The parallel C (pC) programming language

by R. Canetti
L. P. Fertig
S. A. Kravitz
D. Malki
R. Y. Pinter
S. Porat
A. Teperman

We describe pC (parallel C), an extension of the ANSI C programming language to support medium- to large-grain parallel programming in both shared- and distributed-memory environments. pC aims to make programming for parallel processors accessible to the C community by enriching the C programming model with a small set of constructs supporting parallelism. pC supports shared- and distributed-memory environments via a hierarchical computational model. A pC application comprises a static collection of *tasks* with disjoint memory spaces. A dynamic collection of *threads* runs within each task, sharing the data and code of the task. Language constructs specify concurrent execution of threads within a single task. Additional language constructs specify the interactions between threads through the following mechanisms: initiation of threads in remote tasks by *remote function call*, mailbox-based message passing, and synchronization primitives. The paper introduces the computational model and language constructs of pC and describes a prototype pC compiler and run-time system for the Mach operating system. Several program examples illustrate the utility of pC constructs.

1. Introduction

Currently, the development of applications for multiprocessor computers is a difficult and error-prone task requiring highly specialized professionals. If parallel-computing resources are to be properly exploited, it is critical that parallel-program development become accessible to the typical programmer, with minimal retraining.

We describe pC (parallel C), an extension of the ANSI C programming language [1] to support medium- to large-grain parallel programming in both shared- and distributed-memory environments. pC is designed to make parallel programming accessible to the C community by enriching the C programming model with a small set of constructs supporting parallelism.

In developing pC, we had the following goals:

- *Compatibility with C:*
  - Maintain compatibility with ANSI C source (i.e., pC should be a strict superset of ANSI C) and strict compatibility with C object code.
  - Keep all language extensions in the spirit of C, so that C programmers do not suffer "culture shock" when writing and reading pC programs.
- *Wide range of supported architectures:* Provide language support for a broad range of parallel-computing environments, including shared- and distributed-memory systems.

- *Dynamic resource allocation:* Provide flexibility by allocating resources and binding communication at run time.
- *Flexible parallel constructs:* Support both synchronous fork/join and asynchronous parallel activities.
- *Multiple levels of user control:* Facilitate parallel programming by users with a broad range of sophistication. We want to make it possible for expert users to tune their code by controlling resource allocation, without requiring novice users to do the same.

pC supports shared- and distributed-memory architectures via a hierarchical computational model. A pC application comprises a static collection of parallel *tasks* with disjoint memory spaces. A dynamic, parallel collection of *threads* runs within each task, sharing the data and code. Language constructs specify interactions between threads through the following mechanisms: initiation of threads in remote tasks by *remote function call*, mailbox-based message-passing, and synchronization primitives.

To our knowledge, no other parallel-language effort has attempted to support shared and distributed memory and provide object-code compatibility with existing C code.

In the remainder of this section, we examine other published efforts in light of the above-mentioned design goals. In Section 2 we describe the pC computational model and the layout of pC source code. In Section 3 we introduce the pC language constructs in their basic form, and in Section 4 we describe some advanced language features. Section 5 illustrates the use of pC constructs by presenting two pC program fragments. Section 6 describes our implementation of a pC compiler and run-time system for the Mach operating system [2]. We conclude, in Section 7, with a discussion of the language.

- *Rationale*

Several approaches have been taken to bridge the gap between the programmer and the efficient use of parallel computers. Parallelizing compilers [3–6] try to extract parallelism from programs written in sequential programming languages. The major advantages of this approach are the ability of programmers to continue programming in a known language, with a well-understood computational model, as well as the possibility of improving the performance of "dusty decks" without reprogramming. The major drawback of this approach is that it is restricted to loop-level parallelism that the compiler can detect or the user identifies with compiler directives. Parallelizing-compiler technology is progressing but is not yet mature.

Libraries [7, 8] and macro [9] packages have been used to extend serial-programming languages to support parallel-program development. This approach has the major advantages of not requiring a compiler, and operating within familiar language frameworks and computational paradigms. However, the limitations of the underlying programming languages often impose an awkward syntax and unnatural restrictions on the programmer. We feel that a language-level interface to the features supported by these libraries offers major improvements in programmer productivity: The resulting programs are more concise and readable, and easier to develop and maintain.

Parallel-programming languages integrate the abstractions required to construct parallel programs, within a programming-language framework. The goal is to provide concise language constructs that facilitate program development and can be compiled into efficient programs. Parallel-programming languages can be categorized as either new languages or extensions to existing languages.

The temptation is great to abandon existing programming languages in favor of new programming languages with new computational models. [A survey of previous work in the area of new languages is beyond the scope of this paper (see [10]).] However, the major pitfall of designing new languages and computational models is that they force the programmer to learn a completely new way of thinking before the programmer can write a parallel program. An additional and more serious problem, which any new programming language must address, is how to interface with the mountain of existing software written in popular programming languages. We believe that any parallel-programming language that is to be widely accepted must avoid these obstacles by defining a strict superset of a widely used serial-programming language.

Even within the context of C, attempts have been made to introduce auxiliary parallel-computational models. A good example is C-Linda [11], which introduces the powerful tuple-space model of computation. However, to exploit parallelism within an existing serial application may require a complete rethinking of the algorithm in terms of the tuple-space model. The tuple-space is also opaque and offers the programmer little control of its internal structure or opportunity for tuning. As an example, there is no explicit way to exploit locality.

