SAND2000-8843
Unlimited Release
Printed August 2000
Revised January 2001

Usage Manual for
APPSPACK version 2.0*

P. D. Hough†, T. G. Kolda‡, and H. A. Patrick§
Computational Sciences and Mathematics Research Department
Sandia National Laboratories
Livermore, CA 94551-9217

ABSTRACT

Usage manual for APPSPACK, an asynchronous parallel pattern search method for optimization.

Keywords: asynchronous parallel optimization, pattern search, direct search, fault tolerance, distributed computing, cluster computing.

*This research was sponsored by the Mathematical, Information, and Computational Sciences Division at the United States Department of Energy and by Sandia National Laboratory, a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

†Email: pdhough@ca.sandia.gov.
‡Corresponding author. Email: tgkolda@ca.sandia.gov.
§Email: hapatri@sandia.gov

This page intentionally left blank.

Usage Manual for APPSPACK

P. D. Hough, T. G. Kolda, and H. A. Patrick

February 15, 2001

1 Overview

APPS implements an asynchronous and fault tolerant parallel pattern search method for optimization; see [5] for details of the original APPS algorithm. Pattern search uses *only* function values for optimization, so it can be applied to a wide variety of problems. The name *pattern search* derives from the fact that a pattern of search directions is used to drive the search. Parallelism is achieved by dividing the search directions (and corresponding function evaluations) among the different processors. The *asynchronous* part comes about as a consequence of the fact that, unlike parallel pattern search, no synchronization of the searches along each direction is required.

APPS runs a collection of *agents* to manage the searches. This differs somewhat from the original APPS algorithm described in [5] because there may be more than one search direction assigned to each agent. APPS may be run in multi- or single-agent mode. Multi-agent mode insures a high level of fault tolerance, and, in that case, one agent is assigned to each host. Single-agent mode means that every search direction is assigned to one agent. The choice of whether to use multi- or single-agent mode depends on the communication and hardware architecture, as well as on user preference. By default, APPS uses PVM as its communication architecture; however, the user may also specify MPI or a custom communication interface. Multi-agent mode is only available with PVM. Further details of the PVM and MPI implementations are discussed in §§2–3.

The basic semantics of using APPS are described in §4, and a full list of options is given in Appendix A. The output APPS generates is described in §5. Details on using APPS with a user-supplied function evaluation are given in §6.

Pattern search may revisit a single point multiple times. To avoid reevaluating the objective function at these types of points, APPS saves the results of every function evaluation in a cache. The cache is described in more detail in §7.

2 PVM Implementation

The default communication architecture for APPS is PVM [2], although an MPI [8] version (as described in the next section) is also supported and may be better suited to some environments. The primary advantage of APPS-PVM over APPS-MPI is support of fault tolerance.

In multi-agent mode, APPS-PVM runs one agent per host, and each agent is in charge of a subset of the search directions. Consider the following set of search directions:

$$\underbrace{\begin{bmatrix} +1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ +1 \end{bmatrix}, \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ -1 \end{bmatrix}}_{\text{core pattern}}, \underbrace{\begin{bmatrix} -1 \\ -1 \end{bmatrix}}_{\text{extra}}$$

The *core pattern* is a set of search directions that satisfies the necessary convergence properties for pattern search; for example, we require that the core pattern contain a positive basis in the case of unconstrained optimization or that it contain the positive and negative unit vectors in the case of bound constrained optimization. The *extra* search directions are not necessary for convergence but may accelerate the convergence process.

Figure 1 illustrates the distribution of the search directions to agents on three hosts. The distribution is governed by the *load* or *speed* of each host, that is, the ideal number of function evaluations per host. (The speed is specified in the PVM hostfile; if it is not specified, then the speed is assumed to be 1.) In this case, we assume that the speed of each host is 2. The goal of the initial distribution (or any redistribution) is threefold: First, each search direction in the core pattern *must* be assigned, even if that requires violating the ideal load of the hosts. Second, extra search directions are assigned to agents until each host has its ideal load. Third, if a search direction is already assigned to an agent, it should not be moved to a new agent unless it is necessary to maintain balance. Here, Hosts 0 and 1 each have two search directions and Host 2 has one. No agent has more that its ideal load of two.

Each agent each spawn function evaluations ("Feval" in the figure above) on its own hosts. Each function evaluation communicates only with its parent agent, and the agents communicate among themselves as well as with their children function evaluations. The communication channels are illustrated by connections between the processes.

Each time a function evaluation is required, the agent spawns a function evaluation process and sends a message to that process containing the point, x, to be evaluated. The function evaluation process evaluates f at the point x, sends a message containing $f(x)$ to the parent agent, and exits. In the event that a function evaluation fails to complete successfully, the value of $f(x)$ is assumed to be $+\infty$.

In the event that an agent fails, the search directions are redistributed. Consider Figure 1 again and suppose that Host 1 fails. In that case, we need to redistribute the search directions. The $[0, -1]^T$ direction is reassigned to Host 2,

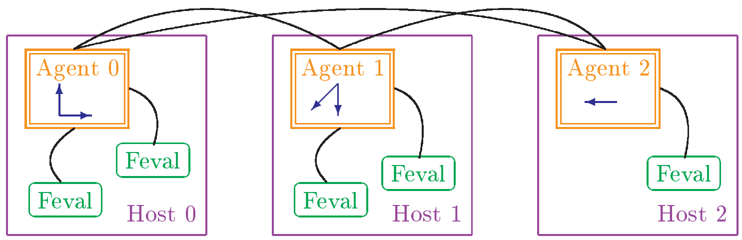

Figure 1: APPS-PVM with five search directions on three hosts.

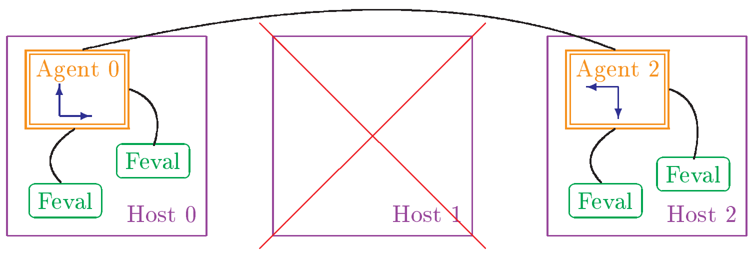

Figure 2: APPS-PVM with a failed host.

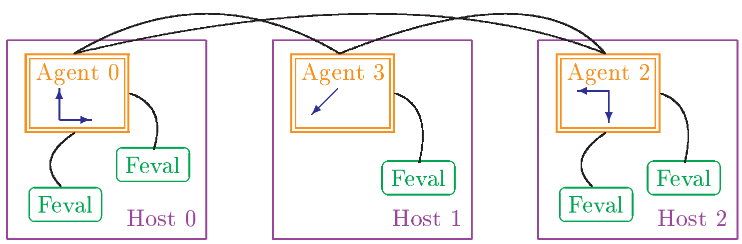

Figure 3: APPS-PVM with a restarted host.

but the $[-1,-1]^T$ direction is not reassigned because then either Host 0 or Host 2 would have more than their ideal load of 2; see Figure 2. The (re)distribution of the search directions is handled by the lowest numbered agent that is still alive (Host 0 in this case).

In the event that a host is added or restarted, a new agent is started on that host. For example, if Host 1 were restarted, a new agent would be assigned to it, and the agent would be assigned some search directions if any are available; see Figure 3. In this example, we can assign the extra search direction $[-1,-1]^T$ to the new agent without violating the ideal load constraints. Note that the agent number of 1 is *not* reused; agents are numbered consecutively as they are created.

