PDS: DIRECT SEARCH METHODS FOR UNCONSTRAINED OPTIMIZATION ON EITHER SEQUENTIAL OR PARALLEL MACHINES*

VIRGINIA TORCZON

RICE UNIVERSITY†

Abstract. PDS is a collection of Fortran subroutines for solving unconstrained nonlinear optimization problems using direct search methods. The software is written so that execution on sequential machines is straightforward while execution on Intel distributed memory machines, such as the iPSC/2, the iPSC/860 or the Touchstone Delta, can be accomplished simply by including a few well-defined routines containing calls to Intel-specific Fortran libraries. Those interested in using the methods on other distributed memory machines, even something as basic as a network of workstations or personal computers, need only modify these few subroutines to handle the global communication requirements. Furthermore, since the parallelism is clearly defined at the "do-loop" level, it is a simple matter to insert compiler directives that allow for execution on shared memory parallel machines. Included here is an example of such directives, contained in comment statements, for execution on a Sequent Symmetry S81.

PDS encompasses an entire class of general-purpose optimization methods which require only that the user provide a subroutine to evaluate the function. These methods require even less of the problem to be solved since direct search methods presume only that the function is continuous. Thus, parallel direct search methods are particularly effective on parameter estimation problems involving a relatively small number of parameters. They are also very interesting as parallel algorithms because they are perfectly scalable: they can use any number of processors regardless of the dimension of the problem to be solved and, in fact, tend to perform better as more processors are added.

Key words. nonlinear optimization, unconstrained optimization, multidirectional search, parallel direct search, parallel computing

1. Introduction.

The parallel direct search algorithms are designed to solve the unconstrained minimization problem

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}),$$

where $f : \mathbb{R}^n \to \mathbb{R}$. What distinguishes the direct search methods from other optimization methods is that they require only that the function f be continuous.

Direct search methods neither require nor estimate derivatives. As a consequence, while they are usually slower to converge than derivative based methods, they are usually much more robust in situations where the function values are subject to noise, analytic derivatives are unavailable, or finite difference approximations to the gradient

* This work was sponsored by Air Force Office of Scientific Research. Use of the Sequent Symmetry S81 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CDA–8619893. Use of the Intel iPSC/860 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement Nos. CCR–8809615 and CDA–8619893 with support from the Keck Foundation. This research was performed in part using the Intel Touchstone Delta System operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CCR–8809615.

† Department of Mathematical Sciences, Houston, Texas 77251–1892.

are unreliable. Furthermore, the direct search schemes given here parallelize very well, although they can certainly be used as sequential methods.

The routines included here were designed to be run on sequential machines as well as on any of Intel's distributed memory parallel machines, including the iPSC/2, the iPSC/860, and the Intel Touchstone Delta, with every expectation that they should run on the Intel Paragon XP/S supercomputer as well. Furthermore, the code is structured in such a way that the parallelism can be observed at the level of a "do-loop"; thus straightforward implementation of this code on any shared memory machine should be possible. In fact, the code contains compiler directives, within comment statements, to allow for parallelization on the Sequent Symmetry S81.

All that is required of the user is a subroutine FCN to compute the value of f for a given $\mathbf{x}$, as well as an INPUT file to define the initial starting conditions for the problem to be solved.

The next section gives a brief description of the parallel direct search methods. Section 3 explains what must be done to use the code included here. Section 4 discusses basic implementation issues so that the order of compilation will be clear. Section 5 describes the test problem included here as an example. The last section discusses the Intel-specific library calls that are used within the subroutines provided here for Intel distributed memory machines so that those interested in using PDS in other distributed memory computing environments can get some feel for the modifications that will need to be made.

2. The Parallel Direct Search Algorithms. A detailed development and description of the parallel direct search algorithms can be found in [1]. Convergence results for the multidirectional search algorithm, the direct search method upon which these parallel schemes are based, can be found in [8]. We briefly summarize the parallel direct search schemes here.

The multidirectional search algorithm is a simplex based algorithm; thus the search begins with a nondegenerate simplex S_0 with vertices $\langle \mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_n \rangle$. The user is required to provide an initial estimate of the solution. This estimate is used as one of the vertices in the initial simplex. (The remaining vertices may be entered by the user, but we have also included subroutines to construct a nondegenerate starting simplex.) We order the vertices of the initial simplex S_0 so that $f(\mathbf{v}_0) \leq f(\mathbf{v}_i)$, $i = 1, \cdots, n$. We then designate $\mathbf{v}_0$ as the "best" vertex.

