basis for the methodology is presented in [CG90]. It focuses on three main paradigms known as "specialist", "result", and "agenda" parallelism.

[1] Epoch is a multi-window version of emacs developed by S. Kaplan of the University of Illinois, Urbana

A template is a program skeleton for a particular paradigm that serves as a guideline to program construction. The template expands into more detailed code. A higher level operation allows tuple manipulation at a conceptually higher level than basic tuple operations. These operations are inserted into the code as buttons which can be expanded for a view of the implementation, but can also be abstracted back to a higher level. The program database maintains tuple, function, and higher-level operation information and is the backbone of the system.

2.1 Templates

Templates are provided and the programmer can choose to follow them all the way through, but he is free to leave this framework whenever he desires and to return when necessary. Many templates have already been implemented, but others may easily be added.

The existing templates follow an incremental approach using buttons to expand various segments of code. Consider the master-worker template as an example. The master-worker paradigm is discussed in [ACG91], and we omit details here. When a master-worker template is chosen from the main menu, a template skeleton is presented. Figure 1 shows the initial buttons, for example, the "Master routine" and "Worker routine" buttons. Clicking a button on a template causes it to expand into more code, which may contain more buttons. Various stages may require further input, for which the user is prompted either by menus or input windows. The program grows as more buttons are expanded and more information is obtained from the user. At certain stages, there may be buttons which cannot be expanded further until other buttons are expanded first — buttons may be dependent on information that can only be acquired through the expansion of other buttons.

The user is always free to enter code or comments anywhere within the partially constructed program. The intention is to restrict the programmer's activity to parts of the program that don't deal explicitly with parallelism.

A master-worker program typically involves task tuples and result tuples. All tuple operations and the associated variable declarations are automatically generated according to the particular model chosen. An intermediate stage in program generation is shown in figure 2.