Other attempts to extend C for parallelism introduce constructs to support coarse-grain distributed-memory parallelism [12] or fine-grain parallel computation within a single address space [13, 14], but not both. Concurrent-C [12] not only lacks support for parallel computation within shared memory but also restricts parallel interaction to an Ada-like [15] rendezvous model. EPEX-C [13] supports fine-grain parallelism in the form of self-scheduled parallel loops. Only parallel loops with integer indices and simple loop increments are supported.

pC supports both shared- and distributed-memory parallelism. The expressive power of pC exceeds the

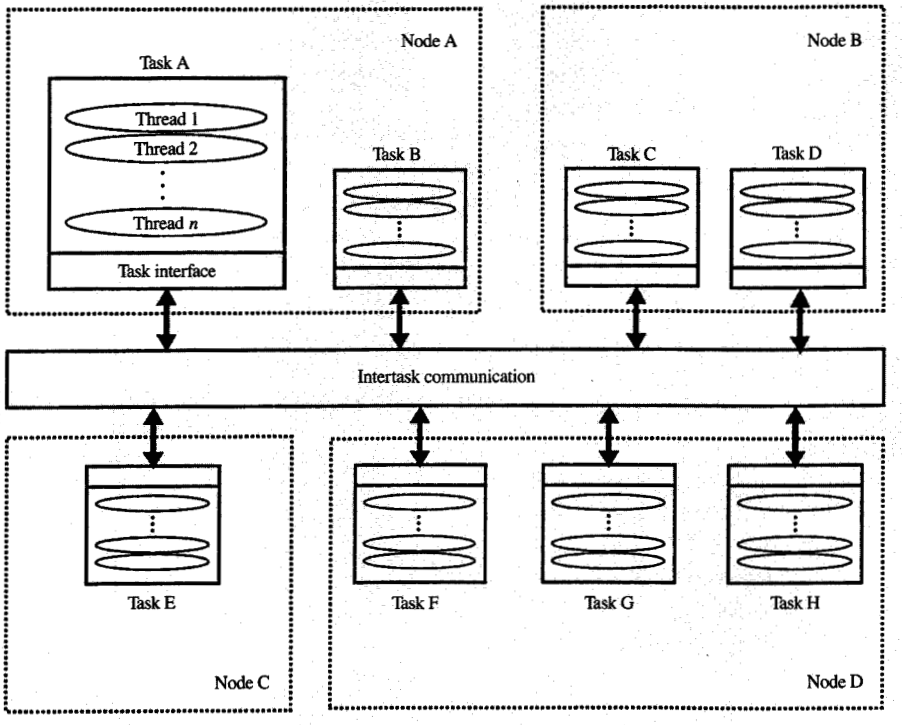


Figure 1

pC computational model.

aforementioned efforts, and pC introduces no artificial restrictions on language syntax. Furthermore, we think fine-grain parallelism is better dealt with by vector operators as proposed by [16], data-oriented approaches as exemplified by [17], or modern compiler optimization techniques, than by language constructs at the statement level.

2. Computational model and application layout

A pC application can be viewed as a program for an aggregate of logical processor nodes. Each node may be thought of as a shared-memory multiprocessor, i.e., a set of processing elements with a single memory space. An auxiliary configuration file defines the (static) mapping of nodes onto actual computational resources. More than one node may be mapped to the same physical host, but if so, their address spaces remain disjoint.

pC supports applications that consist of multiple *tasks*, each of which is loaded on a distinct logical node, as shown in **Figure 1**. This enables the loading of different portions of an application on unlike hosts. Each task supports multiple *threads* of execution, which share the code and data of the task. These threads can execute concurrently when parallel-execution resources are available. Multithreaded tasks are useful, even for programs written for networks of uniprocessors, since they facilitate the masking or hiding of message or input-output delays by a single program, just as multitasking improves throughput at the operating system level. Furthermore, threads may provide a useful way of structuring a program, even for a uniprocessor.

We now describe the important characteristics of the structure of pC programs and of the pC computational model.

Source layout The user specifies the source code of each task. The same task code can be loaded on more than one logical node, producing distinct tasks. One task is designated as the *main task*, containing the body of the C `main`[1] function. All activity in a pC application is started by

[1] Throughout the paper, we use a sans-serif font for C or pC language keywords.

a single thread, which executes the `main` function of the main task. The application terminates when all activity in the application terminates. Any existing C program can run unchanged as a single-task, single-thread pC application.

Each task is an executable instance of some compiled and linked task source program. The source code for a task contains a collection of functions and data, as does a regular C program. In addition, each task contains a set of entry-point functions called *remote functions*. The remote functions are visible outside the task as well as within it, and can be invoked through a remote function call mechanism. Tasks are referred to by global *task-ids*.

Generating parallel activity The basic dynamic schedulable unit of execution in pC is a thread. Multiple threads of control within a single task provide pC support for shared-memory parallel computation and are initiated by pC parallel constructs. The `par` and `parfor` constructs generate *groups* of threads that execute concurrently within a single task. Each thread in a parallel construct executes a C compound statement, which may in turn include remote function calls or parallel constructs.

Multiple tasks distributed across multiple hosts provide pC support for distributed-memory computing. Parallel activity is initiated across task boundaries (or locally) with the remote function call mechanism, as a result of which a new thread within the designated task is activated to perform the called function.

A parallel application is started as a single thread in a main task. This thread can initiate threads within its own task using parallel constructs, or threads in other tasks using remote function call.

Memory model All threads within a task share its global data and function space (subject to standard C scoping rules). Each thread has its own stack. Optionally, a thread that is initiated by a parallel construct is provided with private, initialized copies of variables. These variables are termed *frozen* variables, since their initial values are frozen at the time of thread activation. The handling of these frozen copies has no effect on the original variables.

Communication and synchronization pC defines application-wide communication and synchronization primitives—*mailbox* and *mutex* (mutual exclusion) objects. Mailbox and mutex objects are accessed with uniform syntax and semantics throughout a pC application. That is, the syntax and semantics of an operation on a mailbox (mutex) do not depend on the relative locations of the threads invoking the operation and the mailbox (mutex) itself. These objects are intended to be implemented via a run-time library and make use of synchronization primitives of the local environment. Any atomic synchronization operation in the underlying system is sufficient to support local thread interaction.

All dynamic pC computational resources, such as threads and mutexs, are referred to by *handles*. These handles, and task-ids, are first-class values, which may be communicated between threads and tasks by the usual pC communication mechanisms.

3. Basic language features

This section presents the central aspects of the pC language constructs. Additional features are discussed in the following section. A complete description of the pC language can be found in [18].

The essential aspects of pC are threads and their activation. Therefore, this section and the next one focus on the pC remote function call mechanism and parallel constructs. Each subsection includes pC code fragments in which new pC keywords appear in bold typeface and pseudocode appears in italic typeface.

- *Remote function call*

Multiple threads distributed across several tasks provide pC support for distributed computing. Threads can be activated across task boundaries with the remote function call (`rfc`) mechanism.

