PDDP, A Data Parallel Programming Model

KAREN H. WARREN

Lawrence Livermore National Laboratory, Livermore, CA 94551; e-mail: kwarren@tazdevil.llnl.gov

ABSTRACT

PDDP, the parallel data distribution preprocessor, is a data parallel programming model for distributed memory parallel computers. PDDP implements high-performance Fortran-compatible data distribution directives and parallelism expressed by the use of Fortran 90 array syntax, the FORALL statement, and the WHERE construct. Distributed data objects belong to a global name space; other data objects are treated as local and replicated on each processor. PDDP allows the user to program in a shared memory style and generates codes that are portable to a variety of parallel machines. For interprocessor communication, PDDP uses the fastest communication primitives on each platform.

1 INTRODUCTION

In order to achieve utilization by a large percentage of the scientific community, today's high-performance computers require a high-level programming model. In particular, a shared memory programming environment permits users to concentrate on the algorithms of the code rather than on the details of data communication. The alternative, message passing, has been described as the assembly language of parallel computers.

In 1992, members of the Massively Parallel Computing Initiative project at Lawrence Livermore National Laboratory (LLNL) proposed writing an experimental translator that would allow the user to code in a high-level Fortran-based SPMD language. The resulting code would make efficient use of MIMD computers with nonuniformly accessible memories. The project goals were to examine the technology involved and to investigate the merits of such a language, including whether such an architecture-independent language could indeed be used efficiently on any parallel computer with distributed memory. A valuable additional benefit for both implementors and users would be to gain experience in parallel processing with a high-level programming model.

In this article, we present the resulting language model, PDDP, the parallel data distribution preprocessor. We present the syntax and semantics of PDDP, describe its implementation, discuss portability issues, and present data on its performance.

2 BACKGROUND

PDDP is a hybrid of PFP [1], a parallel Fortran preprocessor used at LLNL, and Fortran D [2], a research compiler from Rice University. Fortran D provides an extensive set of declarations for distributing data across processor memories and also serves as a base for the high-performance Fortran (HPF) [3] distribution directives. Over the past 2 years, the High-Performance Fortran Forum has focused on the need for a high-level Fortran parallel programming model. The resulting HPF lan-

Received June 1995
Revised December 1995

Scientific Programming, Vol. 5, pp. 319–327 (1996)
CCC 1058-9244/96/040319-09

guage specification is a published model ready for implementation [3]. Because PDDP contains a subset of HPF, PDDP codes are easily converted to HPF.

Its other predecessor, PFP, is a task-oriented parallel Fortran programming language. In the PFP programming model, all of the processors, requested at run-time and referred to as a team, enter the main routine in parallel. The user directs this team through the application with the option of dividing the team into subteams to perform tasks in parallel. PFP offers the familiar shared memory programming model elements, including barriers and shared and private storage attributes for variables. In a similar manner, all of the processors requested at run-time execute each statement of a PDDP code except for master blocks and parallel code segments. The processors execute the code statements, in a semisynchronous manner, uninhibited by implicit synchronization in any of the constructs. This multithreading aspect avoids the explicit forking of the processors for each parallel loop. PFP provides a synchronization tool, the `barrier` statement. This construct allows the user to explicitly synchronize the processors and avoids unnecessary implicit barriers. Currently, PDDP does not implement team splitting for parallel tasks; rather parallelism is expressed in the HPF FORALL, the Fortran 90 array syntax, and WHERE statements.

3 PDDP SYNTAX

PDDP consists of a one-pass parser-translator and a run-time library. The parser accepts a superset of Fortran 77 statements. For each source statement, the parser builds a parse tree used to generate Fortran 77 code. User declarations include a subset of HPF TEMPLATE, and ALIGN specification directives. The parser builds a symbol table of declared scalars, arrays, templates, common blocks, and subroutines. For array and template declarations, it records the number of dimensions and extents. It recognizes array-slice and whole-array syntax as well as individual-distributed array accesses. It recognizes distributed arrays used in subroutine arguments and common statements. The use of Fortran 90 [4] array syntax, the WHERE construct, and the HPF FORALL statement imply parallel execution by the members of the team.