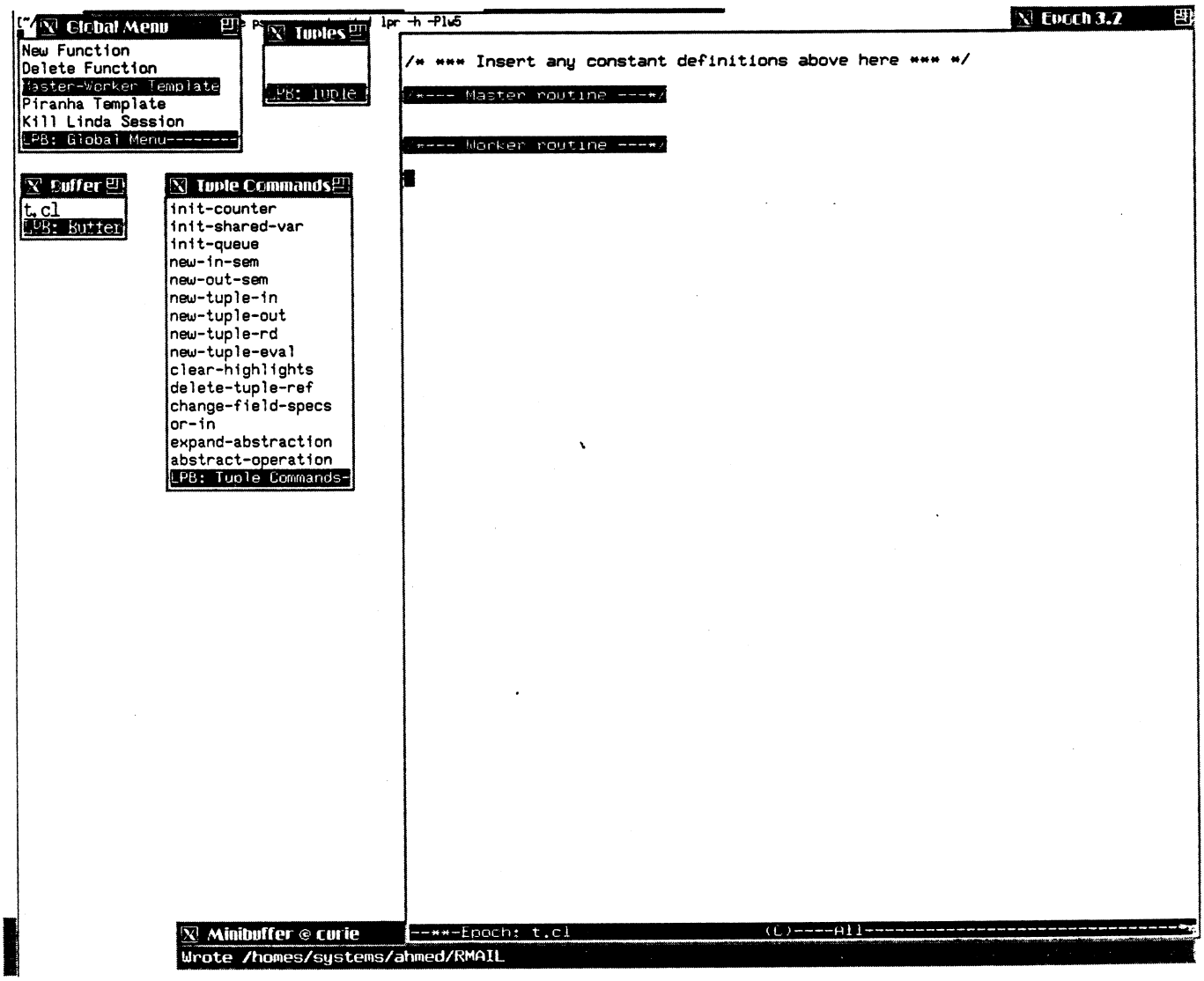

Figure 1: Initial master-worker template

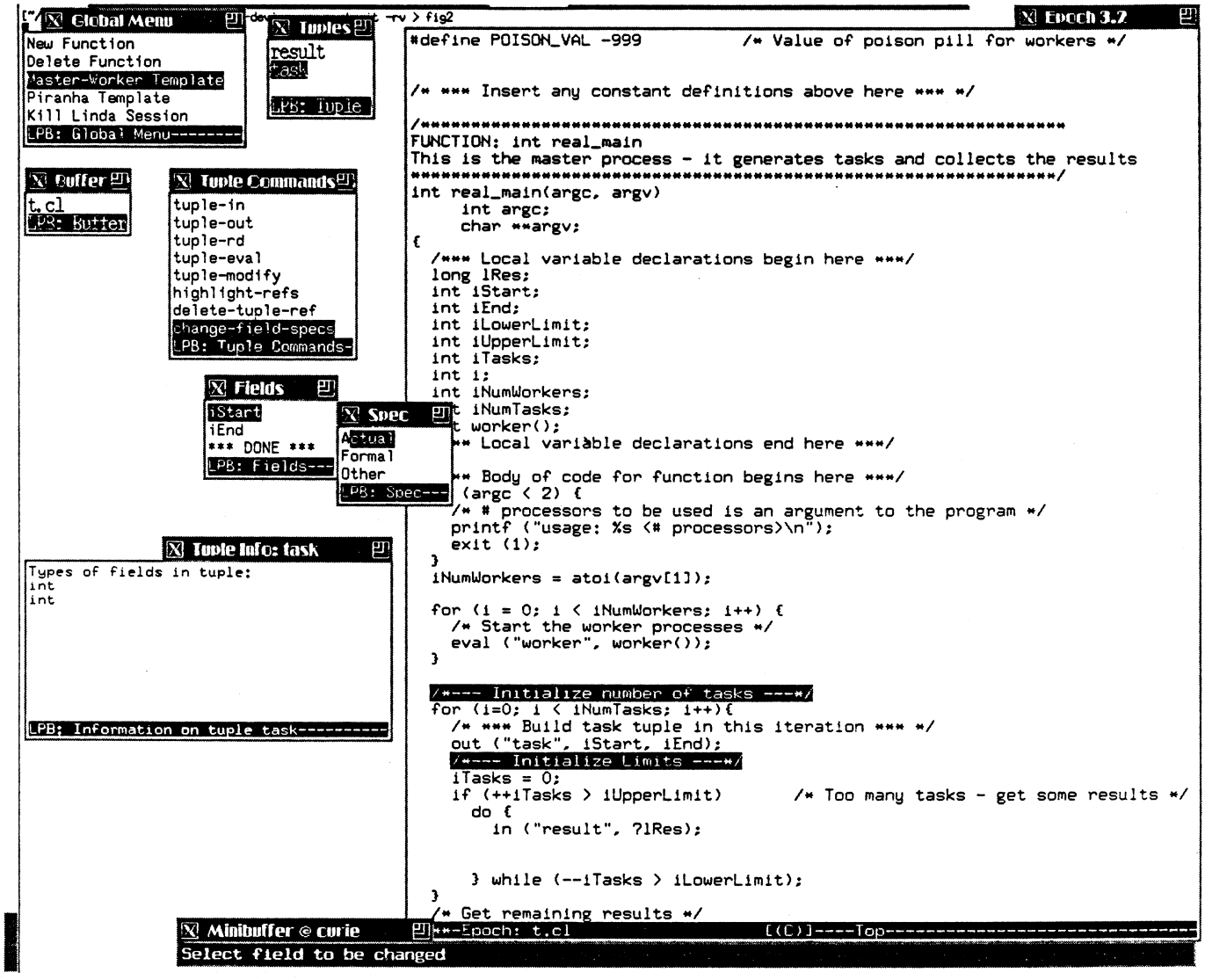

Figure 2: Intermediate stage in master-worker program generation

2.2 Distributed data structures

Linda programs generally make use of distributed data structures such as distributed arrays, task bags, shared variables and so on. Many of these can be anticipated by the coordination-framework templates discussed above. Structures such as task bags, watermarked bags or distributed queues, to name a few, are often incorporated into the choices presented during the construction of a program through a template. But when the case arises where a programmer needs to specify a particular data structure outside of a specific template, the LPB provides the necessary support.

For example, the LPB has menu options to support creating and manipulating shared variables and counters in tuple space. Counting semaphores are supported in a similar manner. The labels of these variables appear in the tuple menu. Picking a tuple of one of these types will cause the commands menu to change accordingly, and the information window to display known information on the tuple.

Distributed queues of various kinds are often required in parallel programs. They may have multiple sources, sinks, or both. The synchronization and handshaking necessary for coordination among the various sources and sinks can be achieved through distributed head and tail pointers in tuple space. The LPB provides a complete set of menu functions to create and manipulate queues. Upon selection of a create-queue command, a popup menu will offer choices on the various models available. Once a model has been selected, all the tuples necessary for maintenance of the queue are automatically generated and initialized. A user is now free to select menu commands to add to or remove from the queue as desired. All tuple operations, declarations, and additional code are automatically inserted at the appropriate places.