The idea of the parallel direct search schemes is quite simple: given an initial simplex S_0, a best vertex $\mathbf{v}_0$, a search scheme, and the *s*ize of the *s*earch *s*cheme (sss) to be employed, we first test for convergence and then compute sss points and their function values. We choose the point with the lowest function value, update the simplex and go on to the next iteration. The basic algorithm is given in Table 1.

This seemingly simple idea has a surprisingly strong convergence result. If we assume that the function f is continuously differentiable over a compact set $L(\mathbf{v}_0) = \{\mathbf{x} : f(\mathbf{x}) \leq f(\mathbf{v}_0)\}$, then we can prove that some subsequence of the best vertices converges to a stationary point $\mathbf{x}_*$ of f, $\mathbf{x}_* \in L(\mathbf{v}_0)$ and thus the sequence of best vertices converges to the set $\{\mathbf{x} : f(\mathbf{x}) = f(\mathbf{x}_*),\ \mathbf{x} \in L(\mathbf{v}_0)\}$. This result can be extended to

Given an initial simplex S_0 with vertices $\langle \mathbf{v}_0, \mathbf{v}_1, \cdots, \mathbf{v}_n \rangle$,
initialize the search scheme
while (stopping criterion is not satisfied) **do**
  for $i = 1, \cdots, sss$ **do**
    $\mathbf{v}_i \leftarrow \mathbf{v}_0 + a_{1i}(\mathbf{v}_1 - \mathbf{v}_0) + \cdots + a_{ni}(\mathbf{v}_n - \mathbf{v}_0)$
    $fv_i \leftarrow f(\mathbf{v}_i)$
  end
  $fv_* \leftarrow \min_i \{fv_i\}$ /* communication */
  update simplex
end

TABLE 1
The parallel direct search algorithm.

include functions that are continuous on $L(\mathbf{v}_0)$, with convergence then to the set containing all stationary points of f on $L(\mathbf{v}_0)$, all points where f is nondifferentiable on $L(\mathbf{v}_0)$, and all points in $L(\mathbf{v}_0)$ at which the gradient exists but is not continuous [8].

The parallel direct search schemes were designed to be used on parallel machines, hence their name, but a simple glance at Table 1 should make it clear that there is nothing to prevent these methods from being run in a standard sequential computing environment. However, the parallelism that can be exploited is also obvious. If we assume that the data, in the form of the scalars $a_{i1}, \cdots, a_{in}$ needed to compute the vertex $\mathbf{v}_i$ are available to processor i, we then have a single program, multiple data (SPMD) model for parallel computation. The only point in the computation at which synchronization, or, in the case of distributed memory machines, communication is required is in the choice of fv_*. Thus the routines for running the optimization on either a sequential machine or an Intel distributed memory machine are *identical* except for a single call to the subroutine GLOBAL, which is required to handle the global communication on the distributed memory machines.

Note that sss, the size of the search scheme, need not depend on the number of processors available since it is possible to compute multiple points on each processor before undertaking the global synchronization/communication. This can be seen in Table 2, where p is the number of processors and the k^i are chosen so that $\sum_{i=1}^{p} k^i = sss$. Thus in the base case, $p = 1$ and $k^1 = sss$—giving us a sequential algorithm.

2.1. The Search Scheme. To divorce the number of processors that can be used from the dimension of the problem to be solved, we rely on search schemes that take the basic multidirectional search algorithm introduced in [7] and look ahead to subsequent iterations of the algorithm. Because the multidirectional search algorithm is essentially a sampling method, the moves allowed are limited but predictable. As a consequence, "look-aheads" are possible and unlimited. This simple strategy leads to extremely flexible, scalable algorithms ideally suited to parallel computation.