The `rfc` activates a remote function, as shown here:

```
result = rfc(task-id, f(param1, ... ));
```

The call of remote function `f` in a task with id task-id automatically allocates a thread in the target task to perform the function. The calling thread waits until the invoked function completes and the value computed by the function is returned. Multiple remote function calls to the same task, even to the same remote function, can be executed concurrently, since each call is executed by a separate thread. Parameters are passed to the remote function using standard C calling conventions. Care must be exercised if pointers are passed to a remote function, because pointer values may not be valid in the remote task. Implementations of pC in heterogeneous distributed environments must support data-format conversion of the return value of remote functions and their arguments.

- *Parallel constructs*

The remote function call mechanism does not by itself create parallelism: A thread performs a single remote function call and is forced to wait for its completion. It is essential to provide a way of generating multiple concurrent jobs. For this purpose, the pC language contains parallel constructs that spawn sets of threads. Each parallel construct defines a *group* of threads that can synchronize with the initiating thread. Variables can be passed to the member threads by the initiator. The pC parallel constructs can be nested to arbitrary depth.

pC provides the `par` construct to define parallel execution of a given set of statements. Each statement

within a par construct is executed by a distinct thread, and all threads begin execution concurrently. The par construct illustrated here executes two local function calls and a remote function call concurrently:

```
par <
     f()
     g(param1, ...);
     rfc(task-id, h(param1, ...) );
>
```

When all of the threads in the par construct set terminate, the par construct completes. The invoking thread may wait for the par construct to complete, or continue asynchronously, as determined by a keyword.

pC also provides parfor—an iterative parallel construct. The syntax of parfor is similar to the C for loop. The body of the loop contains a single statement to be executed by a distinct thread for each loop iteration. However, the evaluation of the loop iteration control is sequential. The threads created by the parfor loop are activated concurrently. Thus, the execution of the parfor loop occurs in two stages:

- First, the loop iterates sequentially. Each iteration allocates a thread and, optionally, passes to the thread copies of frozen variables (discussed below). The threads, however, do not start executing yet.
- Then, all threads are signaled to start execution together.

The user may provide the threads with private copies of global variables, e.g., the loop variable. The par_with directive provides each thread with a *frozen* copy of the specified variable(s). These values are passed at the first stage of the parfor execution (see above), and correspond to a snapshot of the program state at the appropriate sequential stage in the loop iteration. Here is a naive matrix multiplication algorithm that uses parfor and par_with:

```
f(){
     int i;
     parfor (i = 0; i < N; i++)

       par_with (i)/* 'frozen' i */
       {
         /* N separate threads execute this block,
          * one for each value of i
          */
          int j;
          int k;

          for (j = 0; j < N; j++)
               for(k = 0; k < N; k++)
                    c[i][j] += a[i][k] * b[k][j];
       }
}
```

The need for the freezing mechanism might not be obvious at first.

Unlike some parallel languages, pC does not limit parfor loop control to a FORTRAN-style index, and the construct can accept any loop control permitted by C. In the following example, the compiler cannot derive the private context of each iteration without the programmer directives, since the number of iterations and the values of the loop variable p are not known at compile time:

```
parfor (p = head; p != NULL; p = p->next)
     par_with(p)
          f(p);
```

The synchronous nature of the parallel constructs (the fact that all threads within the construct commence execution concurrently) is required to support the group abstraction (see Section 4) and the full semantics of the C for statement. Some advanced features of the threads and the parallel constructs are discussed at the beginning of Section 4.

Both the par and parfor loops generate a separate thread for each component statement. They therefore provide a mechanism for medium- to large-grain parallelism. They should not be confused with the small-grain parallelism provided by some parallel languages in a similar syntax. Therefore, we do not expect the preceding matrix multiplication example to be efficient, and use it for illustration only.

- *Mailboxes*

An obvious need in every parallel program is for communication among parallel threads. pC provides communication of unstructured messages via designated *mailboxes*. Mailboxes are communication endpoints, which are explicitly allocated and deallocated. Mailboxes are referenced through mailbox_handle values, which are created at allocation time and are unique across an entire application. A mailbox may be accessed by any thread that possesses a copy of the associated mailbox_handle value. Mailboxes can thus be used to mediate communication between threads within the same task or threads in different tasks, with location-independent syntax and semantics. In addition, a mailbox enforces a queuing protocol for messages and threads. The mailbox queuing protocols, established by create_mailbox, are described in more detail in Section 4.

Operations on a particular mailbox involve synchronization to maintain a consistent mailbox state. Shown here are the creation of a mailbox with the default protocol, and the basic send_mailbox and receive_mailbox operations:

```
mailbox_handle mh;
int status, length;
char *message, *buffer;

mh = create_mailbox();
...
status = send_mailbox(mh, message, length);
Check status

status = receive_mailbox(mh, buffer, length);
Check status
```

These operations pass unstructured message buffers between sender and receiver. A send_mailbox operation specifies a mailbox_handle, a pointer to the message data, and the length of the data. The sent message is transferred before the send_mailbox operation terminates. The operation returns either the number of bytes actually transferred or an error code. Similarly, receive_mailbox specifies a mailbox, a pointer to a data area for the incoming message, and the maximum length permitted for the message. receive_mailbox is blocked until a message is received. The operation returns either the number of bytes actually received or an error code.

- *Mutex objects*

In a hybrid environment of shared- and distributed-memory parallelism, it is beneficial to provide a fast synchronization mechanism in addition to the mailbox system. pC provides semaphore-like *mutex* objects and supports synchronization and queuing operations on them. The mutex_handle is a new type that identifies mutex objects throughout the pC application. The create_mutex operation returns a mutex_handle:

```
mutex_handle    mux_h;

mux_h = create_mutex();
```

The two basic operations on mutex objects are set_mutex and clear_mutex. set_mutex attempts to set the mutex. If the mutex is already set, the calling thread blocks and is added to the FIFO queue of waiting threads. If threads are waiting on a mutex, the clear_mutex operation releases the first waiting thread, leaving the mutex set. Otherwise, it clears the mutex. The example given here shows a typical usage of mutex objects for critical section protection.

```
set_mutex(mux_h);
{
    critical section
}
clear_mutex(mux_h);
```

4. Advanced language features

The pC language is designed to provide simple ways to express simple programs while offering a higher degree of control as an option for a sophisticated user. In the previous section, we presented the most common pC constructs in their simple forms. This section elaborates on these constructs and presents some additional constructs.