The PDDP parser recognizes the following HPF distribution specification directives: TEMPLATE, DISTRIBUTE, and ALIGN. Together they indicate the mapping of the data to the processor memories. An abstract array is first declared using the TEMPLATE statement. It is partitioned among the processors using the DISTRIBUTE statement along with an HPF data distribution type for each dimension:

1. BLOCK places successive array elements on the same processor, moving to the next processor when the block size, equal to the extent divided by the number of processors, has been used up.
2. CYCLIC causes successive elements of the array to be placed on successive processors in the system, wrapping around after the last processor.
3. The degenerate distribution "*" leaves the entire dimension on a single processor.

Actual arrays are associated with the abstract template using the ALIGN statement.

Distributed arrays are globally accessible and are distributed across the processor memory regions. For communication purposes, PDDP also provides global objects that are not distributed but may be accessed by all processors. They are located in the memory of processor 0 and are referred to as "shared-only" objects. Their names do not occur in ALIGN statements. There are two PDDP storage class modifiers: `shared` and `private`. The `shared` modifier must be used in all declarations of distributed and shared-only objects. By default, nondistributed objects, or those declared using the attribute, `private`, are replicated in local memories and will be referred to as "local" objects.

To indicate execution by only one processor, the user places statements within the following construct:

```
MASTER
ENDMASTER
```

To synchronize the team of processors, the BARRIER statement is available.

PDDP recognizes the Fortran 90 WHERE construct with an optional ELSEWHERE:

```
WHERE logical-array expression
    assignment statement

ELSEWHERE
    assignment statement

ENDWHERE
```

The PDDP FORALL statement is similar to the HPF FORALL. It takes the form

FORALL (*index specifications [, scalar-mask expression]*)
assignment statement

where *index specifications* takes the form:

index-name = subscript : subscript [: stride]

FORALL may be used for a scatter operation if the expression used to designate the resulting location is entirely local. For example, in the following statement, `index(i)`, must be a local array:

```
FORALL (i = 1:100) (x(index(i)) = i**2
```

PDDP recognizes the Fortran 90 syntax for the global reduction functions `max`, `min`, `sum`, `product`, `any`, `all`. For each, it generates inline code that causes each processor to calculate its local result and store it in a shared-only array. After a barrier, processor 0 calculates the result as a global scalar. Each processor then makes a local copy. PDDP also recognizes the `cshift` function.

Following is a sample PDDP subroutine. Arguments `a` and `msk` are distributed arrays. Array `c` is local. Distributed array `x` is communicated to the subroutine through a common statement.

```
 subroutine sub(a, c, msk)
 integer nx,ny,nz
 parameter(nx=12,ny=12,nz=12)
 template t1(5,nx,ny,nz)
 distribute t1(*,*,block,block)
 template t2(nx,ny,nz)
 distribute t2(*,block,block)

 shared real*8 x(nx,ny,nz)
 shared real*8 a(5,nx,ny,nz)
 shared logical msk(nx,ny,nz)

 align x, msk with t2
 align a with t1

 real*8 c(13,5)
 common /cc/ x

 x = 0.
 forall (i=2:nx-1, j=2:ny-1, k=2:nz-1)
>             x(i,j,k) = i+j+k
 barrier

 do m = 1,5
    where (msk)
       a(m,:,:,:) = c(1,m) + c(2,m)*x
    endwhere
 end do
 barrier
 return
 end
```

A PDDP code converts easily to HPF. The programmer should preface `master`, `endmaster`, and `barrier` statements with the column 1 tag `"CPDDP$"` so that HPF will ignore them. The `shared` and `private` attributes used in declarations can easily be removed with the use of macros. Then HPF will compile the source code. Parallelism in both models occurs chiefly through the use of array syntax and the FORALL loop. Because all of the PDDP processors execute the sequential sections, code that has global or side effects (such as file accessing or the altering of global data with local data) may alter the semantics from a strict HPF interpretation. The user should place statements with such side effects within a master block. Alternatively, all sequential sections may be guarded with the master block.