2.3 Higher level program constructs and abstractions

Certain operations ought to be supported, but not at the level of a full program template. The LPB supports high-level operations which can be expanded and then abstracted back again (buttons, on the other hand, can only be expanded). Thus we can have a higher level construct which is presented to the user as an abstraction. If the user wants to see the implementation of the higher-level operation, he can choose to expand it. It can then be abstracted back to the higher level representation which is more concise and easier to understand.

The abstraction feature is a powerful tool which furnishes the user with a novel approach to viewing and constructing programs. For top-down program construction, it presents a high-level view of program structure which can not only be expanded downwards but abstracted back up again to a conceptually more appealing higher-level format. This allows the programmer to concentrate on hierarchical program construction at a high level, and to deal with "blocks" of code represented by abstractions which together form a larger program.

This is somewhat similar to the Cedar [SZBH86] approach in its Tioga structured text-editor. Tioga treats documents in a tree-structured manner where each node is a paragraph or a statement. This hierarchical node structure allows concealing detail to present a conceptually higher-level view, much as in the LPB.

We give an example in Section 4.

2.4 The program database

The LPB offers features to manipulate basic tuple functions. Variable declarations and code insertion are automated, as are cross-module propagation of updates to tuple references when a tuple structure is modified. This is achieved through use of a program-describing database that the LPB maintains at all times.

Every tuple, function, abstraction, higher level function, or any other crucial components of the program are entered into the database as they are used. The database keeps information on a tuple's label, on the variables used in its fields, the status of each of the fields, information on the nature of the tuple and its use, and a record of all the places where references to the tuple exist.

The archive is global across a user's LPB sessions. It is saved together with the program files, and automatically loaded when a file is read in.

This database is the backbone of the LPB, maintaining all the information necessary to perform higher level operations and provide user support, eliminating the need for much memory-work and reducing keystrokes. The stored data supports automation that can prevent errors. It is also the basis of data that is passed on to the compiler and visualizer. The LPB's knowledge of program structure is characteristic of an expert-database approach to intelligent program development, and expert database heuristics of the sort described in [FG91] will be added eventually.

3 Interfacing to other tools

The topics under this heading are still work in progress, not yet fully implemented. We briefly describe our plans nonetheless, because of their significance in motivating the project as a whole.

3.1 The compiler

The Linda pre-compiler parses and analyzes operations on tuples. Given certain sequences of operations, the pre-compiler may draw certain conclusions about the intended effects, but (like any compiler) it can rarely infer the user's *intent* in specifying particular sequences of operations. The LPB has superior knowledge in this regard. Since the program is being constructed through templates or other higher level conceptual frameworks, the LPB "knows" *why* the various operations are being used. For example, the LPB can distinguish between "task" tuples which are used to describe tasks in a master-worker program, and "result" tuples which hold the result elements that are passed back to the master. Given a particular distributed data structure, the LPB knows which tuple operations need to be used to create and manipulate it. The LPB has a much deeper understanding of data structures than an analyzer can develop at compile time.

This knowledge can be valuable to a compiler. Given an understanding of what a series of Linda operations is intended to achieve, the compiler might generate a semantically equivalent series of operations that are more efficient. A series of Linda operations is typically carried out as discrete operations over tuple space. In some cases it will be possible to fuse these operations together and perform a smaller number of discrete operations over tuple space, saving overhead. Queues are a good example. To remove an element from the head of a multi-sink queue, three Linda operations are required (two for atomic update of the pointer tuple, and the third to remove an element from a queue). The LPB supports queue operations, and it is therefore aware of intended effects such as "remove an element from the head of a multi-sink queue." It can pass on this information to the compiler and indicate which three operations are intended to achieve that effect. The compiler, in turn, can make use of this information and fuse these three operations together for a gain in efficiency.

In general, distributed data structure manipulations are good candidates for optimization through fusion. Updating any kind of shared variable or counter typically

requires an **in** operation to remove the tuple from tuple space, followed by an **out** with the updated value. These two operations may be separated by other code, but the LPB is aware of these operations and their purpose, and can advise the compiler how to fuse and optimize the operations. In a related case, a shared variable can typically be updated directly in tuple space without actually being removed and then returned, as the user's code must specify (tuples being immutable objects in logical terms). This is especially significant in the context of distributed memory machines, where tuple space manipulations are somewhat costlier than on shared memory machines.

3.2 Program visualization

Our Tuplescope visualizer [BC90] is a graphical monitoring tool that presents a dynamic image of an executing Linda program. Tuples are represented on the screen and their movement to and from tuple space is displayed as the program executes. The LPB can pass on enough information to Tuplescope to allow a better organization of the display. Where conventional Tuplescope shows mere processes, our enhanced Tuplescope may differentiate master and worker processes. Normally, a counter tuple looks like every other tuple, but with LPB-generated information, Tuplescope can understand counters: a counter can have its own window with a displayed value that changes as the counter is updated.

As it stands, Tuplescope has no concept of a queue; it regards the various tuples involved as just that, bare tuples. Once it knows that a queue is being manipulated, a graphical representation of a queue can be displayed, with head and tail pointers marked appropriately, and the actual queue clearly visible. This level of enhanced visualization allows a user to picture his program at a conceptual level closer to his line of thinking.