Defining the search scheme consists of defining a pattern or template of points,

```
Given an initial simplex S_0 with vertices ⟨v_0, v_1, ···, v_n⟩,
    initialize the search scheme
    while (stopping criterion is not satisfied) do
        for i = 1, ···, p do
            for j = 1, ···, k^i do
                v^i_j ← v_0 + a^i_{1j}(v_1 − v_0) + ··· + a^i_{nj}(v_n − v_0)
                fv^i_j ← f(v^i_j)
            end
            fv_i ← min_j {fv^i_j}
        end
        fv_* ← min_i {fv_i}                    /* communication */
        update simplex
    end
```

TABLE 2
The "chunked" parallel direct search algorithm.

relative to the current simplex and its best vertex, to be constructed at each iteration. In other words, we must initialize the scalars $a^i_{1j}, \cdots, a^i_{nj}$ that reside on each processor. Once this has been done, the vertices $\mathbf{v}_0, \cdots, \mathbf{v}_n$ in the simplex change from iteration to iteration, but the scalars defining the template do not. Thus, while a routine for producing a template based on the original multidirectional search algorithm is provided, the optimization routines assume that this information resides in a file and simply read in the necessary information *once*, as part of the initialization, before the optimization begins. The derivation of the template, as well as the algorithm actually used to construct it, are described in some detail in [1].

3. Usage. Use of the subroutines and drivers for the parallel direct search algorithms will be detailed in this section. We begin with a minimal description of what is necessary to run the parallel direct search schemes using the drivers included here.

The user must provide:

1. An input file INPUT as shown in Table 3. Further explanations can be found in §3.3.
2. A subroutine FCN(N,X,F) to evaluate the function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ at a point $\mathbf{x}$. The parameters of FCN are as follows:

N	integer	dimension of the problem
X(N)	double precision	vector at which the function is to be evaluated
F	double precision	the value of f at $\mathbf{x}$.

FCN must be declared EXTERNAL in the calling program. Neither N nor X may be modified.

Any additional information required by FCN must be passed in COMMON.

The user must compile and link two programs:

1. CREATE

2. OPTIM

The files necessary to successfully link each program can be seen from the call graphs given in Figs. 3, 4, and 5, which are described in more detail in §4. The legend for these call graphs is given in Fig. 2.

The user then first executes CREATE, to create a file containing a search scheme, and then OPTIM, to perform the actual optimization. The result is written out to the file RESULT.

3.1. Running on an Intel Distributed Memory Machine. We assume here that the user has access to the user's guide for the machine of interest and knows the appropriate sequence of system calls to compile Fortran programs, allocate nodes and then load and execute programs.

The main routine for the optimization, PDS, is identical to the sequential version *except* for the call to the subroutine GLOBAL, which performs the global communication call. Before compilation, the "CINTEL" should be removed from the line containing the call to GLOBAL.

Programs compiled for the Intel distributed memory machines will need to set the appropriate switch to link in the communication libraries provided for those machines. Note that there is no host program. To allow these routines to be used on the Intel Touchstone Delta, which does not have a host machine, all I/O and execution occurs on the nodes.

All that need be done to execute these programs is to load them on the node(s). Since CREATE is a sequential process, it should be loaded on a single node.[1] The parallelism resides in OPTIM, which must be loaded on every node participating in the optimization, as required by the global communication calls.

Opening and closing files is left up to the user and is handled in the drivers. This is particularly important for those using either the Intel iPSC/2 or the Intel iPSC/860 since the time to access files can vary dramatically depending on whether or not a file resides on the file system specifically designed to support the nodes of the hypercube. In general, it is best to have all the necessary files residing on this file system (i.e., on the "CFS").

3.2. Running on a Sequent Symmetry. Again, we assume that the user has access to the user's guide for the Sequent Symmetry and knows the appropriate sequence of system calls to compile Fortran programs, allocate processors and then load and execute programs.

The parallelization occurs in a single loop in the main subroutine for the optimization, as should be clear from Table 1. The appropriate compiler directives have been included in the comment statements for the subroutine PDS, which contains this loop. To activate these compiler directives, the user will need to remove the "CSEQUENT "

[1] It is even possible to generate the search scheme on other machines with larger memory, for instance a Sun Workstation, but the byte order of the unformatted output must be consistent with the byte order required by the Intel chips, which requires some C programming.

from the beginning of the four lines containing the directives before compiling the subroutine.

In addition, the user will need to make sure that the variables in FCN, the subroutine to compute the function value, are protected during parallel execution. To do this, insert a "C$ PRIVATE" line after the declaration of the variables. This is demonstrated in the test function included here and can be used, again by removing "CSEQUENT " from the beginning of the line before compiling the subroutine.