- *More on threads*

pC enables the user to explicitly control thread allocation and to manipulate thread resources. The code fragment shown here allocates a new thread and associates with it an identifier th of type thread_handle for further reference:

```
thread_handle  th;

th = create_thread(task-id);
```

The lifetime of a thread extends from the moment it is created with create_thread until it is freed via **free_thread**(th). This is the pattern of usage of *all* dynamically allocated control and communication handles in pC. For each kind of handle there exist explicit create and free operations.

The thread_handle returned by create_thread may be used to control a thread, query its status, or wait for its termination. Every thread has automatic access to its own thread_handle through the variable my_thread. A thread can communicate the value of this handle to other threads. A variant of the remote function call mechanism, rfc_thread, accepts a thread specifier. In the example shown here, the rfc_thread operation is used to cause a preallocated thread with handle th to execute a remote function f, with parameters:

```
result = rfc_thread(th, f(param1, ... ));
```

When the thread with handle th completes the execution of the remote function, the handle th remains valid, and the thread becomes inactive. Similarly, a thread that has been explicitly allocated by the user can be activated using the par and parfor constructs, as described in the next subsection. During its lifetime, the thread can be activated more than once.

- *More on parallel constructs*

The C statements within a parallel construct may be either simple or compound. As presented in Section 3, the threads in a parallel construct have no visible handles. A single, nameless thread executes each statement. As an option, a particular preallocated thread can be specified to execute a given statement. This feature can be used to check the status of or wait for the completion of asynchronously executing threads. In the example shown here, two concurrent threads are activated in the par construct:

```
thread_handle  th;
th = create_thread();
par <
    /* Thread th executes this statement (1) */
     par_thread(th) {
          f();
          g();
          h();
     }
     {       /* Statement 2 */
          rfc(task-id1, f1(param1, ...));
          rfc(task-id2, g1(param1, ...));
          rfc(task-id3, h1(param1, ...));
     }
>
```

Two things are worth noting in this example. First, statement 1 is executed by the thread th, as specified by the par_thread directive, while statement 2 is executed by an unspecified thread. Second, statement 2 causes the sequential activation of three additional threads through the remote function call mechanism.

Both par and parfor can be invoked asynchronously by specifying the keyword asynch before the construct. In this case, the invoking thread continues execution in parallel with the created group of threads.

- *Thread groups*

The par and parfor constructs implicitly create *groups* of threads. Every thread in a group has access to the group handle through the variable my_group. pC enables explicit association of a particular group with a parallel construct for synchronization purposes using the par_group directive, as shown here:

```
group_handle  gh;
gh = create_group();        /* Allocate a group */
...
asynch par par_group(gh) <
/* Activate threads in group */
     f();
     h();
>
...

wait_all_in_group(gh); /*Wait for group completion*/
```

The par_group directive associates the handle gh with the group of two threads—one thread executing function f and the other thread executing function h—invoked by the par construct. The two threads are invoked asynchronously. The group handle gh enables termination to be awaited subsequently using wait_all_in_group(gh). Additional operations on a group provide for termination of all or any of the threads in the group and for querying the status of the group.

- *Mailbox protocols*

Asynchronous message-passing with unlimited buffering is clearly powerful enough to support all communication requirements. For example, synchronous communication can be achieved by enforcing an acknowledgment protocol. This might, however, incur a high performance penalty because of additional message traffic and unnecessary buffering of the message at both source and destination. For instance, when the program follows a synchronous communication protocol, it should be able to direct the system to avoid the buffering required by an asynchronous protocol and transfer the message directly without intermediate buffering. For this purpose, the pC mailbox mechanism supports various synchronization and communication protocols.

The communication protocol and an optional limit on buffering space are set at mailbox-creation time:

```
mailbox_handle mh;

mh = create_mailbox(CONS, N);
        /* Buffered consuming with capacity N */
mh = create_mailbox(UNBUF);
        /* Unbuffered—direct transfer */
mh = create_mailbox(NONCONS);
        /* Buffered nonconsuming—capacity 1 */
```

The following protocols are supported by pC:

- *Consuming*
  Asynchronous (buffered) communication. A receive operation deletes a message from the head of the mailbox queue. A limit N may be set on the mailbox capacity, possibly causing subsequent send_mailbox operations to result in error values.
- *Unbuffered*
  Synchronous communication (rendezvous) with direct transfer. Both sender and receiver must await completion of communication. This protocol is intended to support the transfer of large buffers with minimal copying overhead.
- *Nonconsuming*
  A special mailbox with one-message capacity. The message is retained in the mailbox, for further receivers, after every receive_mailbox request. This protocol is intended to support a limited broadcast facility. Many threads can await a single event by performing a receive operation on a nonconsuming mailbox. When a message is sent to the mailbox, all waiting threads receive the message and proceed. The message remains in the mailbox until it is removed with a clear_mailbox operation.

Each task has an implicit consuming mailbox whose handle is accessible to threads within the task through the global

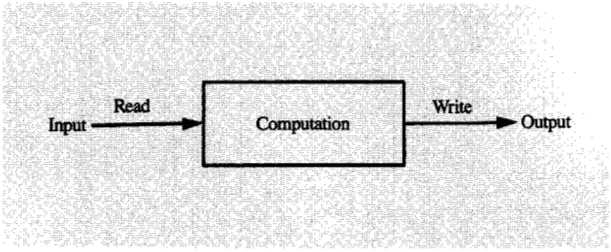


Figure 2

Program with opportunity to reduce delay due to I/O latency.

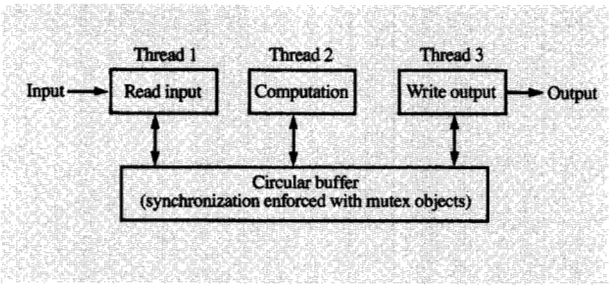


Figure 3

Multithreaded decomposition of serial program of Figure 2.

variable `my_task_mailbox`, and to threads in other tasks through the `get_task_mailbox` (task-id) expression. This built-in mailbox is intended to simplify the establishment of communication between threads in different tasks and to supply an initial set of communication endpoints.