4 PDDP SEMANTICS

Generated code consists of Fortran 77 statements. PDDP translates each distribution and TEMPLATE declaration into a call to a library routine that assigns to the distribution an ID tag and writes a local table of the necessary information. Distributed arrays must be dynamically allocated on the local heap. Shared-only arrays used as subroutine arguments or in common statements must also be allocated. For each distributed array, PDDP translates the appropriate ALLOCATE statement executed by each processor into calls to library routines. These routines give the array an ID tag, allocate the appropriate amount of local memory, and build a local database linking the array to its distribution information and to an address map. The address map gives the starting address of the memory allocated in each processor's memory. The processors use these addresses for requesting remote data (see Table 1).

PDDP codes use the "owner-computes" rule for parallel execution of assignment statements: The owner of the left-hand side element executes the assignment for that element. PDDP initially assumes that the right-hand side is remote; however, it will not issue a `get` on distributed memory machines if the processor number of the requested

Table 1. Array Database

Array Data		
ARRAY TAG		
DISTRIBUTION TAG		
Number bytes/element		
Rank		
Global bounds		
Local bounds		
ADDRESS MAP PTR		
	Address Map	
Distribution	Proc	Address
DISTRIB TAG	0	0x000200
rank	1	0x000120
extent/dim	2	0x000040
distrib type/dim	3	0x000220
no. procs/dim	4	0x000120
	. . .	. . .

address is the same as the requesting processor. Generated declarations include the pointers and variables needed by PDDP to express a Fortran 90 array statement as do loops whose bounds are the indices for the local portion of the left-hand side array. There is no restriction to the number of seven possible dimensions that can be distributed or the extent of any distributed dimension. For multidimensional left-hand side arrays, the do loops are nested with the leftmost dimension being the outermost loop.

Because the left-hand side owner is determined at run-time, PDDP allows dynamic array sizes and varying numbers of processors. For a left-hand side scalar reference to a distributed object, PDDP simply inserts a call to routines that determine the owning processor. Only the owning processor executes the statement. Note that this is substantially different from a scalar reference to a nondistributed data item. In the first case, the statement is executed via owner-computes. In the latter case, all team members execute the statement. The user must be careful with statements that contain data dependency between left- and right-hand sides. For example, in order to achieve the Fortran 90 implied results for the statement, `A(2:10) = A(1:9)`, the array should first be stored in a buffer from which the right-hand side values are taken.

Subroutine linkage in PDDP ensures consistency across subroutine boundaries. With the exception of local routines (see Section 5), array slices are not allowed as arguments in subroutine calls. To pass entire distributed arrays to other modules, PDDP recognizes the use of whole array syntax used in subroutine calls or in common statements. The called subroutine must align a distributed argument to a template with the same distribution as specified in the calling routine. Automatic redistribution on subroutine entry is not supported. Rather than sending a valid address as the argument to a routine, PDDP actually passes the ID tag associated with the array. (The tag is created in the allocation process.) Similarly, it is the ID tag that is actually used in a common block. In the receiving routine, the ID tag allows the module to access information on the data object by using the run-time support routines (see Section 5). The tag is selected so as to cause a fault if referenced without proper declaration and query of the run-time routines. This helps to reduce the number of errors that can be made by new users.

4.1 Optimizations

Because PDDP is a source-to-source language translator, it is limited in the range of possible optimizations. It is dependent on the backend compiler optimizer for many performance improvements.

The PDDP parser recognizes matching array syntax and distribution for left- and right-hand expressions and avoids the time-consuming calculation of the owner. It also avoids divisions involving a stride of 1. If the rank of the left-hand and right-hand arrays is unequal and the extra dimensions have a degenerate distribution, the parser also omits generating code that performs calculation of the owner.