We close with the observation that this code was initially designed to be used on a distributed memory machine. The Sequent compiler directives for handling parallel computation make it very easy to use this version of PDS, in parallel, on the SEQUENT. However, the need to "lock" a critical section of the loop that is being parallelized is not very satisfying and may impede performance for some applications. To improve performance, the creation of extra global workspace to allow comparison across processors *after* the loop has finished execution—as is done in the version for the Intel distributed memory machines—is advisable.

3.3. Input. To run the parallel direct search methods the user must provide a formatted input file as specified in Table 3. Descriptions of the variables are given in Table 4. Two of these are standard for all optimization algorithms: N, the dimension of the problem to be solved, and VO, an initial guess at the solution. The choice of SSS follows from the discussion found in §2.1; we add that the convergence result for multidirectional search, the core algorithm upon which the parallel direct search schemes are based, requires a minimum of $2n$ function values per iteration to guarantee convergence [8]. The rest of the input variables are described below.

Line		Variable	Fortran Type	# of Entries
1	required	N	integer	1
2	required	STEPTOL	floating point	1
3	required	MAXITR	integer	1
4	required	V0	floating point	n
5	required	TYPE	integer	1
[6]	[depends on TYPE]	SCALE	floating point	1
$[6+n-1]$	[depends on TYPE]	SIMPLEX	floating point	n^2 (n/line)
7[$+n$]	required	DEBUG	integer	1
8[$+n$]	required	SSS	integer	1

TABLE 3
Format of the input.

3.3.1. The stopping conditions. The optimization stops when either one of two criteria are satisfied: the maximum number of iterations specified by the user has been reached or a step length test has been satisfied. PDS uses the test

$$\frac{1}{\delta} \max_{0\leq i<j\leq n} \|\mathbf{v}_j - \mathbf{v}_i\|_2 ,$$

Variable	Description
N	dimension of the problem to be solved
STEPTOL	stopping tolerance for the relative step length
MAXITR	maximum number of iterations allowed
V0	the starting point for the search
TYPE	the type of simplex to be used
SCALE	the base length of the edges if the simplex is to be constructed
SIMPLEX	the remaining n vertices if the simplex is to be provided by the user
DEBUG	the level of information to be logged to a debugging file
SSS	the size of the search scheme

TABLE 4
Description of the input variables.

where

$$\delta = \max\left(1, \|\mathbf{v}_0\|\right).$$

Note that it is *not* necessary to compute the length of every edge in the simplex at each iteration; the length of the longest edge in the initial simplex S_0 is determined before the optimization begins and is updated at each iteration using a single scalar multiply. This test is a slight modification of a test proposed in [9]. Its main advantage in the parallel setting is that each processor can test for convergence independently without any need for further synchronization. Its main limitation is that it considers only the relative length of the steps being taken; no measure of relative decrease in the function values is considered.

3.3.2. The initial simplex. The user is allowed either to enter the entire initial simplex or to have one of three types of simplices constructed to start the search. The options available are given in Table 5. If the user chooses to enter the entire simplex,

Flag	Option
0	user defined
1	right-angled
2	regular
3	scaled right-angled

TABLE 5
Options available for the type of simplex.

the entries for the remaining n vertices must be entered, one per line (the starting point for the search is included as one of the $n+1$ vertices in the initial simplex). Otherwise, a scale factor must be entered as a base length (and orientation) for the edges in the simplex to be constructed.

The choice of initial simplices that can be constructed is best demonstrated for the two dimensional case as seen in Fig. 1. Note that for a right-angled simplex, the n edges adjacent to $\mathbf{v}_0$ are of length SCALE. For a regular simplex, every edge in the simplex has the same length, SCALE. The algorithm for constructing a regular simplex is taken from [2] but appears also in [6]. Both right-angled and regular simplices are standard choices for simplex based algorithms for solving unconstrained optimization problems.

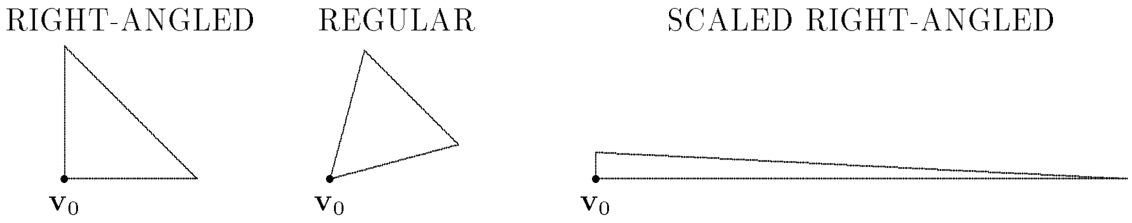

FIG. 1. *Simplex choices.*

We have also included an option for a scaled right-angled simplex, where the scaling is based on the individual components of V0. This is a less common and, quite frankly, a less desirable choice. We have included this option for those problems where the variables are poorly scaled so that there is some hope of making progress in all the variables. (A relatively mild example, for which the ratio of one coordinate to the other is 20 to 1, can be seen in Fig. 1.)