The various templates the LPB offers could each have unique representations in Tuplescope. Piranha programs[BCG+91], for example, all follow a particular pattern which is similar to general master-worker programs. The LPB is aware of what constitutes a Piranha program and which process are "feeder" and "piranha" processes.

3.3 Performance monitor

A performance monitor will make use of information from the LPB as well. Depending on which template was used, a performance model is selected and displayed

with the appropriate graphical displays. In conjunction with the LPB, the compiler, and Tuplescope, this will complete the environment, being useful in later stages of program development when performance tuning comes into play.

4 The LPB as an alternative to higher-level languages

The methodologies and higher level operations supported by the LPB address an issue that is wider than CASE for parallelism. LPB-like program builders present an attractive alternative to very-high-level languages.

Since the late sixties, many researchers have held that programming languages are too "low level." High level operations are desirable for ease of programming, for maintainability, and for quick software development.

The conventional approaches are to add very-high-level features to existing languages, or to implement completely new languages. The LPB presents a clear alternative. Instead of a very high-level *language*, we can layer a very high-level *program builder* over a general-purpose language. The user sees similar advantages in terms of strong support for high-level models and constructs. But he doesn't pay the traditional price in terms of a language that is highly complex (e.g. Ada [Bar80] or Common Lisp [Jr82] [Jr84]) or restrictive, inflexible or narrow in its range of applicability (terms in which logic and functional languages, for example, are often described). The program builder may be very high-level, special-purpose, idiosyncratic — but it may also be changed easily, customized or evolved readily — and simply bypassed when the very-high-level constructs it supports aren't the right ones.

An examination of two different examples in the context of the LPB will illustrate these points. Consider distributed queues. When the user wants to create a queue of tuples, he picks the menu option for queue creation. This causes a new popup menu to appear (Figure 3) which offers a choice of different queue models. Once a model has been identified, the user is prompted for some initialization data and the appropriate code is automatically created and inserted. Any further queue manipulation can now be accomplished through the options in the command menu that appears when the queue label is selected. Adding to the tail and removing from the head of the queue are both operations that are supported at this level — the appropriate code is generated automatically (Figure 4). As we have mentioned, there are two significant consequences. First, the task of removing the element from