There is a tradeoff between prefetching all of the data for a loop and PDDP's fetch on demand. On machines with a quick remote memory access such as the Cray T3D, the one-word fetch may prove to be faster because it does not periodically overload the network and there is no false data fetching. The method would certainly be superior if the fetching were overlapped with calculations.

5 RUN-TIME LIBRARY

As Nitzberg and Lo [5] point out, a useful distributed shared memory system must automatically transform shared memory access into interprocess communication. To achieve this, it is necessary for each processor to have knowledge of the mappings of the distributed arrays so that nonlocal memory may be accessed and the owner of array elements may be determined. As mentioned above, the

PDDP parser generates calls to the run-time library routines that build and access linked tables that make up a local database. The number of processors is a run-time parameter. The data are used to determine the run-time owner, the bounds for the generated do loops, and the location of each right-hand-side distributed object in terms of processor number and offset from the starting address on that processor.

Given the global iteration set specified by the user in array syntax and the knowledge of the resident elements from the database, PDDP uses Euclid's extended algorithm [6] to calculate the intersection, a set of local loop indices for a processor. For block distribution, the run-time module takes shortcuts in calculation of the owner. The local array address map allows PDDP to express the actual assignment statement in terms of pointers and offsets, and optional processor numbers for the right-hand side.

To demonstrate the use of the database and run-time libraries, consider the following PDDP code. Different distributions are used on left- and right-hand sides to demonstrate the use of the library:

```
integer nx
real x(nx), y(nx)
template t1(nx)
distribute t1(block)
align x with t1
template t2(nx)
distribute t2(cyclic)
align y with t2

x = y
```

Below is the PDDP generated pseudo code:

```
      pointer (ptr0, local_mem)
      pointer (ptr_rh, remote_mem)
      integer address_map_x(no_procs)
      integer address_map_y(no_procs)
      ...
c     loop setup:
      ptr0 = address_map_X(myproc)
      lo_indx = 1
      hi_indx = nx
      stride  = 1
      call get_local_indx(X_id,
     >              lo_indx, hi_indx, stride)
c     lo_indx, hi_indx, stride are now local bounds
      stride_rh  = stride
      lo_indx_rh = lo_indx
      do indx1 = lo_indx, hi_indx, stride
         offset     = mod(indx1, no_procs)
         proc_no_rh = mod(lo_indx_rh, no_procs)
         offset_rh  =  div(lo_indx_rh, no_procs)
         ptr_rh     =  address_map_Y(proc_no_rh)
c     assignment statement:
         local_mem(offset) =
     >     get(proc_no_rh, remote_mem(off_set_rh))
         lo_indx_rh = lo_indx_rh + stride_rh
      enddo
```

Note that if the two arrays had the same distribution, the generated code would be:

```
      ...
c     loop setup:
      ptr0    = address_map_X(myproc)
      ptr_rh  = address_map_Y(myproc)
      lo_index = 1
      hi_indx = nx
      stride  = 1
      call get_local_indx(X_id,
     >              lo_indx, hi_indx, stride)
c     lo_indx, hi_indx, stride are now local bounds
      do indx1 = lo_indx, hi_indx, stride
         offset  = mod(indx1, no_procs)
c     assignment statement:
         local_mem(offset) = remote_mem(off_set)
      enddo
```

In addition to supplying routines that are called by the generated code to calculate the owner, the run-time library supplies routines for the user and debugging tools to query the database. Inquiry functions give the rank and global and local bounds of a distributed array as well as the size in terms of the number of elements of the local block of memory, and the starting address of the local block of memory.

One of the library routines gives the starting address and size of the local array block and thus allows the user to pass the local array section to local routines. Other routines supply the processor number and total number of processors.