To generate a scaled right-angled simplex, a right-angled simplex is constructed and then the edges adjacent to $\mathbf{v}_0$ are rescaled by the nonzero components of $\mathbf{v}_0$, the assumption being that any knowledge the user has about the relative scale of the variables at the solution is included in the choice of an initial guess. However, as the picture amply demonstrates, this is not the most desirable option; it would be far better, if possible, to reformulate the problem so that the variables are of approximately the same magnitude.

3.3.3. The levels of intermediate output. Users may also specify several levels of "debugging" output if they wish to monitor the progress of the parallel direct search methods more closely. These options are given in Table 6.

Flag	Option
0	no debugging output
1	log the iteration count, the best vertex and its function value
2	include the simplex and flag whether or not strict decrease was obtained
3	include all vertices constructed and their function values
> 3	include the points that define the search scheme

TABLE 6
Options available for debugging.

One word of warning to those using this code, as written, on the Sequent Symmetry S81: the Symmetry does *not* support I/O in a loop that has been parallelized. Thus, on the Symmetry, if parallelization has been specified, only debugging levels 0, 1, or 2 should be used.

3.3.4. Restrictions. We close by noting in Table 7 any restrictions on the values of the input but we hasten to add the following caveat: *it is up to the user to verify that the input satisfies these restrictions.*

Variable	Restrictions
N	> 0
STEPTOL	> 0
MAXITR	> 0
V0	none
TYPE	0, 1, 2, or 3
SCALE	$\neq 0$
SIMPLEX	nondegenerate
DEBUG	≥ 0
SSS	$\geq 2n$

TABLE 7
Restrictions on the input variables.

We also extend a special caution to those who choose to enter their own starting simplex: be sure that the simplex is nondegenerate; otherwise the search will be restricted to a hyperplane that may or may not contain a stationary point of the function. One simple test for nondegeneracy is to verify that the edges adjacent to the best vertex are linearly independent before proceeding with the optimization.

3.4. Files. The opening and closing of all files is handled within the drivers for the two programs. There are at least three files involved in running these programs. If debugging information is requested, the number of files opened for this information equals the number of processors used. The files, their format, and their use by the two drivers can be seen in Table 8; descriptions of the files are given in Table 9.

File		Format	CREATE	OPTIM
INPUT	required	formatted	input	input
SCHEME	required	unformatted	output	input
RESULT	required	formatted	output	output
DEBUG###	optional	formatted	*not used*	output

TABLE 8
I/O files.

The driver to create the search scheme relies on the existence of the file INPUT to open and read in the dimension of the problem to be solved. This can be easily modified. However, the call to the subroutine SEARCH requires a unit number for the file SCHEME so that the points in the search scheme can be written out once they have been generated.

The driver to actually run the optimization passes the unit number for the file INPUT to the subroutine DEFINE, which handles the initialization of the information

File	Description
INPUT	information to define the problem and set optimization parameters
SCHEME	a search strategy for problems of a given dimension
RESULT	either the final result or an appropriate error message
DEBUG###	information specified by the choice of debugging levels

TABLE 9
File descriptions.

that must be provided by the user. The call to the subroutine GETSS requires a unit number for the file SCHEME so that the required number of points can be read in for the search scheme. The call to PDS requires a unit number for the debugging file(s) if debugging has been specified. (We append "###" to the name of the debug file(s) so that on the Intel distributed memory machines a unique file can be assigned to each processor.) Finally, the call to RESULT requires the unit number for the file RESULT.

4. Implementation. There are two sets of subroutines associated with the parallel direct search methods: one set to construct the search scheme and the other to perform the actual optimization. The two sets of subroutines can be seen in Figs. 3 and 4, respectively. We follow the convention set forth in Fig. 2.