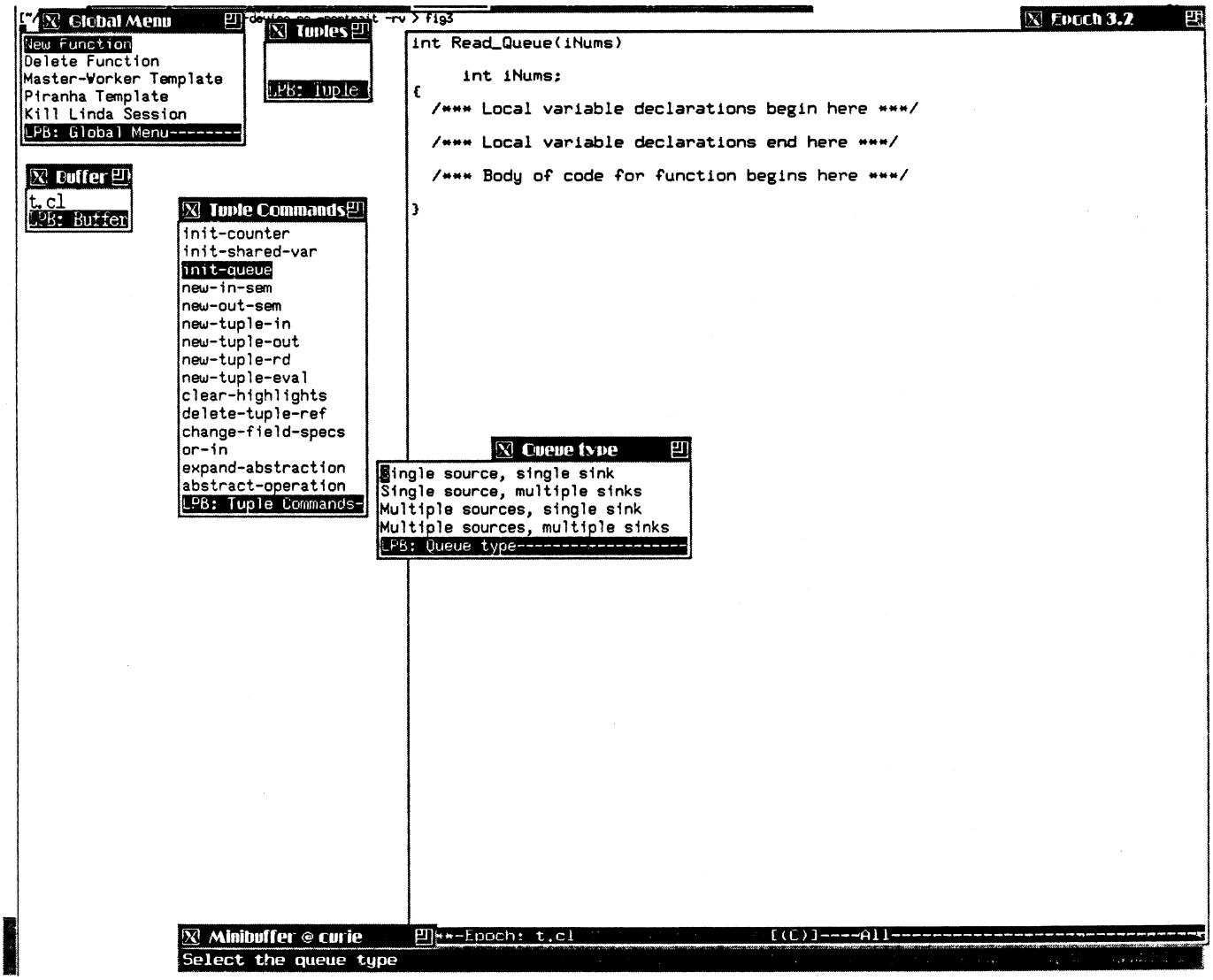

Figure 3: The different queue model selections

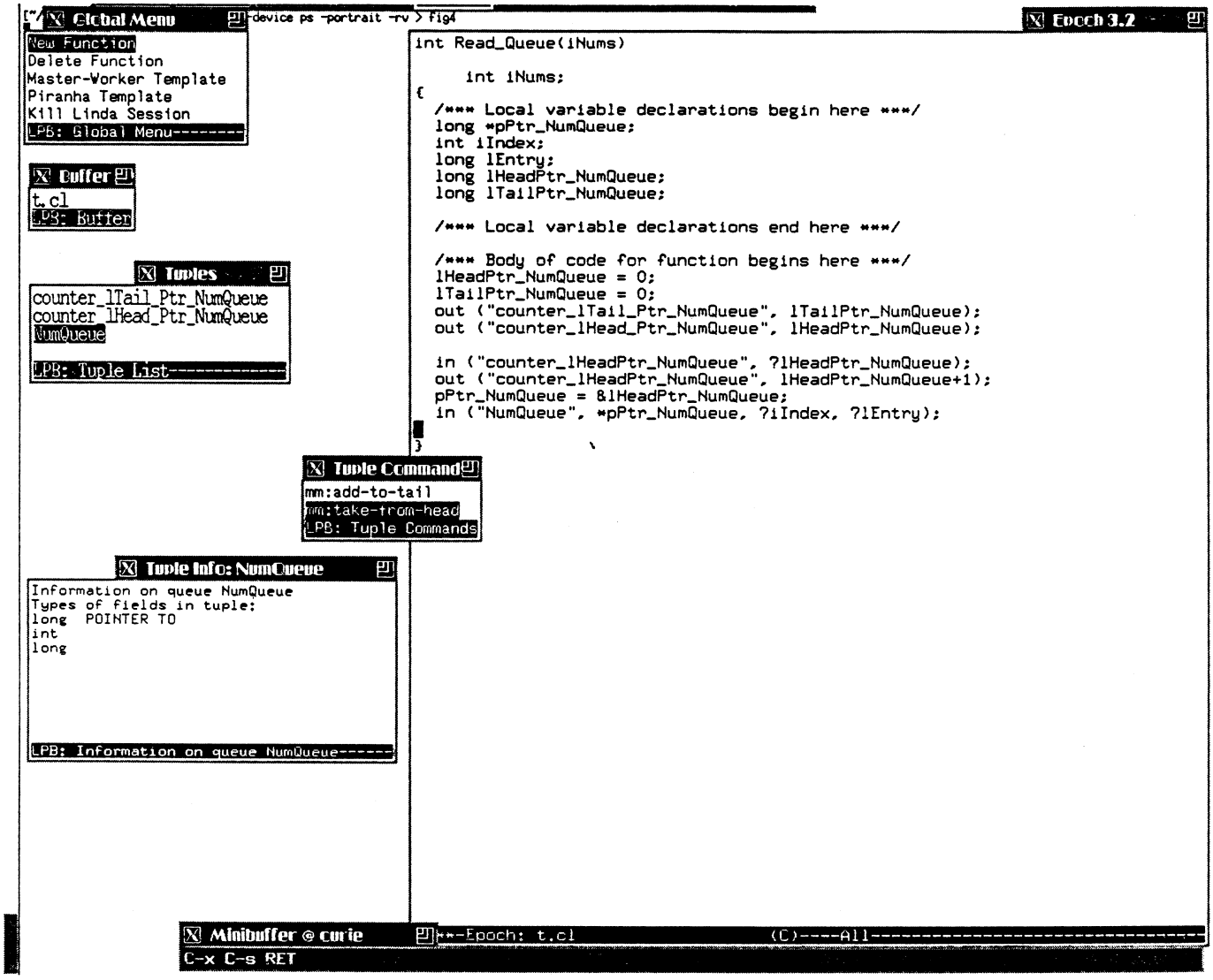

Figure 4: Queue manipulation operations

the queue is simplified, and the programmer specifies it at a "high" conceptual level. Second, the compiler can make use of the information to optimize the operations.

There is an inevitable question that arises here. If the programmer knows that he is removing an element from the queue, and the compiler makes use of this information, would it not be easier simply to add these higher level features directly to the base language? Why not extend C-Linda to offer higher-level queue operations, or throw Linda out and replace it with something "higher level"?