6 I/O

PDDP does not offer parallel input/output. Write and read statements must be placed within `master, endmaster` blocks, and the variables used must either be local or shared only (i.e., not distributed). This is obviously awkward and a definite weakness in most high-level parallel programming languages.

7 USER INTERFACE

PDDP accepts files with the suffix `.pddp`, as well as `.PDDP`, `.F`, `.f`, and `.o`. It passes options other

than those directed to the parser on to the compiler and loader [7]. For example:

```
pddp -o code.x -g obj.o code.pddp
```

In the above example, PDDP translates the file `code.pddp` into `code.f`, which is passed to the Fortran 77 compiler along with the option `-g`. Then PDDP passes the resulting `code.o` along with `obj.o` to the loader. The option `-barrier` may be used to place a barrier after each array syntax statement translation to test for race conditions. This puts PDDP into a SIMD-like mode for array operations only.

Use of the `-nodist` option causes PDDP to ignore data distributions statements, substituting shared memory declarations. The resulting code is a shared memory program that can be used for timing and debugging.

Debuggers can display the generated Fortran 77 code or, in the case that the native compiler recognizes lines beginning with "#[line]", the debugger can display the original user code. This was advantageous for PDDP users on the BBN TC2000. They were able to use the Totalview X-window debugger to easily debug their PDDP codes. In either case, the run-time library provides debugging functions to display the values of a distributed array, array slice, or designated array element. Indices, bounds, and resident processor may also be printed. To see the memory configuration of a given distributed array, `pddp_config` displays the processor number, local lower and upper bounds, stride, and distribution type of the entire array.

8 PORTABILITY

One of the most important characteristics of PDDP is its portability. It is designed to generate code for any parallel computer with shared or distributed memory that has the capability, either in hardware or software, for one processor to request and receive data located in another processor's memory.

When porting PDDP to the various platforms, we had to consider several issues besides the major one of internodal communication. These included the peculiarities of the native Fortran 77 compiler. For example, cf77 does not allow "#[line]" line directives.

For shared memory machines, we had to decide how to implement distributed memory, and on those machines with only distributed memory we had to decide how to implement shared-only memory. Because all of the processors execute the entire code, we had to arrange for all of the processors to be forked and ready to execute the first statement.

8.1 Platforms

On architectures with hardware support for remote memory references, such as the BBN TC2000 and the CRI T3D, the task of writing a compiler for the data parallel programming model is greatly simplified. With the owner-computes rule in effect, the processor that handles the computation for a section of an array receives the remote data that it needs through the use of remote memory reference support. The nature of the compiler is that of a finite-state engine that handles all of the actions for the processor that is performing the work. To perform efficiently on other architectures, PDDP uses the fastest available means of communication to obtain remote data.

PDDP was initially developed on the BBN TC2000, a computer with distributed but globally addressable memory. PDDP currently is available on the CRI T3D, the Meiko CS-2, and the SGI Power Challenge.

Each BBN processor had a 12-Mbyte low-latency "local" memory and thus resembled a distributed memory architecture. Each processor also contributed 4 Mbytes to an interleaved shared memory wherein successive cache lines were placed on successive processors and wrapped around. Because there was a single address space, it also resembled shared memory. The hardware handled nonlocal accesses, so there was no need for explicit message passing. On the BBN, a run-time library module called "niam" started first; this routine forked the necessary processors and then called the user's main program. When the main program returned, niam terminated the other processors and then itself exited.

On the T3D and Meiko CS-2, the system takes care of starting up all of the requested processors. On these two platforms, processor 0 serves as the resident of shared-only objects. On the Meiko this is much less efficient than the interleaved shared memory on the BBN.

Each node on the Meiko has 128 Mbyte of memory. The Meiko has a 70-MHz multistage fat tree interconnect, an Elite network switch, and an Elan communications processor. The Elite switch is an eight-way crossbar switch allowing input/output pairing without contention. Usable bandwidth is 50 Mbyte/s/link in each direction. To read remote

data, PDDP uses `fetch` from the Elan Widget Library. The Elan Widget communications library views the address spaces of processors as distributed global memory and explicitly addresses nonlocal memory by network DMA operations.