APPS-PVM can also run in single-agent mode. In single-agent mode, one agent controls all the search directions and spawns function evaluations on all hosts. If that agent fails there is no way to restart it, so single-agent mode is not fault-tolerant. The way APPS-PVM runs in single-agent mode is very similar to the way APPS-MPI runs.

3 MPI Implementation

The MPI [8] version of APPS is quite different from the PVM version. Since the MPI-2 standard [6] does not provide some necessary functions for fault-tolerance, such as detecting the failure of processes, APPS-MPI makes no attempt at fault tolerance. It therefore runs only in single-agent mode.

The advantage of single-agent mode is that since one agent controls all the search directions, every search direction knows immediately when a new best point is found. The disadvantage is the lack of fault tolerance and the possibility that the one agent will be overwhelmed by all the extra communication it has to do as a result of being in charge of all the search directions. This has not, however, been an issue in our experiments.

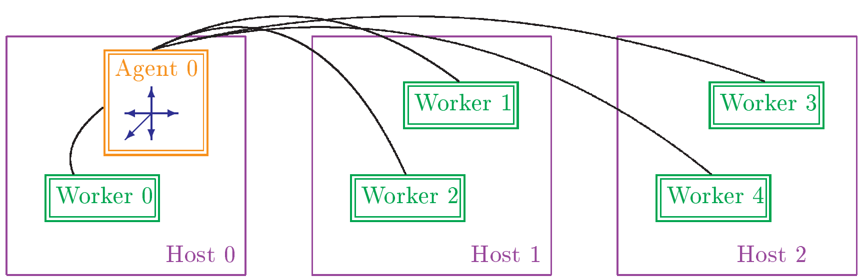

Figure 4: APPS-MPI with five search directions on three hosts.

Unlike APPS-PVM, which spawns function evaluation processes as necessary, all the processes that will be used for function evaluations by APPS-MPI come into existence when APPS starts and run until APPS terminates. All the processes start the same program. Process 0 branches to execute the code for an agent. If the cache is being used, the last process branches to execute the code for the cache; see §7. The other processes branch to the code for a *worker*; see Figure 4. Each worker waits for a message from the agent to perform a function evaluation. When it receives such a message, it evaluates the objective function at the given point and returns the result to the agent.

The user should note that the distribution of processes across hosts is controlled by MPI; Figure 4 represents just one possible distribution.

APPS-MPI uses two classes to manage and perform function evaluations. Each worker owns an instance of `FevalWkr` ("function evaluation worker").

There is only one instance of the other class, `FevalMgr` ("function evaluation manager"), and it is owned by the agent. `FevalMgr` is the agent's interface to the workers. The agent calls methods on `FevalMgr` to start function evaluations and to check for the results of finished function evaluations.

APPS-MPI compares favorably to APPS-PVM in terms of speed. Table 1 compares the speed of APPS-PVM in single- and multi-agent mode to APPS-MPI using two freely available implementations of MPI, LAM [7] and MPICH [3, 4]. (For information on obtaining LAM or MPICH, see Appendix D.) We ran a four-dimensional test problem (broyden2a of Example 3) with eight search directions 25 times for each APPS configuration and averaged the timings. We did the same for 16, 24, and 32 search directions. For these tests we used a convergence tolerance of 10^{-3}, the default "stall factor" of 100, no cache, and no bounds. The tests were run on a cluster of DEC Alpha Miata 433 MHz processors. Because of the way the runs were timed, timings for MPI do not include the time needed to start all the processes before the program begins running (usually about one second). Timings for PVM *do* reflect the time needed to spawn all processes, since the spawning takes place after APPS-PVM has started.

Search Directions	**LAM**	**MPICH**	**PVM Single**	**PVM Multi**
8	3.60	3.61	11.12	17.43
16	1.21	2.84	15.22	12.88
24	0.41	2.48	15.45	11.95
32	0.61	2.27	8.80	11.96

Table 1: Time (in seconds) to solve a four-dimensional test problem (averaged over 25 runs) for APPSPACK version 1.5. (Version 2.0 of APPSPACK is significantly faster for PVM, and new results will be included in a future version of this manual.)

4 Compilation and Usage

After downloading APPSPACK, unzipping it will create the directory `appspack`, and the full path to this directory is hereafter referred to as `$APPSPACK`. The directory structure inside `$APPSPACK` is shown in Figure 5

To compile, follow the directions in `$APPSPACK/INSTALL.txt`. In general, the procedure is

```
>> cd $APPSPACK
>> ./configure
>> make
```

The default is to install the PVM version, and special options to `configure` are required to enable MPI or simply disable PVM. Full details are given in

`$APPSPACK`	Main directory
`$APPSPACK/apps`	APPS application and makefile
`$APPSPACK/apps/src`	APPS source files
`$APPSPACK/apps/include`	APPS include files
`$APPSPACK/bin`	MPI & other executables plus perl utilities
`$APPSPACK/doc`	Documentation files
`$APPSPACK/ex1`	Source code and makefile for Example 1
`$APPSPACK/ex2`	Source code and makefile for Example 2
`$APPSPACK/ex3`	Source code and makefile for Example 3

Figure 5: APPSPACK Directory Structure

the installation instructions in `INSTALL.txt`. Whether using the PVM or MPI version, you must insure that all executable files are copied to every host in the computing platform. (The perl script `$APPSPACK/bin/multicmd` may be used to accomplish the copying. Type `multicmd -h` for instructions.)

The basic usage statement is:

```
$APPS [$OPTIONS] $PARAMS
```

where `$APPS` is the application name (which depends on the communication architecture and the problem name), `$OPTIONS` refers to various APPS options (such as the convergence tolerance) as described in Appendix A, and `$PARAMS` refers to the function evaluation parameters, which depend upon the particular function evaluation being used and are described later in this section.

The formula for constructing the APPS executable name is

```
$APPS = apps_$XCOM_$X
```

where `$XCOM` is the communication architecture and `$X` is the problem name. For example, the generic PVM version version of APPS is named

```
apps_pvm_generic
```

The generic PVM version of APPS depends on a generic function evaluation interface. If a special function interface is required (as illustrated for Example 3 in §6.3 below), then the name would be something like

```
apps_pvm_ex3
```

For MPI, every executable is customized since it must be linked with the function evaluation at compilation. So, it would be named, e.g.,

```
apps_mpi_ex3
```

To run an MPI version, type

```
mpirun -np $N apps_mpi_ex3 [$OPTIONS] $PARAMS
```

where `$N` is the number of search directions plus two if APPS is run with the function value cache (the default) or plus one if APPS is run without the cache (see §7 for details about the cache).

For implementations that do not use PVM or MPI, "other" is the default name of the communication architecture, although that may be customized. For example, we have included a serial version of APPS for each example, and the version for Example 3 is named:

```
apps_other_ex3
```

The location of the executables depends on the communication architecture. The PVM programs are put in the directory

```
$HOME/pvm3/bin/$PVM_ARCH
```

while other executables are put in the directory

```
$APPSPACK/bin
```

The location of the executables may be modified by specifying `--bindir` with `configure` or modifying the file `$APPSPACK/Makefile.defs` after running `configure`.