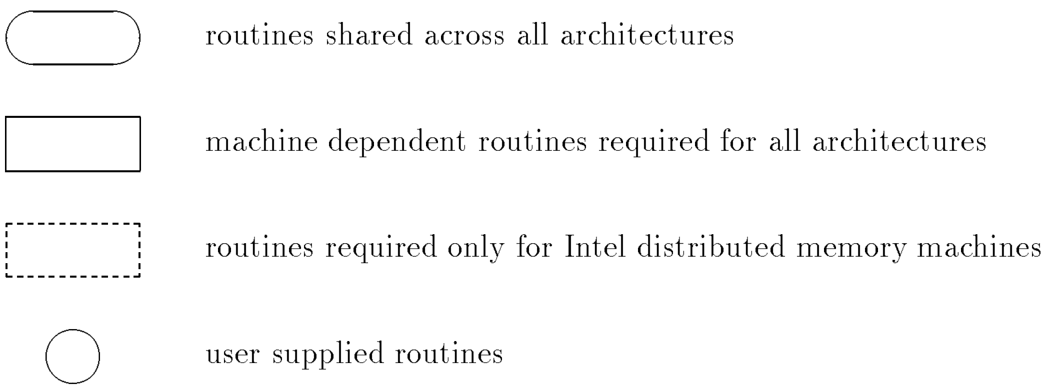

FIG. 2. *Legend for the call graphs.*

The comment block that heads each subroutine contains a description of the subroutine as well as a description of every parameter in the calling sequence.

4.1. Generating the Search Scheme. The main subroutine for generating the search scheme is SEARCH; its call graph can be seen in Fig. 3. Note that the only machine dependency involved here comes in writing the search scheme to a file, which is handled by the subroutine WRITES. For the Intel distributed memory machines we include a *separate* version of the WRITES subroutine that makes use of a special Fortran library call to handle the unformatted writes.

4.2. Parallel Direct Search. The main subroutine for the optimization is PDS; its call graph is shown in Fig. 4. The subroutine PDS assumes that the information

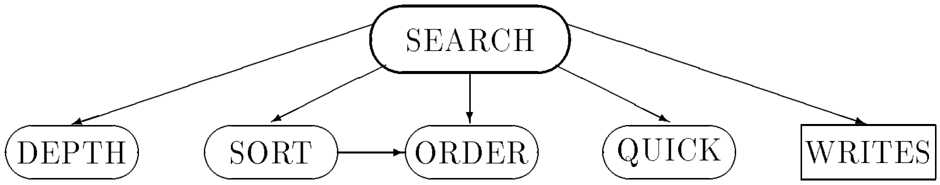

FIG. 3. *The call graph for generating the search strategy.*

that must be specified by the user has been passed, along with the points necessary for the search scheme. Thus, there is no I/O unless debugging has been specified. The auxiliary routines for initializing this information are discussed in the next section.

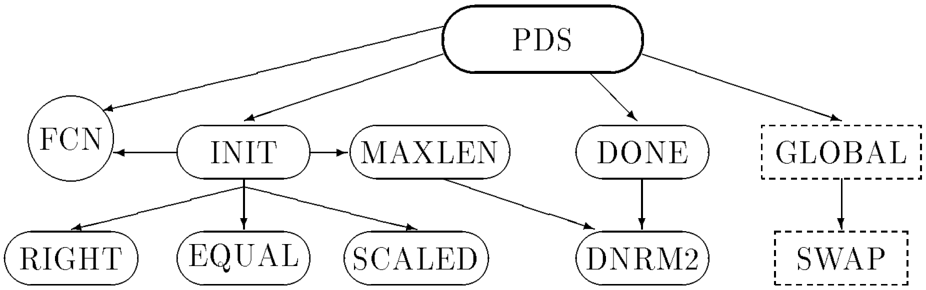

FIG. 4. *The call graph for the optimization.*

The subroutines DONE and MAXLEN make use of DNRM2, one of the basic linear algebra subprograms (BLAS) [3], to compute the Euclidean vector norm. A version has been included with this code, but users can certainly link to a local library instead.

When using the parallel direct search methods on an Intel distributed memory machine, the outcome of the search on each individual node must be exchanged with the remaining processors after the function values have been computed for every point on the node (the "/* communication */" step seen in Tables 1 and 2). This requires the subroutine GLOBAL to invoke the special purpose Fortran library calls provided to handle global communication. The function SWAP is used to effect the actual exchange. Note that these routines should *not* be linked when PDS is being used on other machines; these two routines are specific to the Intel distributed memory machines.