Memory on the CRI T3D is globally accessible and physically distributed, 64 Mbyte per processor. Remote memory referencing is done with a replicated virtual memory address space and separate tracking of processor indices. The 128 processors of the T3D are linked with a 3D torus communications network capable of low-latency data transfers of over 140 Mbyte/s node to node. Peak per processor performance is 150 Mflop. In a manner similar to PDDP, the CRI data parallel programming model, CRAFT [8], allows the user to view the distributed memory as logically shared and sets the default storage type to private. However, CRAFT restricts the user to powers of two in the distributed dimensions. On the T3D, PDDP allocates memory on the shared memory heap and uses `shmem_get` from the SHMEM library to access right-hand side data. `shmem_get` does a blocking transfer of data from the remote address into the local address using remote loads. It would be advantageous to do a `put` instead, but that is not compatible with the owner-computes rule. To avoid segmentation violation errors when accessing remote addresses on the T3D, we allocate the same amount of memory on each processor for a distributed array regardless of whether it is used.

Although the PDDP model is directed to nonuniform access distributed memory architectures, it can also be used on computers with a single shared memory. PDDP was ported to the SGI computer to provide a developmental platform for massively parallel computer users. On the SGI, it forks the desired number of processors, which executes the code as a team. It ignores the shared and private attributes and translates the use of Fortran 90 syntax, `FORALL`, and `WHERE` statements, into `do` loops in which the indices are interleaved among the processors in a wrap around manner.

8.2 Performance

To demonstrate the performance of PDDP, we present results obtained from four codes in our benchmarking suite (see Tables 2, 3, 4, and 5). The Gaussian nonpivoting elimination solver uses a `CYCLIC` distribution for the second dimension of the matrix. The highly parallel shallow water code is a two-dimensional finite difference algorithm on a 512 × 512 grid. The second dimension is distributed in a blockwise manner across the processors. For the Gaussian and shallow water codes, we include times from the CRI CRAFT model and the Portland Group HPF, version 1.1-1. We also include PVM numbers on the CRAY T3D for the Gaussian code, the smallest of the group.

Table 2. Gaussian Elimination Algorithm (Nonpivoting) 1,024 × 1,024

	Time (s)				
N*	T3D PDDP	T3D CRAFT	T3D PVM	MEIKO PDDP	MEIKO PGHPF
16	48	22	17	85	678
32	32	17	12	173	1,160
64	26	16	12	353	2,780
128	23	19	12		

* Number of processors.

LU, the NAS benchmark implicit PDE solver for five coupled, nonlinear partial differential equations, uses a `BLOCK` distribution in the last two dimensions. The data in the quantum lattice gauge (QLG) code are four and six-dimensional arrays of complex variables representing 3 × 3 arrays in four-dimensional space. The arrays are distributed `BLOCK`, `BLOCK`, `BLOCK` in the three right-most dimensions. A large portion of the calculation is the multiplication of 3 × 3 matrices. The early PGHPF and Craft compilers were unable to handle our versions of the LU and QLG codes.

On platforms that do not efficiently support remote memory referencing, e.g., the Meiko, latency of short messages can be a limiting factor. The read bandwidth on the T3D is 2 ns versus 30 ns on the Meiko. On a platform such as the Meiko, if one cannot repackage communications into long messages and transmit them prior to need, perfor-

Table 3. Shallow Water (512 × 512) 50 Iterations

	Time (s)			
N*	T3D PDDP	T3D CRAFT	MEIKO PDDP	MEIKO PGHPF
16	30.0	9.3	82	13
32	16.7	4.8	64	11
64	10.2	2.5	54	15
128	7.0	1.4	47	

* Number of processors.