The `$PARAMS` are the parameters for the function evaluation. For PVM, the default parameters are `$NAME`, the executable name of the spawned function evaluation program, and `$NDIM`, the problem dimension. For MPI there is only one default parameter — `$NDIM`, the problem dimension. The parameters may be changed by modifying the `MyFevalMgr` object, as described in §6. Type

```
$APPS --help
```

to list the function evaluation parameters.

5 Output

APPS's output can give varying levels of information about what the algorithm is doing. The level of output is controlled by APPS's `debug` option. The default is `--debug=2`, which outputs initialization information for the first agent started, each new minimum found, any agent failures detected by the first agent, and the location of the best point found. The eleven debug levels are described in Appendix A.

By default, APPS prints all output to standard I/O, but output can be redirected to a file using the `output` option described in Appendix A.

Here we will describe APPS's output. The output examples come from running APPS-PVM with two agents on a two-dimensional problem (broyden2a of Example 3; see §6.3). The command line used was:

```
>> apps_pvm_ex3 broyden2a 2
```

Since APPS is a parallel application, output sometimes appears out of order or jumbled. In general, APPS's output will appear roughly in the order described here, although small variations should be expected. The output from the command above is shown in Figure 6. The line numbers are not part of APPS's output; they have been added for clarity. The text `xx->` at the beginning of each line, where `xx` is an agent number, indicates what agent that line of output originated from.

5.1 Initialization Output

Lines 1–24 in Figure 6 show the initialization information.

Lines 1–3 show parameter choices; see `debug`, `profile`, and `epsilon` in Appendix A.

Lines 4–6 show details about the agents. The agent that constructed the agent information is Agent 0. The globally unique identifier (GUID) attached to this agent information in specified by the tag of 000_004. Whenever the agent information is refreshed (due to the addition or deletion of the host), a new GUID is generated. The GUID can be used to determine the most up-to-date agent information in case messages arrive out of order. Line 4 also lists the number of agents and the name of the apps executable. Lines 5 & 6 list details about each agent. Most importantly, `isalive` says whether the agent is still functioning. In single agent mode, the details about each agent are not listed.

Lines 7–8 show the details of the mapping of search directions to agents. Line 7 indicates that there are 4 search directions and that all 4 directions are in the core pattern (i.e., active), search directions may be reassigned, there is no model (this feature is under development), and the total number of jobs to assign is 4. The actual job assignment is list in Line 8: Search Directions 0 and 2 are assigned to Agent 0 and Search Directions 1 and 3 are assigned to Agent 1.

Line 9 lists details about the cache manager process. The cache is described in detail in §7. Here we see that the cache is being used `usecache=1`, the PVM task id for the given, and a GUID is assigned to the information. Once again, the GUID may be used to determine the most up-to-date information.

Line 10 will detail the model manager information when that functionality is implemented.

Line 11 lists the parameters used in the line search; see Appendix A for description of each item.

Line 12 lists information about the function evaluation. This differs for each type of function evaluation. In this case, the name of the executable is listed, the specific problem name, the dimension, and two other parameters (see §6.3).

Line 13 indicates that there are no bounds. If there were bounds, then that line would be replaced by the following two lines:

```
00-> Bounds lower=[ -1.00e+00 0.00e+00 ]
00-> Bounds upper=[  1.00e+00 2.00e+00 ]
```

```
00-> Global debug = 2
00-> Global profile = 2
00-> Point epsilon = 1e-08
00-> Agent myid=0 tag=000_004 nagents=2 name=apps_pvm_ex3
00-> Agent 0: isalive=1 hostid=40000 taskid=40023 speed=1
hostname=hopper
00-> Agent 1: isalive=1 hostid=80000 taskid=8006f speed=1
hostname=hypatia
00-> JobMap nsearch=4 nactive=4 reassign=1 ismodel=0 njob=4
00-> Job Assignment = [ 0 1 0 1 ]
00-> CacheMgr usecache=1 cachetid=80070 guid=000_005
00-> ModelMgr usemodel=0
00-> Step initstep=1 tol=0.0001 theta=0.5 minstep=0.0002 maxstep=100000
alpha=0 inc=1 break=1
00-> FevalMgr name=ex3 probname=broyden2a ndim=2 usebounds=0 stall=1
00-> Bounds:  None
00-> Bounds scale=[ 1.00e+00 1.00e+00 ]
00-> Pattern ndim=2 nsearch=4 type=0 rnormtol=1e-08
00-> Pattern column column 0 = [ 1.00e+00 0.00e+00 ]
00-> Pattern column column 1 = [ 0.00e+00 1.00e+00 ]
00-> Pattern column column 2 = [-1.00e+00 0.00e+00 ]
00-> Pattern column column 3 = [ 0.00e+00 -1.00e+00 ]
00-> INITIAL POINT f=<null> x=[-1.00e+00 -1.00e+00 ] step=1.00e+00
tag=000_003 conv=[ 0 0 0 0 ]
00-> Search 0 Scaled Dir=[ 1.00e+00 0.00e+00 ]
01-> Search 1 Scaled Dir=[ 0.00e+00 1.00e+00 ]
01-> Search 3 Scaled Dir=[ 0.00e+00 -1.00e+00 ]
00-> Search 2 Scaled Dir=[-1.00e+00 0.00e+00 ]
01-> Search 1 New Min f= 6.64e+01 x=[-1.00e+00 0.00e+00 ] step=1.00e+00
tag=001_006 conv=[ 0 0 0 0 ]
00-> Search 0 New Min f= 2.00e+00 x=[ 0.00e+00 -1.00e+00 ] step=1.00e+00
tag=000_008 conv=[ 0 0 0 0 ]
00-> Search 2 New Min f= 1.33e+00 x=[-5.00e-01 -1.00e+00 ] step=5.00e-01
tag=000_014 conv=[ 0 0 0 0 ]
00-> Search 0 New Min f= 1.13e+00 x=[-2.50e-01 -1.00e+00 ] step=2.50e-01
tag=000_021 conv=[ 0 0 0 0 ]
00-> Search 2 New Min f= 1.00e+00 x=[-3.75e-01 -1.00e+00 ] step=1.25e-01
tag=000_028 conv=[ 0 0 0 0 ]
00-> Search 0 New Min f= 1.00e+00 x=[-3.71e-01 -1.00e+00 ] step=3.91e-03
tag=000_042 conv=[ 0 0 0 0 ]
00-> Search 2 New Min f= 1.00e+00 x=[-3.72e-01 -1.00e+00 ] step=9.77e-04
tag=000_051 conv=[ 0 0 0 0 ]
00-> Search 0 New Min f= 1.00e+00 x=[-3.72e-01 -1.00e+00 ] step=4.88e-04
tag=000_057 conv=[ 0 0 0 0 ]
00-> FINAL MINIMUM f= 1.00e+00 x=[-3.72e-01 -1.00e+00 ] step=4.88e-04
tag=000_057 conv=[ 1 1 1 1 ]
00-> Search 2 Total time:  8.04e-01 Idle time:  5.04e-01 Fevals:  14
Fail:  0 Break:  1 Flop:  0 Cache:  7 Total:  22
00-> Search 0 Total time:  8.04e-01 Idle time:  4.70e-01 Fevals:  15
Fail:  0 Break:  2 Flop:  0 Cache:  6 Total:  23
00-> Cache Lookup Time Min:  5.10e-03 Max:  4.86e-02 Mean:  1.67e-02
00-> Agent 0 (hopper) exiting
01-> FINAL MINIMUM f= 1.00e+00 x=[-3.72e-01 -1.00e+00 ] step=4.88e-04
tag=000_057 conv=[ 1 1 1 1 ]
01-> Search 3 Total time:  7.99e-01 Idle time:  3.53e-01 Fevals:  32
Fail:  0 Break:  7 Flop:  0 Cache:  0 Total:  39
01-> Search 1 Total time:  7.99e-01 Idle time:  3.72e-01 Fevals:  38
Fail:  0 Break:  5 Flop:  0 Cache:  0 Total:  43
01-> Cache Lookup Time Min:  5.33e-04 Max:  8.43e-03 Mean:  5.44e-03
01-> Agent 1 (hypatia) exiting
```

Figure 6: Sample APPSPACK initialization output.