There are several reasons why not. Linda operations were meant to be simple and powerful. The simplicity of Linda as it stands gives users a flexible base from which to construct *any* desired operations. Adding new operations in addition to the old yields an over-complicated language; allowing new operations to supercede the old yields an insufficiently powerful one. The LPB consigns high level operations to the LPB level, general purpose operations to the language level. Note that complexity is nowhere near as damaging in software like the LPB as it is in the language itself. If some aspect of the LPB is of no use in some context, programmers don't have to learn it and *their implementations don't need to support it.* And Linda itself remains a simple *lingua franca* for the exchange of source code.

There is a second problem associated with higher-level languages of the kind described above. Languages must be fairly static. Neither programmers nor implementors can tolerate rapid or radical change. But programming *methodology* must evolve. In a young field like parallel programming, change can be rapid. Program builders such as the LPB provide a convenient alternative solution. In principle we would like to revise our language as methodologies become clear to us, without losing compatibility with the rest of the user community. At the same time, we don't want a galaxy of constructs that we do not need. If we produce mainly numerical scientific applications, the very-high-level constructs that are valuable to (say) graphics or database people may be of no use to us, and we don't want to bother with them: don't want them complicating our language manuals; don't want them complicating our compilers. Still, we want to *understand*, interface to, and execute graphics and database applications if we need to. All this would appear to be impossible to achieve within the programming language framework, but the LPB can support it. The LPB framework can be modified as desired and higher-level constructs can be added, tested and refined. The end product is Linda code that is portable and understandable, but the program construction phase offers the higher-level support that is needed. These operations can be site-specific, making use of the program-describing database that the LPB maintains. Given a portable and powerful base language, we can attach a specialized high-level program such as

the LPB on top to provide a high-level environment of the kind that programmers have always wanted but rarely obtained.

The second LPB example will address another aspect of this argument. Evidence suggests that there are cases where a Linda programmer needs to **in** one of a selection of tuples. Any one of the selection would do. We refer to this as an **or-in**, i.e. the program will **in** one tuple *or* another tuple *or* another one, and so on. This is an operation which Linda itself does not provide, and yet it is used often enough to be worth a debate over whether it should be incorporated into the language. Adding it to the language would involve a significant programming effort, and modification to the various kernels. The LPB is a convenient forum in which to test this construct.

The LPB implements the **or-in** function as a higher-level operation. If the user selects the menu option for an **or-in**, a menu pops up with a list of the tuples that are known to the database. The user is free to select which of these tuples will constitute the **or-in**. What gets inserted into the code is the higher-level operation. It appears to be a regular program construct, but the relevant lines in the code are underlined (Figure 5). The underlining indicates that it is a higher-level operation. Expanding this abstraction will indicate to the user how this is implemented in Linda code. The **or-in** becomes an **in** of a bit vector to check which tuples may be available. This is followed by a conditional which checks which bit is on, and based on that, reads in the appropriate tuple. The bit vector has to be generated whenever one of the tuples of the **or-in** is used in an **out** or **eval**. Thus, the expansion causes all relevant references to those tuples in all open modules to be followed by a new **out** which puts out a bit vector with the bit corresponding to the **out**'ed tuple turned on (Figure 6). If the cursor is placed on the main section of the **or-in** expansion, and the abstraction menu item is selected, all the expansion details disappear, and the abstraction reappears, making the **or-in** look very much as if it is a part of the language.

We have thus implemented a proposed language addition in the LPB, and enabled programmers to use and test it before it has actually been added to the language. This allows the operation to run through a trial period before the major task of adding it to the language is undertaken. Its usage patterns and usefulness need to be investigated before we commit ourselves to a language change, and the LPB is the testing medium.

There is one further advantage that the LPB offers over conventional languages with added features. The graphical user interface provides a level of user-friendliness that the base language cannot match. In combination with the abstraction facility, this becomes a powerful tool.