5. Program examples

In this section, we present two program examples that illustrate the use of the pC constructs presented in the previous sections and demonstrate the usability and expressive power of pC. We emphasize code clarity over performance. In the examples, we sometimes replace serial code sequences with a description in italics. New pC types and constructs appear in bold typeface in the code fragments.

The first example demonstrates the benefit of using multiple threads to reduce delays due to I/O latency, by overlapping computation with I/O in a serial program, and the ease of expressing such a program in pC. The multithread code may run faster even on a uniprocessor.

The second example is a simple branch-and-bound solver for the well-known traveling-salesman program. pC code implementing the solver for both shared-memory and distributed-memory environments is presented.

• *Exploiting I/O latency within a single program*

The ability to specify the behavior of a program in terms of multiple threads allows a single program to exploit input-output latencies and communication latencies in a single program, as do multitasking operating systems. A serial program that benefits from a multithreaded description is sketched in **Figure 2**, and its parallel decomposition is shown in **Figure 3**. The program reads a stream of data from one file in chunks, performs intensive computation on each chunk, and writes the result to a second file. This is a generic serial program structure that arises in numerical applications such as digital filtering. The serial code would look like this:

```
main(){
   ...
   while(!done){
       read_chunk();      /* Reads a chunk */
       compute_chunk();   /* Computes */
       write_chunk();     /* Writes a chunk */
   }
   ...
}
```

By splitting the program into three separate threads communicating through a circular buffer (as shown in Figure 3), we can overlap computation with the input-output operations, even within a single program. Here is the parallel code:

```
main(){
   ...
   Initialize circular buffer with
   mutex synchronization;
   ...
   par <    /* Start three parallel threads */
       pread_chunk();      /* Reads a chunk   */
       pcompute_chunk();   /* Computes  */
       pwrite_chunk();     /* Writes a chunk  */
   >
}

/* pcompute_chunk and pwrite_chunk are similar */
pread_chunk(){
   while(!done){
      Get next available read buffer rb;
      read_chunk(rb);
      Make rb available for pcompute_chunk;
   }
}
```

Although some additional code is required to implement the circular queue, the read, write, and compute components of the original serial program are unchanged. An implementation of a stylized filter of this type using our prototype implementation of pC on an IBM RT Personal

```
struct partial_path {
   int  cities[NCITIES];                              /* Describes path */
   int  pathlen;                                      /* Number of cities on path */
   int  cost;                                         /* Cost of path */
   long cities_used;                                  /* Bit mask */
};

/* ExtendPaths:
 * Extend a partial path by one position in all possible ways, and deposit new partial paths
 * in joblist.
 */
void
ExtendPaths(mailbox_handle joblist,                   /* Used by GenerateJob */
            mailbox_handle minpath,                   /* Used to Get and Broadcast Bound */
            mutex_handle minpath_mutex,               /* mutex on minpath */
            struct partial_path *base)                /* Partial path to be extended */
{
            struct partial_path *local_minpath;       /* Placeholder for GetBound */
            int i;

  if (base->pathlen == NCITIES) {                     /* This is already a full path */

       Compute full cost of path;

       if (base->cost < GetBound(minpath, local_minpath))   /* Better than global bound */
            BroadcastBound(minpath, minpath_mutex, base);
       return;
  }

  for (i = 0; i < NCITIES; i++) {                     /* Attempt all possible extensions */
       if (!(BIT(i) & base->cities_used)) {           /* City 'i' not already used in path */


           Extend path in 'base' with city 'i' and compute new cost;

           if (base->cost < GetBound(minpath, local_minpath))   /* This is the pruning condition */
                GenerateJob(joblist, base);          /* Deposit current partial path in joblist */

           Roll back last extension (i.e., remove city 'i' from path);
       }
  }
}
```

Figure 4

ExtendPaths routine for the parallel traveling-salesman problem solver.

Computer® running Mach yielded a 25–30% speedup over the original serial program.

- *Parallel traveling-salesman problem solver*

Given a set of cities and the distance between each pair of cities, a solution to the traveling-salesman problem is the minimum-length path that visits each city exactly once. In this section we present a branch-and-bound [19] traveling-salesman solver and describe its parallel implementation in pC.

A *complete path* is an ordered list of all the cities, with a common start and end city. A *partial path* consists of an

ordered list of a subset of the cities. The *path length* of a path is the total distance traveled from start city to end city.

The ExtendPaths routine shown in **Figure 4** extends a partial path with each possible (i.e., as yet unvisited) city. The loop in ExtendPaths considers in turn each possible single-city extension to the base path. If the path length of such an extended partial path exceeds the length of the best complete path yet known, as returned by GetBound, this partial path clearly does not lead to an improved solution and is pruned (i.e., discarded). Otherwise, the extended path remains a possible basis for a solution and is added to the job list, using GenerateJob. The original base path is then restored by deleting the city just added before the next path extension is considered. If a path is completed whose length is shorter than the best solution known, BroadcastBound is used to establish it as the best solution known.

We start with a partial path that contains a single city and an arbitrary complete path as the best solution yet known. At each step, a partial path is selected and is extended with ExtendPaths. This continues until no partial paths remain to be explored. Since the extensions to each partial path are disjoint, multiple path extensions can be performed independently, in parallel. This leads to a work arrangement that employs a central job list and multiple (identical) threads fetching partial paths from the job list and adding extended partial paths to the list. Threads can also retrieve and update the best solution known.

Management of the job list and global bound is shown below. The job list is implemented using a buffered pC mailbox. GenerateJob simply sends a partial-path data structure to the joblist mailbox. Similarly, GrabJob receives a partial path from the mailbox. The best solution known is stored in a nonconsuming mailbox, minpath.

```
void GenerateJob(mailbox_handle joblist,
                 struct partial_path *pt)
{
  send_mailbox(joblist, (char *)pt, sizeof(*pt));
}

void GrabJob(mailbox_handle joblist,
             struct partial_path *pt)
{
  receive_mailbox(joblist, (char*)pt, sizeof(*pt));
}
```

Threads access the current best bound with GetBound, which receives the message stored in minpath. Updating the bound must be done atomically, so the BroadcastBound function enforces atomicity by using a mutex object, minpath_mutex, as shown in **Figure 5**.