Line 14 shows the scaling information for the problem. The default scaling is 1.0, i.e., no scaling.

Lines 15–19 show the information of the search pattern. Line 15 gives details about the pattern; see options `pattern` and `rnormtol` is Appendix A. The next four lines list the search directions.

Line 20 lists the initial point – more details on this are given in the next section. The `f=<NULL>` indicates that there is no function value assigned to this point.

Lines 21–24 are the scaled search directions. Note that lines 21 and 23 come from Agent 1, the owner of those directions.

5.2 Status Output

After the initialization information, APPS prints out information about the progress of the algorithm. Lines 25–32 in Figure 6 show the status of the search. At Debug Level 2, only new minimums (i.e., an improvement on the best known point) are shown. Figure 7 explains each part of these output lines.

$$\underbrace{\texttt{01->}}_{a}\ \underbrace{\texttt{Search 1}}_{b}\ \underbrace{\texttt{New Min}}_{c}\ \underbrace{\texttt{f= 6.64e+01}}_{d}\ \underbrace{\texttt{x=[-1.00e+00 0.00e+00]}}_{e}$$

$$\underbrace{\texttt{step=1.00e+00}}_{f}\ \underbrace{\texttt{tag=001\_006}}_{g}\ \underbrace{\texttt{conv=[0 0 0 0]}}_{h}$$

a	The agent this output originated from.
b	The search direction that found a new minimum.
c	The type of event. `New Min` is the only event reported at the default output level.
d	The function value at the new minimum.
e	The coordinates of the new minimum.
f	The step size used when the new minimum was found.
g	This point's globally unique identifier (GUID).
h	The convergence table. Indicates whether each search direction had (1) or had not (0) converged.

Figure 7: Explanation of New Minimum output lines.

In addition to new minimums, APPS also reports the following other type of messages about points it is evaluating (or trying to evaluate).

- `F Cached` — The function value has been retrieved from the cache.
- `F No-Go` — The function evaluation is not started because the trial point is outside of the bounds and cannot be made feasible
- `F NoSpawn` — The function fails to properly spawn. This is rare.

- `F Insuff` — Insufficient decrease, i.e., even though the function value is better than the best known, it is not enough of a decrease.
- `Converge` — The search in this direction is converged for this point.
- `Contract` — The step length is reduced.
- `Merge Min` — Two convergence tables are merged.
- `Switch` — Switching to a better point produced by another search direction.
- `F Start` — Starting a new function evaluation.

APPS also reports its progress with other messages as well. For example, it will report `Breaking Feval` and indicate the PVM taskid of the function evaluation. (This only occurs in the PVM version.)

Failures and additions of hosts/agents are also reported. For example, if an agent fails *and* the failure is detected by the first agent:

```
00-> Deleting agent x
```

where `x` is the number of the agent that failed. APPS then prints lines 4–10 of the initialization output again, updated to reflect the new state of the agents.

5.3 Shutdown Output

Once APPS converges, each agent prints its final minimum; see lines 33 and 38 in Figure 6. In general, the final minimums should agree, but there may be occasions when an agent finds a better point between the another agent determines the process has converged and the message saying so is received. In this example, the convergence table contains all 1's, indicating every search direction has converged. This will not always be the case. If some subset of the search directions all converge and that subset forms a positive basis, then APPS could terminate before all the search directions converge [5].

A *search profile* is printed for each search direction; see lines 34–35 and 39–40. The times are wall-clock times, in seconds. `Total time` is the amount of time the search direction ran. `Idle time` is the amount of time the search direction spent not doing function evaluations. The second part of the search profile deals with the number of function evaluations done in that direction.

- `Fevals` is the number of function evaluations this search direction performed without a problem.
- `Fail` is the number of function evaluations that did not return a value.
- `Break` is the number of function evaluations started but terminated prematurely because the search direction found out about a new best point.
- `Flop` is the number of function evaluations that were not executed because either

   1. PVM failed to spawn them, or
   2. the trial point was outside the bounds and its projection back inside the bounds was the base point of the pattern.

- `Cache` is the number of times the search direction used a saved function value from the cache instead of evaluating the objective function again.
- `Total` is the total number of function evaluations:
  `Total = Fevals + Fail + Break + Flop + Cache`.

If the cache is being used, each agent also prints out a *cache profile*; see lines 36 and 41. All the numbers in this profile are times, in seconds, and include time spent communicating between processors.

- `Min` is the smallest amount of time spent on any single cache lookup.
- `Max` is the largest amount of time spent on any single cache lookup.
- `Mean` is the average time spent per cache lookup.

The utility `$APPSPACK/bin/parsestats` can be used to evaluate the output from APPSPACK; see Figure 8 for the statistics that would be generated from the output in Figure 6. The function values are listed — this enables us to quickly judge is all processors are in agreement. Additionally, cumulative statistics are printed for the searches and the cache managers. For the searches, the summary is the *maximum* total time and the *mean* for all the other columns. For the cache, it is interesting to note the Agent 1's cache access was much faster than Agent 0's. This is likely due to the fact that the cache process resided on the same host as Agent 1.

6 Using APPS with a User's Function Evaluation

This section describes the three example problems included with APPS. These examples demonstrate how to plug a function evaluation into APPS in several different ways.

6.1 Example 1

This example demonstrates how APPS may be used with an objective function whose calling sequence has the format:

```
double feval(int ndim, double* x)
```

where `ndim` is the number of dimensions and `x` is the point to be evaluated (an `ndim`-array of doubles). The return value is a double.

Our example uses a simple quadratic function. To compile Example 1, change directories to `$APPSPACK` and type

```
*** FINAL F VALUES (2) ***

 Agent   Final F(X)
 -----   ----------
     0 :  1.00e+00
     1 :  1.00e+00
         ----------
Mean   :  1.00e+00
Median :  1.00e+00
Min    :  1.00e+00
Max    :  1.00e+00

*** SEARCH STATISTICS (4) ***

 Agent/
 Search  Total Time Idle Time Fevals Fails Breaks Flops Cache Total
-------  ---------- --------- ------ ----- ------ ----- ----- -----
 0 / 2 :   8.04e-01  5.04e-01     14     0      1     0     7    22
 0 / 0 :   8.04e-01  4.70e-01     15     0      2     0     6    23
 1 / 3 :   7.99e-01  3.53e-01     32     0      7     0     0    39
 1 / 1 :   7.99e-01  3.72e-01     38     0      5     0     0    43
-------  ---------- --------- ------ ----- ------ ----- ----- -----
Mean   :   8.01e-01  4.25e-01   24.8   0.0    3.8   0.0   3.2  31.8
Median :   8.02e-01  4.21e-01   23.5   0.0    3.5   0.0   3.0  31.0
Min    :   7.99e-01  3.53e-01     14     0      1     0     0    22
Max    :   8.04e-01  5.04e-01     38     0      7     0     7    43
-------  ---------- --------- ------ ----- ------ ----- ----- -----
Summary:   8.04e-01  4.25e-01   24.8   0.0    3.8   0.0   3.2  31.8

*** CACHE STATISTICS (2) ***

 Agent    Min Time  Max Time  Mean Time
 -----    --------  --------  ---------
     0 :  5.10e-03  4.86e-02   1.67e-02
     1 :  5.33e-04  8.43e-03   5.44e-03
          --------  --------  ---------
Mean   :  2.82e-03  2.85e-02   1.11e-02
Median :  2.82e-03  2.85e-02   1.11e-02
Min    :  5.33e-04  8.43e-03   5.44e-03
Max    :  5.10e-03  4.86e-02   1.67e-02
```