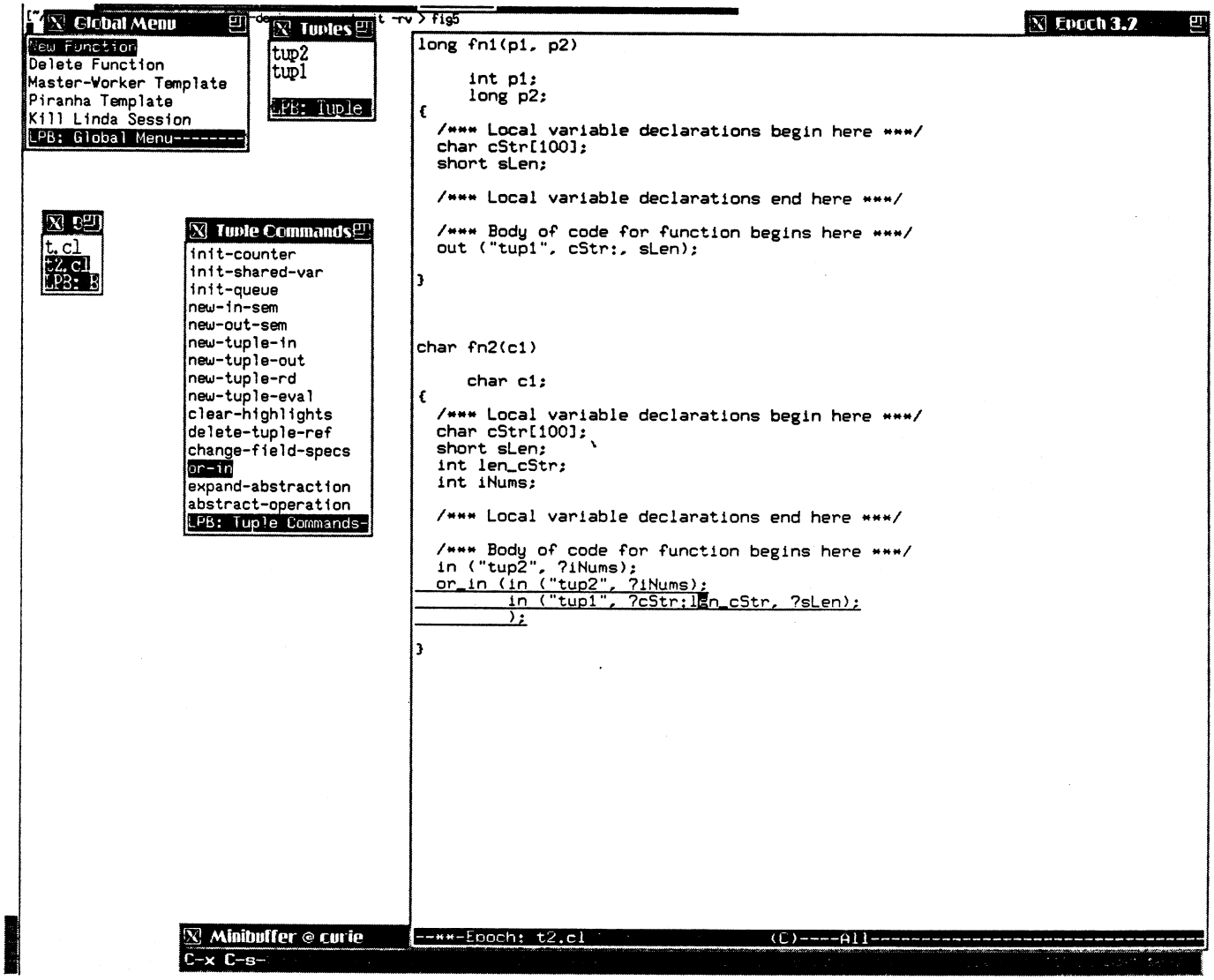

Figure 5: The **or-in** abstraction

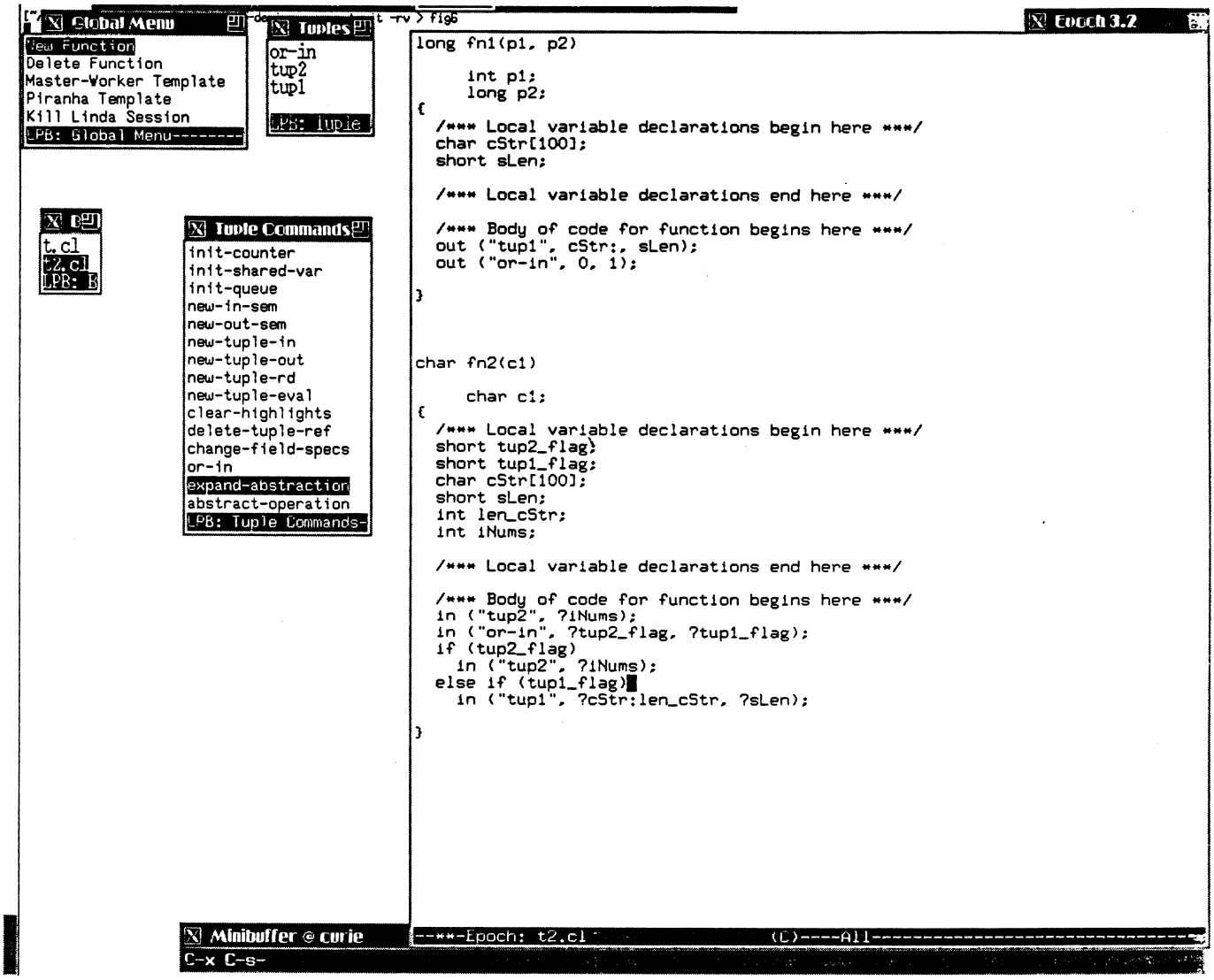

Figure 6: Expanded **or-in**

The disadvantage of the LPB in this context is efficiency. Wiring an operation into the base language allows the compiler to optimize its support. While the LPB does not provide that level of optimization, it does provide the compiler with semantic information that can lead to a different kind of optimization, as discussed in earlier sections. In combination with the above mentioned advantages, this amounts to a strong offsetting argument against the efficiency disadvantage.

5 Related work

The LPB's most important template-based structure editor predecessor is the Cornell Program Synthesizer [RT89b]. Unlike the synthesizer, however, the LPB does not enforce a rigid framework. Instead, the LPB captures methodologies and supports them, without imposing a strategy. What the LPB produces is source code, and the programmer is free to ignore or modify this as desired, a flexibility that is vitally necessary to any expert programmer.

Extensible parallel programming environments such as SIGMACS [SG91] generate a program database during compile time that can be used in later modifications to the program. The LPB, on the other hand, maintains a dynamic program-describing database that grows as the program is constructed. This allows the system to maintain semantic as well as syntactic information on the programs being developed. This information is used for guiding program development, for checking consistency, for documentation purposes, for providing optimizing information to the compiler [CG91], for benchmarking utilities to visualize performance in the spirit of [HE91], and for enhancing graphical monitoring.

There is currently much research effort in visualizing the dynamic behavior of parallel programs. [KC91] is a good example. Since the LPB can convey semantic information to a graphical monitoring tool [BC90], programmers can visualize dynamic information at a higher abstraction level than would otherwise be possible.

6 Conclusions

There is increasingly widespread agreement that programmers need to write explicitly parallel programs. Tools like the LPB are designed to support this in various ways.

The LPB is a CASE tool which supports basic tuple operations as well as higher-level functions, and offers complete templates and program structures. It captures programming methodologies and guides the user through program development.

Further, the LPB's program-describing database can supply information to other tools in the environment, enabling optimization at compile-time, enhanced visualization for monitoring at run-time, and performance monitoring for efficiency.