4.3. The Drivers. Two drivers are included for the parallel direct search schemes: the driver CREATE to create a file containing the points necessary to define a search scheme and the driver OPTIM to perform the optimization. The call graphs for each are shown in Fig. 5.

Creating the search scheme requires only a very simple driver to call SEARCH. This driver must pass the dimension n, a unit number for the file to which the points in the search scheme are to be written (all files must be opened and closed within the driver), and workspace. The only output from the call to SEARCH is an error flag to

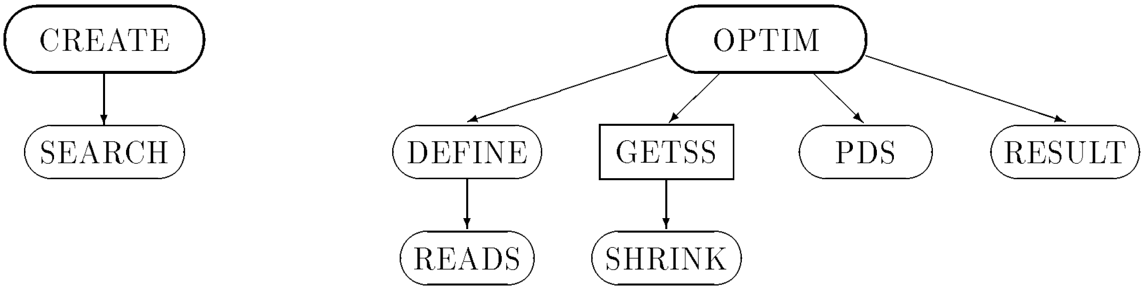

FIG. 5. *The call graphs for the two drivers.*

indicate whether or not the limit for the internal stack variable in the two Quicksort routines was exceeded. (This is documented in the comments for SORT and QUICK, but either routine can sort arrays with up to $2^{32} \sim 2 \times 10^9$ entries before the internal stack size is exceeded.)

The driver for the optimization comes with several service subroutines, of which only one is machine dependent. Since, as discussed above in §4.1, writing out the points in the search scheme to a file is machine dependent, it should not be too surprising that reading in the points in the search scheme is also machine dependent. Thus there are two versions of the subroutine GETSS. Again, the primary difference between the two versions of GETSS is that the version for the Intel distributed memory machines uses special Fortran library calls to set the file pointers (one for each processor) and to read in the unformatted data.

Both versions of GETSS read four pieces of header information from the file containing the search scheme before reading in the points for the search scheme. The first two pieces are used to make sure that the search scheme matches the specifications for the problem to be solved. In particular, we wish to ensure that the dimension used to generate the search scheme matches the dimension of the problem to be solved and that the total number of points contained in the file to be read is at least as great as the number requested by the user. If either condition is violated, an error flag is set and GETSS returns to the calling program. The remaining two pieces of information contained in the header are for algorithmic purposes and are discussed in comments contained within the appropriate subroutines.

5. Testing. Included for testing purposes is a sample driver, CREATE, to create the search scheme, a sample driver, OPTIM, to handle the optimization, a function evaluation routine FCN that evaluates the extended Rosenbrock function [4], [5], and an input file INPUT to test the two-dimensional Rosenbrock function. This is the example given in [1].[2] The entries for the file INPUT can be seen in Table 10.

When we run CREATE on either a Sun SPARCstation 1 or a Sequent Symmetry

[2] For the numbers reported in [1], we used a different stopping test so that the results could be compared for different choices of *sss*. Since the true solution was known to be $\mathbf{x}_* = (1,1)^T$ with $f(\mathbf{x}_*) = 0$, we stopped the optimization when the absolute value of the function at $\mathbf{v}_0$ fell below 10^{-7}.