Figure 8: Sample output from `parsestats` utility.

```
>> make example1
```

To execute Example 1 for `$NDIM` dimensions, type

PVM	`apps_pvm_generic ex1 $NDIM`
MPI	`mpirun -np $N apps_mpi_ex1 $NDIM`
OTHER	`apps_other_ex1 $NDIM`

In the MPI example, `$N` $= 2 \cdot$ `$NDIM` $+\ 2$.

To substitute your own function proceed as follows. First, copy some files.

```
>> cd $APPSPACK/ex1
>> cp Makefile $MYFUNCDIR/Makefile
>> cp ex1.c $MYFUNCDIR/$MYFUNC.c (PVM only)
```

We assume there is a file called `feval.c` containing the user's function definition. Be sure the function evaluation call is of the form

```
double feval(int n, double *x)
```

Then, modify the Makefile by replacing `ex1` everywhere with `$MYFUNC`.

PVM Compile `$MYFUNC` as follows:

```
>> cd $MYFUNCDIR
>> make
```

To run the PVM version of APPS, type

```
>> apps_pvm_generic $MYFUNC $NDIM
```

Note that APPSPACK itself does not need to be recompiled.

MPI To use MPI, be sure to run configure with the `--enable-mpi` option. Compile `feval.o` by typing

```
>> cd $MYFUNCDIR
>> make feval.o
```

For the MPI version, we need to recompile APPS as follows:

```
>> cd $APPSPACK/apps
>> make X=$MYFUNC XOBJS=../$MYFUNCDIR/feval.o
```

The variable `$XOBJS` specifies files that APPS should link with; in this case, the object file which contains the function evaluation.

To run the MPI version of APPS, type

```
>> mpirun -np $N apps_mpi_$MYFUNC $NDIM
```

where `$N` $= 2 \cdot$ `$NDIM` $+\ 2$.

Other We have also included a serial version for an example of how to run APPS without PVM or MPI (be sure to run configure with the `--disable-pvm` option). Instead of a serial version, however, the user may plug into third-party software for executing function evaluations. Compile `feval.o` by typing

```
>> cd $MYFUNCDIR
>> make feval.o
```

For the serial version, we need to recompile APPS as follows:

```
>> cd $APPSPACK/apps
>> make X=$MYFUNC XMGRDIR=$MYFUNCDIR XOBJS=$MYFUNCDIR/feval.o
```

Here `$XMGRDIR` specifies the directory containing an alternate version of `myfevalmgr.C` that should be used in place of the default in `$APPSPACK/apps/src/myfevalmgr.C`. This file contains the `MyFevalMgr` object which is derived from `FevalMgr`. This object controls the interface to the function evaluation. Details on creating an alternate `MyFevalMgr` are given for Example 3 in §6.3. Adding `XCOM=serial` to the `make` command line would replace `other` with `serial` in the name of the executable.

To run the "other" version of APPS, type

```
>> apps_other_$MYFUNC $NDIM
```

6.2 Example 2

Example 2 illustrates how to use APPS when the function evaluation cannot be called via a simple subroutine, but instead must execute independently. In Example 2 we use a *wrapper program* to create a unique working directory, generate an input file, execute the function evaluation via a C system call, read and process the output file, and then delete the working directory and its contents. To compile, change directories to `$APPSPACK` and type

```
>> make example2
```

Execute Example 2 as follows:

PVM	`apps_pvm_generic --nobreak ex2_wrapper $NDIM`
MPI	`mpirun -np $N apps_mpi_ex2 $NDIM`
OTHER	`apps_other_ex2 $NDIM`

The `--nobreak` option is required in the PVM version in order to make sure all the temporary files are deleted. In the MPI example, `$N` $= 2 \cdot$ `$NDIM` $+\ 2$.

To substitute a user's function evaluation named `$MYFUNC` in the directory `$MYFUNCDIR`, first copy the contents of the `$APPSPACK/ex2` directory as follows:

```
>> cd $APPSPACK/ex2
>> cp Makefile $MYFUNCDIR/Makefile
>> cp ex2.h $MYFUNCDIR/$MYFUNC.h
```

```
>> cp wrapper.C $MYFUNCDIR/wrapper.C
>> cp ex2_wrapper.C $MYFUNCDIR/$MYFUNC_wrapper.C
>> cp myfevalmgr.C $MYFUNCDIR/myfevalmgr.C
>> cp myfevalwkr.C $MYFUNCDIR/myfevalwkr.C
```

Then follow these steps:

1. We assume that the program `$MYFUNC` is already compiled and expects an input file of the form:

   ```
   $NDIM
   $X[1]
   ⋮
   $X[$NDIM]
   ```

   where `$NDIM` is the number of parameter and {`$X[1]`, ..., `$X[$NDIM]`} is the set of parameters to be evaluated. The output file should contain the function value as its first entry. To change the format of the input file or to read the output file differently, modify `$MYFUNCDIR/wrapper.C`.

2. Change `$MYFUNCDIR/$MYFUNC.h` to specify the names of the input and output files for the program `$MYFUNC`.

3. Next, for PVM modify `$MYFUNCDIR/$MYFUNC_wrapper.C`, replacing `ex2` everywhere with `$MYFUNC`. For MPI, do the replacement in `$MYFUNCDIR/myfevalwkr.C`, and for the serial version, do the replacement in `$MYFUNCDIR/myfevalmgr.C`.

4. Finally, modify the Makefile by replacing `ex2` everywhere with `$MYFUNC`.

PVM We assume that `$MYFUNC` is already compiled. Compile the wrapper program by typing

```
>> cd $MYFUNCDIR
>> make $MYFUNC_wrapper
```

Be sure to copy `$MYFUNC` to the directory containing PVM executable files (for example, `$HOME/pvm3/bin/$PVM_ARCH`) on all hosts. Then, to run the PVM version of APPS, type

```
>> apps_pvm_generic $MYFUNC_wrapper $NDIM
```

MPI We assume that `$MYFUNC` is already compiled. Compile `wrapper.o` by typing

```
>> cd $MYFUNCDIR
>> make wrapper.o
```

For the MPI version, we need to recompile APPS as follows:

```
>> cd $APPSPACK/apps
>> make X=$MYFUNC XWKRDIR=$MYFUNCDIR \
>>      XOBJS=$MYFUNCDIR/wrapper.o
```

The variable `$XWKRDIR` specifies that an alternate version of `myfevalwkr.C` should be used in place of `$APPSPACK/apps/src/myfevalwkr.C`. The `MyFevalWkr` object is the function evaluator for the MPI version, and it is derived from the default `FevalWkr` object. In this case, we want `FevalWkr` to do something different than the default version that is used in Example 1. In addition to the constructor and destructor, we only need define one function in the derived object: `doFeval()`. In addition, the following function must be defined in the `myfevalwkr.C` file as shown:

```
FevalWkr* myfevalwkr() { return new MyFevalWkr; }
```

Notice also that a special class, `AppsComm`, is used as a wrapper to the standard MPI and PVM commands. See `$APPSPACK/apps/src/appscomm.C` and `$APPSPACK/apps/include/appscomm.H` for the exact specification. Note that by default, the compile flag `-I$XWKRDIR` is included in the compilation of `myfevalwkr.C`.