Most importantly, the LPB is characteristic of a potentially significant trend in programming language design. It addresses the traditional conflict between keeping a language simple and demanding that it be higher-level. The proposed solution is to combine a simple, general coordination language with a higher-level, domain-specific system that provides the power and higher-level abstractions that a programmer can selectively choose to employ. In sum, we can have our cake and eat it too. If we can capture the methods and idioms that skilled programmers rely on *without* complicating the language itself with a galaxy of high-level, special-purpose constructs, we have a solution to an important problem.

References

[ACG91] Shakil Ahmed, Nicholas Carriero, and David Gelernter. The Linda Program Builder. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990) (invited paper).* Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[Bar80] J.G.P. Barnes. An Overview of Ada. *Software Practice and Experience*, pages 851–887, 10 1980.

[BC90] Paul A. Bercovitz and Nicholas J. Carriero. TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs. Research Report 782, Yale University Department of Computer Science, April 1990.

[BCG+91] R. Bjornson, N. Carriero, D. Gelernter, D. Kaminsky, T. Mattson, and A. Sherman. Experience With Linda. Research Report 866, Yale University Department of Computer Science, 1991.

[CG89] Nicholas J. Carriero and David H. Gelernter. Linda in Context. *Communications of the ACM*, April 1989.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs.* The MIT Press, 1990.

[CG91] Nicholas Carriero and David Gelernter. A Foundation for Advanced Compile-time Analysis of Linda Programs. Technical report, Yale University Department of Computer Science, 1991.

[FG91] Scott Fertig and David Gelernter. A Software Architecture for Acquiring Knowledge from Cases. In *Proc. of the International Joint Conference on Artificial Intelligence*, August 1991.

[HE91] Michael T. Heath and Jennifer A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, September 1991.

[Jr82] Guy Steele Jr. An Overview of Common Lisp. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pages 98–107, 1982.

[Jr84] Guy Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[KC91] James A. Kohl and Thomas L. Casavant. Use of PARADISE: A Meta-Tool for Visualizing Parallel Systems. In *Proceedings of the Fifth International Parallel Processing Symposium.* IEEE Computer Society Press, April 30 - May 2 1991.

[RT89a] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator : a System for Constructing Language-based Editors.* Springer-Verlag, 1989.

[RT89b] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, 1989.

[SG91] Bruce Shei and Dennis Gannon. SIGMACS A Programmable Programming Environment. In *Proc. Third Workshop Languages and Compilers for Parallelism (Irvine, 1990).* Languages and Compilers for Parallel Computing II, MIT Press, 1991.

[SZBH86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, pages 419–490, October 1986.