Variable	File Entry
N	2
STEPTOL	1.0D-3
MAXITR	50
V0	-1.2 1.0
TYPE	2
SCALE	1.0
DEBUG	0
SSS	256

TABLE 10
Input for example

S81, we produce a file SCHEME containing 96048 bytes and the following message in the file RESULT:

```
SUCCESSFULLY COMPLETED A SEARCH STRATEGY FOR PROBLEMS OF DIMENSION    2
THE TOTAL NUMBER OF UNIQUE POINTS AVAILABLE IS                     2000
THE FACTOR NEEDED TO RESTORE THESE POINTS TO THEIR REAL VALUES IS    32
```

When we then run OPTIM, we get the following output in the file RESULT:

```
FINISHED WITH TOTAL NUMBER OF ITERATIONS:          7
THE BEST VERTEX IS:
           1.00063008120658
           1.00127939948865
WITH FUNCTION VALUE            0.00000043249716.
```

When we run CREATE on a single node of either an iPSC/860 or the Intel Touchstone Delta, we produce a file SCHEME containing 32016 bytes and the same message given above contained in the file RESULT. When we then run OPTIM, on any number of processors, we also get the same message given above in the file RESULT. Note that on the Intel machines, the special purpose routines for unformatted writes to a file produce smaller files. Note also that while the total number of processors used for the optimization may affect the execution time, it does not affect the final outcome.

6. Using PDS on Other Distributed Memory Machines. While PDS was originally designed to be run on an Intel distributed memory machine, and thus the machine dependent routines provided here make explicit use of the Fortran libraries for the Intel machines, PDS can easily be ported to other distributed memory computing environments by making the appropriate substitutions for the Intel-specific library routines. The Intel routines used here, and their descriptions, can be seen in Table 11.

We know of at least one successful port of this code to an Ncube. We also believe that it should be possible to use PDS in "parallel" without the need for special purpose parallel machines using, for instance, either a transputer board or a network of workstations with software to handle the global communication/synchronization required in the optimization.

Call	Description
cread	high-speed, synchronous read from a CFS file
cwrite	high-speed, synchronous write to a CFS file
gilow	global MIN operation used for integer scalars
gopf	make a global operation of a user-defined function (SWAP)
gisum	global sum operation used for integer scalars
lseek	move the read/write file pointer
mynode	get the node ID of the calling process
numnodes	return the number of nodes in the hypercube or partition
setiomode	set the I/O mode and perform a global synchronization operation

TABLE 11

Intel-specific Fortran calls.

Acknowledgments. I wish to thank Robert Michael Lewis for his many helpful comments regarding both the implementation and documentation of this code, for his help in getting this code up and running on the Sequent Symmetry S81, and for sharing his nonrecursive Fortran Quicksort routine. Thanks also need to be extended to the many people who served as guinea pigs on earlier versions of this code, in particular Andrea Reiff. And special thanks to Danny Soroker at Shell Development Company, Irv Lustig, and Bert Buckley for their many and considered comments on ways to structure the code so that it would be more flexible and more useful for others.

REFERENCES

[1] J. E. DENNIS, JR. AND V. TORCZON, *Direct search methods on parallel machines*, SIAM Journal on Optimization, 1 (1991), pp. 448–474.

[2] S. L. S. JACOBY, J. S. KOWALIK, AND J. T. PIZZO, *Iterative Methods for Nonlinear Optimization Problems*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[3] C. L. LAWSON, R. J. HANSON, D. R. KINCAID, AND F. T. KROGH, *Basic linear algebra subprograms for fortran usage*, ACM Transactions on Mathematical Software, 5 (1979), pp. 308–323.

[4] J. J. MORÉ, B. S. GARBOW, AND K. E. HILLSTROM, *Testing unconstrained optimization software*, ACM Transactions on Mathematical Software, 7 (1981), pp. 17–41.

[5] H. H. ROSENBROCK, *An automatic method for finding the greatest or least value of a function*, The Computer Journal, 3 (1960), pp. 175–184.

[6] W. SPENDLEY, G. R. HEXT, AND F. R. HIMSWORTH, *Sequential application of simplex designs in optimisation and evolutionary operation*, Technometrics, 4 (1962), pp. 441–461.

[7] V. TORCZON, *Multi-Directional Search: A Direct Search Algorithm for Parallel Machines*, Ph.D. thesis, Department of Mathematical Sciences, Rice University, Houston, TX, 1989; also available as Tech. Report 90-7, Department of Mathematical Sciences, Rice University, Houston, TX 77251-1892.

[8] ———, *On the convergence of the multidirectional search algorithm*, SIAM Journal on Optimization, 1 (1991), pp. 123–145.

[9] D. J. WOODS, *An Interactive Approach for Solving Multi-Objective Optimization Problems*, Ph.D. thesis, Department of Mathematical Sciences, Rice University, Houston, TX, 1985; also available as Tech. Report 85-5, Department of Mathematical Sciences, Rice University, Houston, TX 77251-1892.