Be sure to copy `$MYFUNC` to `$APPSPACK/bin` on all hosts. Then, to run the MPI version of APPS, type

```
>> mpirun -np $N apps_mpi_$MYFUNC $NDIM
```

where $\texttt{\$N} = 2 \cdot \texttt{\$NDIM} + 2$.

Other We assume that `$MYFUNC` is already compiled. Compile `wrapper.o` by typing

```
>> cd $MYFUNCDIR
>> make wrapper.o
```

For the serial version, we need to recompile APPS as follows:

```
>> cd $APPSPACK/apps
>> make X=$MYFUNC XMGRDIR=$MYFUNCDIR \
>>      XOBJS=$MYFUNCDIR/wrapper.o
```

Note that by default, the compile flag `-I$XMGRDIR` is included in the compilation of `myfevalmgr.C`.

Be sure to copy `$MYFUNC` to `$APPSPACK/bin` on all hosts. Then, to run the "other" version of APPS, type

```
>> apps_other_$MYFUNC $NDIM
```

where $\texttt{\$N} = 2 \cdot \texttt{\$NDIM} + 2$.

6.3 Example 3

Example 3 is a special case where we must redefine `MyFevalMgr`, even for the PVM version. There are several reasons why this problem is different than the others:

1. We have (optional) bounds on the variables. The default `MyFevalMgr` assumes all problems are unconstrained.
2. The user has the choice of *six* different functions, all accessible via one program.
3. The user may specify an optional parameter to slow down the function evaluation—this is useful in simulating more expensive problems. We call this parameter the *stall factor*, and it specifies the size of an arbitrary non-negative least squares (NNLS) problem to be solved by the function evaluation.

The manner in which these differences are accommodated is described below. The new calling sequence is

```
apps_$XCOM_ex3 $PROBLEM_NAME $NDIM [$USE_BOUNDS] [$STALL]
```

Here, `$PROBLEM_NAME` is one of six function evaluation names (e.g., `broyden2a`), `$NDIM` is the problem size, `$USE_BOUNDS` is `1` to use the bounds and `0` otherwise, and `$STALL` is a positive integer representing the size of the NNLS problem to be solved. Both `$USE_BOUNDS` and `$STALL` are optional, as indicated by the brackets, with defaults of `0` and `1`, respectively.

To compile Example 3, change directories to `$APPSPACK` and type

```
>> make example3
```

To use Example 3:

PVM	`apps_pvm_ex3 $PROBLEM_NAME $NDIM`
MPI	`mpirun -np $N apps_mpi_ex3 $PROBLEM_NAME $NDIM`
OTHER	`apps_other_ex3 $PROBLEM_NAME $NDIM`

All three also take the following optional arguments: [`$USE_BOUNDS`] and [`$STALL`].

As stated above, an alternate to `MyFevalMgr` is provided in the file `$APPSPACK/ex3/myfevalmgr.C`. The new object has three additional variables to specify the problem name (`probname`), whether or not to use the bounds (`usebounds`), and the stall factor (`stall`). The following functions are defined in the derived object `MyFevalMgr`.

- `MyFevalMgr()`: The constructor initializes the additional variables to NULL or equivalent defaults. The constructor is not passed any arguments. The parameters used to set these objects are passed into `setFevalMgr` as described below.
- `~MyFevalMgr()`: The destructor deletes any memory used by the additional variables.

- `showUsage()`: The usage statement is modified to describe the parameters that are expected by this particular function evaluation.
- `setFevalMgr()`: This function is passed a vector of strings named `fargs` containing the parameters on the command line as well as a boolean indicating whether or not APPS is in single-agent mode (`issingle`). This is where all the parameters on the `FevalMgr` object are actually set. The vectors `initx`, `lower`, and `upper` are set in the functions as well. By default, they have length zero.
- `leftshift()`: Prints function evaluation parameters to the given stream. This is modified since we have additional parameters for this function evaluation.
- `spawn()`: This is the function which starts a function evaluation. In this case, we need only modify it in PVM mode.
- `pack()` (PVM only): We need to pack any additional variables.
- `unpack()` (PVM only): We need to unpack any additional variables.
- `initWorkers()` (MPI only): Initialize the function evaluation worker tasks. The only thing that should need to be modified here is what data is sent to initialize the tasks.

In addition to the new derived class, the following function *must* be defined within `myfevalmgr.C`:

```
FevalMgr* myfevalmgr() { return new MyFevalMgr; }
```

In the MPI version, we similarly require a new version of the `FevalWkr` object, in the file `$APPSPACK/ex3/myfevalwkr.C`. The new derived class has two additional variables: `probname` and `stall`, as defined above. We redefine the following functions:

- `MyFevalWkr()`: The constructor now must also initialize the additional variables.
- `~MyFevalWkr()`: The destructor must also destroy any additional variables.
- `initialize()`: A different initialization packet is sent by the manager, so the worker must receive different information.
- `doFeval()`: This is a pure virtual function and as such must always be redefined.
- `leftshift()`: Prints out the worker's parameters to the given stream.

To modify Example 3 to your own function evaluation named `$MYFUNC` in the directory `$MYFUNCDIR`, proceed as follows:

```
>> cd $APPSPACK/ex3
>> cp Makefile $MYFUNCDIR/Makefile
>> cp myfevalmgr.C $MYFUNCDIR/myfevalmgr.C
>> cp myfevalwkr.C $MYFUNCDIR/myfevalwkr.C
```

Then, place your function evaluation inside the file `feval.C` and modify `myfevalmgr.C` and `myfevalwkr.C` accordingly, using the guidelines above. There are other vectors in the `FevalMgr` object that the user may want to initialize in their version of `MyFevalMgr` such as:

- `islower`: A boolean array indicating which lower bounds are active and inactive.
- `isupper`: A boolean array indicating which upper bounds are active and inactive.
- `scale`: A scaling vector; the default scaling is 1.0 (i.e., none).

Finally, modify the Makefile by replacing `ex3` everywhere with `$MYFUNC`.

PVM Compile the PVM version as follows:

```
>> cd $MYFUNCDIR
>> make feval.o
>> cd $APPSPACK/apps
>> make X=$MYFUNC XMGRDIR=$MYFUNCDIR \
>>      XOBJS=$MYFUNCDIR/feval.o
```

MPI The main difference between the MPI and PVM version is the inclusion of the `MyFevalWkr` object, which PVM does not require. Compile the MPI version as follows:

```
>> cd $MYFUNCDIR
>> make feval.o
>> cd $APPSPACK/apps
>> make X=$MYFUNC XMGRDIR=$MYFUNCDIR \
>>      XWKRDIR=$MYFUNCDIR XOBJS=$MYFUNCDIR/feval.o
```

Other Compile the serial version as follows:

```
>> cd $MYFUNCDIR
>> make feval.o
>> cd $APPSPACK/apps
>> make X=$MYFUNC XMGRDIR=$MYFUNCDIR \
>>      XOBJS=$MYFUNCDIR/feval.o
```

7 Function Value Cache

APPS uses the function value cache to speed up convergence by avoiding evaluating the objective function at the same point more than once. Every time APPS does a function evaluation, it saves the function value and point in the cache. Before a function evaluation, APPS checks the cache to see if that evaluation has already been done. If so, APPS uses the cached function value instead of doing the function evaluation again.