The function executed by the threads is an infinite loop that retrieves jobs from the job list and tries to extend them, as shown here:

```
void
pmain(mailbox_handle joblist,
      mailbox_handle minpath,
      mutex_handle minpath_lock)
{
struct partial_path base;

  /* Main body of worker thread */
  for (;;) {
      /* Get a job from the list */
      GrabJob(joblist, &base);
      /* Try to extend the path */
      ExtendPaths(joblist, minpath, minpath_lock,
                  &base);
   }
}
```

The main body of the application creates and initializes the job list and global bound, and sets up parallel activity via parfor, as shown in **Figure 6**. The global minimum is found when all worker threads are waiting and there are no more job descriptors in the job list. The main thread detects this situation, prints the solution, and terminates the worker threads. Termination is facilitated by associating all worker threads with the group g.

The single-task traveling-salesman solver just described is easily adapted for a multitask implementation. The central job list and global bound are implemented via mailboxes and will therefore work unchanged in a multitask environment. Only the initialization of parallel activity requires modification. In the single-task version, the parfor statement establishes worker threads within the current task by invoking the pmain function directly. In the multitask version, a parfor invokes remote function calls that invoke the pmain function in other tasks, as shown below. The rfc also transfers the values for the joblist, minpath, and minpath_mutex handles, facilitating use of these resources by all of the invoked threads. Each task has an integer index but is manipulated using a task-id, with get_task providing the mapping:

```
asynch parfor (i = 0; i < nworkers; i++)
    rfc(get_task(i),
        pmain(joblist, minpath, minpath_lock));
```

Performance of both the shared- and distributed-memory versions of the solver could be improved by reducing the frequency of reads and writes to the job list. This is easily achieved by modifying ExtendPaths to perform path extensions of more than one city before reinserting partial paths in the job list.

6. Implementation

In this section, we describe our prototype implementation of a pC compiler and run-time system for the Mach

```
/*  GetBound returns the cost of the best bound */
int GetBound(mailbox_handle minpath, struct partial_path *pt)
{
  receive_mailbox(minpath, (char *)pt, sizeof(*pt));
   return pt->cost;
}

/* BroadcastBound:
 * A partial solution is submitted as best solution to 'minpath'.
 * The query and update are made atomic, to ensure storing of best bound.
 */
void BroadcastBound(mailbox_handle minpath,
                    mutex_handle minpath_mutex,
                    struct partial_path *pt)
{
   struct partial_path  best;

   set_mutex(minpath_mutex);                          /* Start atomic */
   GetBound(minpath, &best);
   if (best.cost > pt->cost) {                        /* New bound is better */
        clear_mailbox(minpath);
        send_mailbox(minpath, (char *)pt, sizeof(*pt));
   }
   clear_mutex(minpath_mutex);                        /* End atomic */
}
```

Figure 5

GetBound and BroadcastBound routines for the parallel traveling salesman problem solver.

operating system [2]. Since we intended that pC eventually run on many systems, emphasis was placed on an efficient, yet easily portable, system. Parallel machines are quite diverse in both instruction-set architectures and software services. Therefore, if portability is an issue, it is unwise to have the compiler target a particular assembly language or a particular set of system services. Thus, in our implementation of a pC compiler we translate pC code into standard C code with calls to a run-time library. The run-time library supports an abstract machine model between the language and the underlying system. This facilitates portability, since a port to a new hardware or software architecture requires modifications to only small, system-specific portions of the run-time library, while the compiler remains unchanged.

- *The pC compiler*

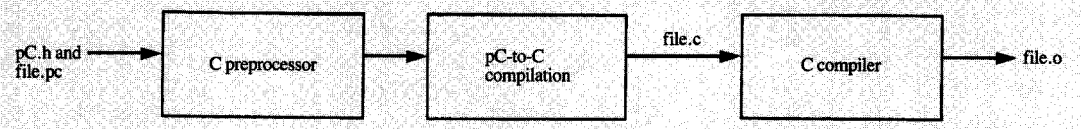
The pC grammar [18] is a strict superset of that of C. There are two forms of language extensions: control constructs (e.g., `par`) and built-in functions (e.g., `create_thread`). The work of the compiler is mainly to produce correct C translations of the new control constructs. The new built-in functions are dealt with simply by introducing library functions with the same name and declaring function prototypes. The new pC types are dealt with through `typedefs`. The compilation of a pC program is illustrated in **Figure 7**. Before the user's source file is preprocessed, it is appended to an include file that contains the function prototypes for built-in functions and the typedefs for the pC data types. A task executable is produced by linking object files produced from C and pC source files with the pC run-time library, as shown in **Figure 8**.

The essential task of the pC compiler is to perform the resource management (i.e., allocation of threads and groups) implicit in the parallel constructs and to generate C code for statements appearing within the pC parallel constructs. This generated C code must correctly conform to the pC memory model. The code to be executed by each thread within a parallel construct is encapsulated in a

```
main()
{
        struct partial_path solution;
        mailbox_handle joblist;
        mailbox_handle minpath;
        mutex_handle minpath_mutex;
        group_handle g;

        g = create_group();
        joblist = create_mailbox(CONS, UNBOUNDED);  /* joblist is a mailbox */
        minpath = create_mailbox(NONCONS);          /* Global bound is a nonconsuming mailbox */
        minpath_mutex = create_mutex();             /* Lock for global bound coherency */

        Create an initial path consisting of a single city and deposit in joblist;
        Compute the cost of an arbitrary path and deposit in minpath;

        /* Invoke nworkers threads executing pmain */
        asynch parfor par_group(g) (i = 0; i < nworkers; i++)
                pmain(joblist, minpath, minpath_lock);

        Wait until joblist mailbox is empty, and all threads are waiting to receive;
        Solution is current bound;

        receive_mailbox(minpath, (char *)&solution, sizeof(solution));
        Print solution;

        kill_group(g);        /* Kill all threads */

}
```

Figure 6

Main function of the parallel traveling salesman problem solver.



pC.h includes typedefs for pC types and function prototypes for pC built-in functions and is included automatically by the pC compiler.