Table 4. LU (64 × 64)

N*	Time (s) T3D PDDP	MEIKO PDDP
16	4.480	9.285
32	2.367	5.811
64	1.235	3.635
128	773	2.646
256	421	

* Number of processors.

mance suffers. To date, this prefetch has been a task treated by hand in programs using the message-passing programming model. In the case of high-level languages, it would be advantageous to accomplish this transparently under control of the compiler. William Carlson from SRC has recently developed an AC compiler [9] for the T3D that does a prefetch and shows good results.

Under the control of PDDP, processors act as a vector unit for the duration of the loop and consequently would greatly benefit from having the performance characteristics of a conventional vector processor.

9 CONCLUSIONS

It is evident from our numbers that some form of hardware support for accessing remote memory is necessary for PDDP codes to run well. This may also be necessary to achieve good performance from high-level parallel programming languages in general. If this support is not present, then some form of a prefetch mechanism is necessary. PDDP is unique in its utilization of hardware support for accessing remote addresses. Implementation of a shared memory programming style itself has proven to be a fundamental feature of massively parallel programming environments. Vendors are striving to place this functionality in the hardware itself.

Table 5. Quantum Lattice Gauge Code 1 Loop for 2,048 Elements

N*	Time (s) T3D PDDP	MEIKO PDDP
2	150.6	194.0
4	79.5	139.0
8	43.6	114.8
16	24.3	89.8
32	21.	130.
64	11.	123.

* Number of processors.

In evaluating a model such as PDDP, we need to consider the effort required to write a code in a language such as PDDP and compare it to that of porting a code written in message passing, analyzing its performance on the target architecture, and tuning it in some cases via assembly language to obtain reasonable performance. These latter tasks take considerable time and effort and require in-depth knowledge of the target architecture.

A reasonable fraction of this performance can be achieved by using a high-level programming model such as PDDP. While the code does not perform as well as vendor-specialized software, scientists prefer the portability trade-off gained. PDDP users can attain reasonable performance with considerably less work than is required today on massively parallel systems. In addition, portability gives application programmers the benefit of single-source maintenance.

PDDP is a research vehicle and a simple language. Nevertheless, we have shown that it is possible to program codes for parallel computers in a high-level language, avoiding the complexities of message passing and achieving satisfactory performance with one source code on multiple parallel platforms.

ACKNOWLEDGMENTS

Other contributors to the PDDP project have been Brent Gorda, Andrew Ingalls, James Stichnoth, Alan Riddle, Bor Chan, and Paul Lu. Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

REFERENCES

[1] K. H. Warren, B. Gorda, and E. D. Brooks III, "Programming in PFP," Lawrence Livermore National Laboratory, Livermore, CA, Tech. Rep. UCRL-MA-107028, 1991.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D Language specification," Department of Computer Science, Rice University, Tech. Rep. TR90-141, Dec. 1990.

[3] High Performance Fortran Forum. "High performance Fortran language specification." Rice University. Houston. TX. Version 1.1. November 10, 1994.

[4] ISO. "Fortran 90." May 1991. [ISO/IEC 1539: 1991 (E)].

[5] B. Nitzberg and V. Lo. "Distributed shared memory: A survey of issues and algorithm." *Computer*, pp. 52–60. Aug. 1991.

[6] D. E. Knuth. "The art of computer programming," in *Fundamental Algorithms*, vol. 1, R. S. Varga and M. A. Harrison, Eds. Reading, MA: Addison-Wesley, 1973, pp. 14–17.

[7] K. Warren. "PDDP: A parallel data distribution preprocessor." Lawrence Livermore National Laboratory. Livermore. CA. pp. 42–51 in MPCI Yearly Report 1992: Harnessing the Killer Micro, Tech. Rep. UCRL-ID-107022-1992.

[8] Cray Research Inc., *Cray MPP Fortran Reference Manual*. Cray Research. Inc. SR-2504 6.1, 1994.

[9] W. Carlson and J. Draper. *Distributed Data Access in AC*. Bowie, MD: IDA Supercomputing Research Center. December 14, 1994.