Without the cache, the chance of repeating function evaluations at the same point is high. On one four-dimensional test problem (chebyquad of Example 3), an average of about 22% of function evaluations were at previously visited points. Because APPS is visiting points that lie on a regular grid, there is a good chance it will come back to a point several times.

For a simple example of how this can happen, consider the situation in Figure 9. APPS evaluates the function at the point indicated by $\times$ and finds the value there is greater than the best known value so far (a). APPS therefore performs a contraction (b) and evaluates the function at the point indicated by $\star$. As it turns out, the function value at the point indicated by $\star$ is the lowest function value seen so far, so APPS moves that search direction. Now APPS is evaluating the function at the point indicated by $\times$ once again (c).

Although the cache is designed to save time when situations like that in Figure 9 occur, there is a chance that using the cache will increase the time it takes APPS to converge. If a function evaluation takes less time than a cache lookup, then the extra work of doing a cache lookup before every function evaluation is not worthwhile. However, the worst average time for a cache lookup (including time for communication between processors) we have seen in our tests is $O(10^{-2})$ seconds.

The cache could also slow down convergence if APPS never or almost never revisits a point. Then APPS will incur the cost of the cache lookup at each function evaluation but never get any benefit. Again, though, this is not expected to be a major problem because of the short time required for a cache lookup.

If the cache does cause problems, it can be disabled using APPS's `--nocache` option (see Appendix A).

Table 5 in Appendix C gives some numerical results concerning the performance of the cache.

7.1 Cache Data Structure

APPS's function value cache stores points in a splay tree. A splay trees is a binary search tree that uses a series of rotations to move any accessed node to the root. Splay trees are described in [9].

We also considered using red-black trees, a type of balanced binary search tree, as the underlying data structure for the cache. Red-black trees are described in [1]. Unlike splay trees, which may in rare cases degenerate into linked lists, red-black trees are always guaranteed to be "approximately" balanced

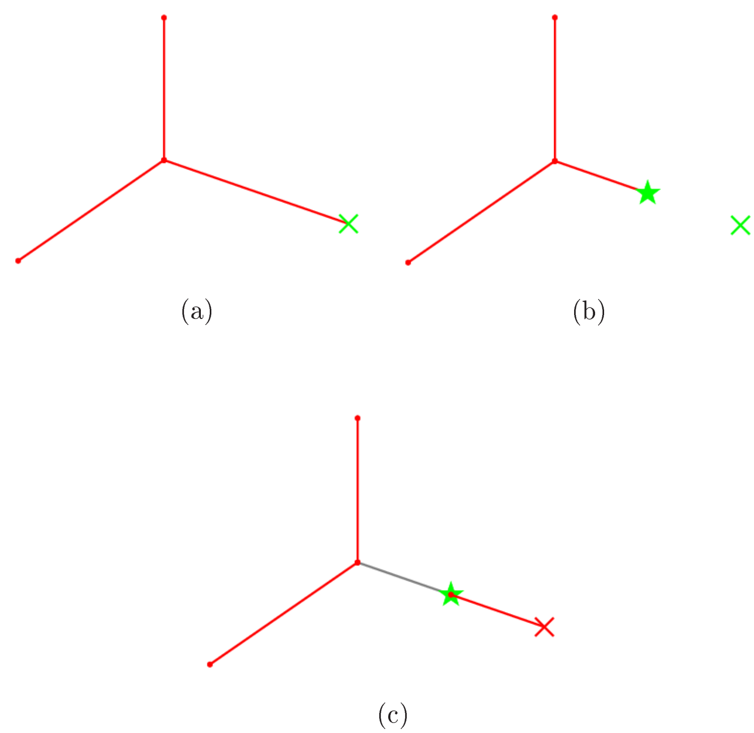

Figure 9: A repeated function evaluation.

(specifically, no path from the root to a leaf will be more than twice as long as any other). This in turn means that searching a red-black tree is an $O(\log n)$ operation, whereas splay trees only guarantee searching in *amortized* $O(\log n)$ time, and can be as bad as $O(n)$. In addition, splay trees must do extra $O(\log n)$ work every time a node is accessed to move it to the root of the tree.

Despite these shortcomings, splay trees have advantages over red-black trees in this application. They are easier to implement and understand than red-black trees. Also, because splay trees keep the most recently accessed nodes near the root of the tree, searching can be quite fast *if* those nodes are also the ones most likely to be searched for. This is the case for APPS, which rarely revisits a point it evaluated long ago in the iteration history.

We compared the caching performance of APPS using splay trees and red-black trees on several test problems; see Table 2. The objective function for the four-, eight-, and sixteen-dimensional problems is `chebyquad`, one of the six functions available in Example 3 (see §6.3). The seventeen-dimensional problem (SPICE) involves fitting a computer simulation of voltages in a circuit to experimental data. It is described in [5]. The timings *do not* include communication between processors.

The splay tree implementation required far fewer comparisons, on average, for both insertion and search than the red-black tree implementation, but the difference in time is negligible. However, insertion and search times for red-black trees will grow as more points are inserted in the tree. For splay trees they will stay about the same. The time difference could become significant if many more function evaluations are stored in the cache.

		Insert			Search		
		Num	Time	Comps	Num	Time	Comps
4-D	Splay	275	$0.7 \cdot 10^{-5}$	2.62	340	$0.5 \cdot 10^{-5}$	2.29
	R-B	273	$0.9 \cdot 10^{-5}$	9.29	338	$0.6 \cdot 10^{-5}$	7.79
8-D	Splay	1688	$0.9 \cdot 10^{-5}$	3.20	2419	$0.7 \cdot 10^{-5}$	3.07
	R-B	1657	$1.8 \cdot 10^{-5}$	12.42	2386	$1.0 \cdot 10^{-5}$	11.00
16-D	Splay	6549	$0.8 \cdot 10^{-5}$	3.01	13074	$0.8 \cdot 10^{-5}$	2.26
	R-B	5517	$1.7 \cdot 10^{-5}$	14.96	10270	$1.2 \cdot 10^{-5}$	13.26
17-D	Splay	1400	$4.5 \cdot 10^{-5}$	5.98	1478	$2.3 \cdot 10^{-5}$	2.97
	R-B	1246	$5.1 \cdot 10^{-5}$	10.96	1312	$3.0 \cdot 10^{-5}$	9.62

Table 2: Comparison of caching using splay trees and red-black (R-B) trees. For each problem type, the number of inserts and searches (Num) is given, along with the average time for each insert or search (in seconds) and the average number of comparisons (Comps) for each insert or search.