Figure 7

Compilation of a pC program.

distinct compiler-generated function. Automatic variables visible in the context of the parallel construct are shared by all threads invoked within the construct. The pC compiler must ensure that the functions executed by the invoked threads can access all automatic variables referenced within the parallel construct. The original code is modified to refer to the automatic variables through dereferenced pointers, and these pointers are passed as parameters. We illustrate here the required code transformation. In the following example, the automatic variable `i` is referenced inside the `par` construct:

```
main(){
   int i;

   par <
        i++; /* References automatic variable i */
   >
}
```

The statement inside the `par` is converted to a function, `PC_00037`, and `i` is referenced through a pointer passed as a parameter of the function. Frozen variables and nesting of `par` constructs introduce additional complexity that is handled by the compiler. Here is the C code generated by the pC compiler:

```
/* Function generated by pC compiler */
PC_00037(int *i){
       Glue code;
       (*i)++;  /* References original automatic i
                 * through pointer
                 */
       Glue code;
}

main(){
   int i;

   Par glue code;

   Invoke thread to execute PC_00037(&i),
   pass address of i;

   Par glue code;

}
```

In designing the pC compiler, we adopted a minimalist approach to error checking. That is, we perform only the correctness checks required to support correct parsing of pC source or to report errors that the C compiler would detect, but would provide the user with an unintelligible diagnostic. It is critical to report all errors in a way that permits the user to relate them to the original pC source code, without having to delve into the C code generated by the pC compiler. Full parsing of the pC source is required to detect and report pC (rather than C) syntax errors, and

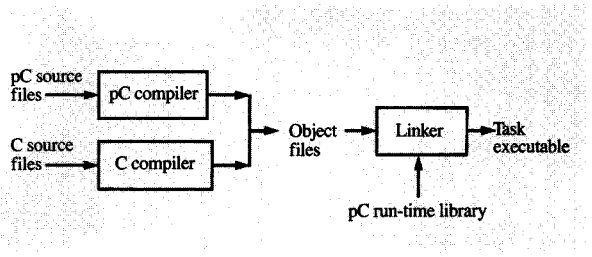


Figure 8

Creating a task executable from C and pC source files.

to fully analyze data declarations and remote function definitions. A shift-reduce parser with error recovery for the pC LALR(1) grammar is automatically generated by the LPG[2] tool[3]. A symbol table of data declarations is maintained. No type-checking of expressions is performed, however, since the type-correctness of all pC language constructs is verified by the C compiler. The pC compiler outputs compiler directives that permit the C compiler to associate errors in the C file output by the pC compiler with the location of the errors in the original pC source file.

- *pC run-time support*

The pC run-time package constitutes the logical-machine support for the pC program. Every pC object-type has a type-manager, i.e., a *thread manager*, a *group manager*, a *mutex manager*, and a *mailbox manager*. These are logical entities consisting of the type definition, its allocation management, and the operations provided for manipulating it. Each object-type has a set of associated interface routines. The pC compiler produces calls to these interface routines and gains access to the pC services by linking the program object modules with the pC run-time library.

The invocation of the service operations by the interface routines is mediated by a logical *local-request server* (LRS). This server is designed to permit extension to a multitask implementation. The LRS in a multitask implementation should transfer all requests referring to nonlocal objects to the servers in remote tasks. This is done transparently, while preserving the uniform interface.

The single-task implementation of pC for the Mach operating system [2] is an extension of the C Threads library [7] developed at Carnegie Mellon University. The underlying support layer for multiprocessing is the C Threads *cprocs* layer. The cprocs mechanism is a software

[2] Philippe Charles, Gerry Fisher, and Yvonne Lesesne, *LALR Parser Generator Version 2.0 User's Guide*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY.

[3] We are currently reimplementing the pC parser using YACC [20] and LEX [21], to enable dissemination of the prototype compiler.

layer supporting concurrent execution threads and providing operations on them such as allocation, control, and synchronization. The pC run-time package can be ported to a system if the system provides the cprocs functionality. Thus, the cprocs interface serves as the definition of the pC package environment requirements.

7. Discussion

In this paper we have presented the design of pC, a new language for parallel processing. The examples presented, as well as our experience developing programs with pC, demonstrate that pC has met our initial design goals. In particular, pC is easy to integrate with existing code and easy to learn.

We chose the C language as the base for extension for practical reasons only. We believe that C++® [22] is a more suitable language for extension, since it better accommodates new object definition and handling. For example, we chose to omit typed mailboxes with structured messages, which are cumbersome in C; however, in C++, structured mailboxes could easily be defined as a derived class of unstructured mailboxes. Similarly, `rfc` syntax could be streamlined.

We are currently extending our implementation of pC to support multitask applications, adapting our run-time library to use the POSIX pthreads library [23] instead of Mach cthreads, and gaining experience developing programs in pC. As a result, we are contemplating several language extensions to support additional mailbox protocols and additional operations on groups of threads.

Acknowledgments

We have benefitted from helpful discussions with Anthony Bolmarcich, Howard Operowsky, Michael Rodeh, Leslie Scarborough, and Marc Snir. We thank the Toronto Programming Laboratory for supplying us with a C grammar, and Jerry Fisher at the IBM Thomas J. Watson Research Center for supplying us with LPG. Finally, we thank the anonymous reviewers for their comments.

References

1. Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Second Edition, Addison-Wesley Publishing Co., Inc., Reading, MA, 1989.
2. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for Unix Development," *Proceedings of Summer Usenix*, Atlanta, July 1986, pp. 93–113.
3. C. D. Polychronopoulous, D. J. Kuck, and D. A. Padua, "Optimal Processor Allocation to Nested Parallel Loops," *Proceedings of the International Conference on Parallel Processing*, Institute of Electrical and Electronics Engineers, New York, 1986, pp. 519–527.
4. F. E. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Distributed Computing* **5**, No. 5, 617–640 (October 1988).
5. A. Carle, K. D. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S. K. Warren, "A Practical Environment for Scientific Programming," *IEEE Computer* **20**, No. 11, 75–89 (November 1987).
6. D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *J. Supercomputing* **2**, No. 2, 151–169 (October 1988).
7. E. C. Cooper and R. P. Draves, "C Threads," *Report No. CMU-CS-88-154*, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, June 1988.
8. B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Presto: A System for Object Oriented Parallel Programming," *Software—Pract. & Exper.* **18**, No. 8, 713–732 (August 1988).
9. J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, and J. Patterson, *Portable Parallel Programs for Parallel Processors*, Holt, Rinehart, & Winston, New York, 1987.
10. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tannenbaum, "Programming Languages for Distributed Computer Systems," *ACM Computing Surv.* **21**, No. 3, 261–322 (September 1989).
11. N. Carriero and D. Gelernter, "Linda in Context," *Commun. ACM* **32**, No. 4, 414–458 (April 1989).
12. N. H. Gehani and W. D. Roome, "Concurrent C," *Software—Pract. & Exper.* **16**, No. 9, 821–844 (September 1986).
13. W. Chang and A. Norton, "EPEX/C," *Research Report RC-12572*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, February 1987.
14. L. Rudolph and Y. Ben-Asher, "The PARC System," *Research Report No. CS-88-8*, Hebrew University, Jerusalem, Israel, August 1988.
15. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, U.S. Department of Defense, Washington, DC, 1983.
16. *NCEG Document Register*, NCEG 90-000, 1991, Numerical C Extensions Group, ANSI Working Group X3J11.1, R. Jaeschke, convener, 2051 Swans Neck Way, Reston, VA 22091.
17. D. Klappholz, A. Kallis, and X. Kong, "Refined C: An Update," *Second Workshop on Languages and Compilers for Parallel Processing*, Urbana, IL, August 1989; MIT Press, Cambridge, MA.
18. Ran Canetti, L. Paul Fertig, Saul A. Kravitz, Dalia Malki, Ron Y. Pinter, Sara Porat, and Avi Teperman, "The Parallel C (pC) Programming Language," *Technical Report 88.307*, IBM Israel Science and Technology, Haifa, June 1991.
19. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1983.
20. S. C. Johnson, "YACC—Yet Another Compiler Compiler," *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
21. M. E. Lesk, "LEX—A Lexical Analyzer Generator," *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
22. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Co., Reading, MA, 1986.
23. Computer Society Technical Committee on Operating Systems, *Threads Extensions for Portable Operating Systems*, Order No. P1003.4aD4, Institute of Electrical and Electronics Engineers, New York, August 1989.

Received October 16, 1990; accepted for publication January 25, 1991

Ran Canetti *Department of Computer Science, Technion, Haifa 32000, Israel.* Mr. Canetti received the B.A. degree in computer science in 1988 and the B.A. degree in physics in 1989, both from the Technion–Israel Institute of Technology, Haifa. He is currently completing his M.Sc. degree in computer science at the Technion. Mr. Canetti spent the summer of 1990 at the IBM Israel Scientific Center working on parallel C. His interests include computational complexity and parallel and distributed computation.

L. Paul Fertig *IBM Israel Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Fertig received a B.Sc. degree in mathematics from the University of Manchester, England, in 1978, and a D.Phil. degree in computing science from the University of Oxford, England, in 1982. From 1983 to 1984 he was a postdoctoral research officer at the University of Oxford; from 1985 to 1986 he was a software engineer for ICL, West Gorton, Manchester, England. In February 1988 Dr. Fertig joined the IBM Israel Scientific Center, where he is currently involved in the development of microcoding environments. His research interests include techniques for formal specification, parallel languages, and persistent languages. Dr. Fertig is a member of the Association for Computing Machinery and ACM SIGPLAN.

Saul A. Kravitz* *IBM Israel Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Kravitz received the B.A. degree in physics from The Johns Hopkins University, Baltimore, in 1982 and the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, in 1986 and 1989, respectively. Since 1989, he has been a Research Staff Member at the IBM Israel Scientific Center, where he is currently involved in developing programming language support for parallel computers, and optimizing compilers. He is the leader of the pC project. His interests are in the areas of parallel and distributed computation and computer architecture. Dr. Kravitz is a member of the Institute of Electrical and Electronics Engineers, Phi Beta Kappa, and Sigma Xi.

Dalia Malki *Department of Computer Science, Hebrew University, Jerusalem 91904, Israel.* Ms. Malki received the B.Sc. and M.Sc. degrees in computer science from the Hebrew University in Jerusalem in 1985 and 1988, respectively. She is currently pursuing a Ph.D. degree in the area of high availability for distributed systems in the Computer Science Department of the Hebrew University in Jerusalem. During the years 1988–1990 she worked at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. In 1990 she was a Research Fellow at the IBM Israel Scientific Center, Haifa. Ms. Malki's interests include parallel programming and distributed operating systems.

Ron Y. Pinter *IBM Israel Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Pinter received the B.Sc. degree in computer science from the Technion–Israel Institute of Technology, Haifa, in 1975, and the S.M. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1980 and 1982, respectively. From 1982 to 1983 he was a Member of the Technical Staff in the Computing Sciences Research Center, AT&T Bell Laboratories, Murray Hill, NJ. In December 1983 he joined the IBM Israel Scientific Center, where he is currently the manager of the Programming Languages and Environments Department. He is also an Adjunct Senior Teaching Associate with the Electrical Engineering Department at the Technion. Dr. Pinter has taught at the Hebrew University, Jerusalem, and spent the academic year 1988–1989 as a Visiting Scientist in the Department of Computer Science, Yale University, New Haven, Connecticut. His research interests include parallel-programming techniques, code-generation algorithms, and layout for integrated circuits. Dr. Pinter is a member of the Association for Computing Machinery, ACM SIGPLAN, and the IEEE Computer Society.

Sara Porat *IBM Israel Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Porat received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the Technion–Israel Institute of Technology, Haifa, in 1977, 1981, and 1986, respectively. From 1978 to 1981, 1982 to 1986, and 1986 to 1990 she was a Teaching Assistant, an Instructor, and a Lecturer, respectively, in the Computer Science Department at the Technion. From 1986 to 1987 and 1987 to 1988 she was a Research Associate and an Assistant Professor, respectively, in the Computer Science Department of the University of Rochester, New York. In February 1990, she joined the IBM Israel Scientific Center, where she is currently a Research Fellow in the Programming Languages and Environments Department. Dr. Porat's research interests include formal specification of systems and programs, semantics of programming languages, and machine learning.

Avi Teperman *IBM Israel Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Teperman received the B.Sc., M.Sc., and Ph.D. degrees in electrical engineering from the Technion–Israel Institute of Technology, Haifa, in 1968, 1971, and 1979, respectively. From 1979 to 1982 he was a Lecturer in the Department of Electrical Engineering at the Technion. In December 1982 he joined the IBM Israel Scientific Center, where he is currently working on operating systems, distributed environments, and parallel languages. Dr. Teperman is also an Adjunct Teaching Associate with the Computer Science Department at the Technion; he spent a year (1987–1988) as a Visiting Scientist at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York. His research interests include, in addition to the above, code-optimization algorithms and object-oriented systems.

*Correspondence concerning this paper should be addressed to Dr. Kravitz.