Since APPS is designed to run in heterogeneous environments, the test the cache uses for equality of two points must take into account the possibility that the points it is comparing are not represented with the same precision. We do this by comparing points only to a user-specified degree of precision. For vectors x and y of length n, we say $x = y$ when (see Figure 10):

$$|x_i - y_i| < \frac{tol}{2}, \quad \text{for } i = 1, 2, \ldots, n.$$

where *tol* is APPS's convergence tolerance (set with the `--tol` parameter; the default is 10^{-4}). APPS's step size will not shrink below *tol*, so using the standard pattern of the plus and minus coordinate directions APPS cannot visit points less than *tol* apart (although using other search patterns APPS *could* visit points less than *tol* apart). If two points *are* less than *tol* apart, we can safely assume they are the same point.

When the cache finds a match for a point APPS has queried it about, the cache returns the point and the function value at that point. APPS not only uses the cached function value, but also changes its point to the point returned from the cache. This allows APPS's agents to meaningfully compare points when they are trying to decide if they have converged to the same point.

It is also necessary to impose some order on the points so the cache can place them in the tree. The test used for ordering points, like the test for equality, is

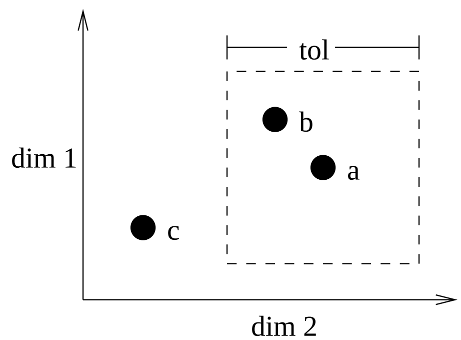

Figure 10: The cache considers $b = a$ and $c < a$.

based on a user-specified precision. We say x is less than y when there exists j such that (see Figure 10):

$$\begin{aligned} |x_i - y_i| &< \frac{tol}{2}, \quad \text{for } i = 1, 2, \ldots, j-1 \quad \textit{and} \\ y_j - x_j &> \frac{tol}{2}. \end{aligned}$$

The first inequality says that, by our definition, the first $j - 1$ coordinates of x and y are equal. The second inequality says that the jth coordinate of x is sufficiently less than the jth coordinate of y.

7.2 Cache Implementation

The APPS function value cache consists of two parts: the *cache worker* and the *cache manager*; see Figure 11. There is one cache worker, which runs as a separate process and contains all the cached data. Each agent contains a cache manager. The cache manager handles all communication between that agent and the cache worker. Cache managers communicate with the cache worker by sending two types of messages. A *lookup* message causes the cache worker to search the cache for a given point and return the function value at that point if it is in the cache. An *insert* message causes the cache worker to insert a given point in the cache.

In APPS-PVM, the cache worker is a separate executable. It is spawned at startup by the master agent and terminated by a message sent by a cache manager when it shuts down. The cache worker executable is named `apps_cachewkr`.

In APPS-MPI, the cache worker is part of the APPS executable. For n search directions, APPS-MPI is started with $n + 2$ processes to use the cache. As is the case without the cache, one of those processes becomes the agent and n others become workers, performing function evaluations. The remaining process becomes the cache worker. Since APPS-MPI only operates in single agent mode, there is only one cache manager.

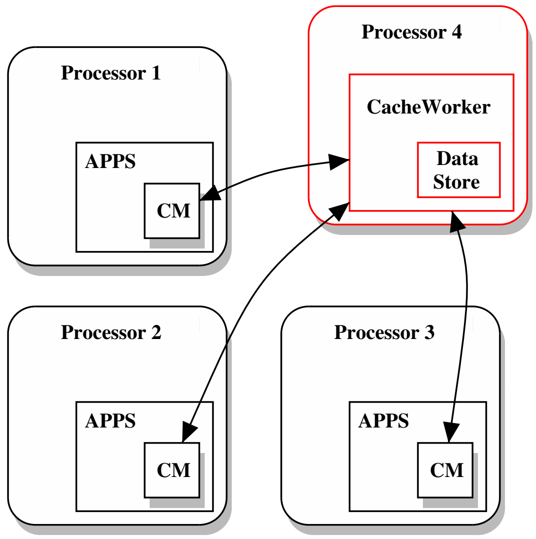

Figure 11: Each agent's cache manager (CM) handles communication with the cache worker.

A Options

The following are a list of parameters for APPSPACK. The options may also be listed by typing `$APPS --help`. Every option has a long form, and common options have a short (one letter) abbreviation. Parameters which do not match any of the options listed below are ignored. If parameters are repeated, the first one will be used. Options for APPS may be specified on the command line, in a file, or both.

On the command line, the long form is also preceded by a double dash and followed by an equal sign with while the short form is preceded by a single dash and followed immediately by the option value. There should be no spaces. Boolean options can take the form `--option=BOOL`, where `BOOL` is true or false. Alternatively, one can simply use `--option` for `--option=true` or `--nooption` for `--option=false`.

If some of the options are specified in a file, then the -i option must be specified to indicate the filename. The options in the file must be in the long format without the double-dash preceding the name (e.g., `tol = 1.0e-2`). Boolean options must be of the form OPTION = BOOL (e.g., `break = true`). There must be spaces before and after the equal sign.

- `-i, --input=FILENAME`

  The name of the APPSPACK options filename. It must be accessible on the host on which APPSPACK is started. If the filename is "", then no input file is read. Default: “”

- `-o, --output=FILENAME`

  Record the output from each agent in the file named FILENAME.XXX where the XXX is the agent id (between 0 and $n-1$ where N is the number of agents.) The file is created in the working directory as specified by PVM (usually `$HOME` or `/tmp`). The exception is that if the program is launched from the command line, agent zero will write its file in the directory in which it was launched. If the filename is "", then all output is directed to standard output of the first process or the PVM console. Default: ""

- `--single[=BOOL], --nosingle`

  Use 'single agent' mode. This is the default if PVM is not used. When APPS-PVM is run in single-agent mode, all parallelism is handled by the `FevalMgr` object — including starting and stopping PVM. For APPS-PVM, this mode may be faster than the default multi-agent mode, but it is much less fault tolerant. Default: false (PVM) / true (MPI & other)

- `--cache[=BOOL], --nocache`

  If true, use the function value cache. If false, do not use the cache. In APPS-MPI, using the cache requires an additional process. Since the cache currently requires either MPI or PVM, the `--nocache` option should be used with a custom communication interface. See §7 for more details about the cache. Default: true

- `-t, --tol=REAL`

  Specify the convergence tolerance. Default: 1.0e-4

- `--alpha=NUM`

  Specify the sufficient decrease parameter. Default: 0

- `-p, --pattern=NUM`

  Specify the pattern type where NUM is 0,1,2,3 corresponding to the following patterns:

  0: Plus/Minus Unit Directions (default)

  1: Regular Simplex

  2: Plus Unit Directions and -1 Vector

  3: Plus/Minus Shifted Regular Simplex

  If the 'pattern size' is bigger than the minimum required, additional search directions are generated by randomly combining the 'base' directions. For example, a 'random direction' with pattern type 0 might be $[-101]^T$. Only pattern type 0 is valid if bound constraints are used. Default: 0

- `-s, --search=NUM`

  Specify the total number of search directions. Default: minimum size needed for pattern type

- `-a, --active=NUM`

  Specify the minimum number of search directions that must be actively searched. Default: minimum size needed for pattern type

- `--break[=BOOL], --nobreak`

  Specifies whether or not function evaluations can be terminated before they are complete in the event that a better point comes along. Default: true (PVM) / false (MPI & other)

- `--reassign[=BOOL], --noreassign`

  Allow active searches to be reassigned when a new host is added. Default: true

- `-d, --debug=NUM`

  Determines how much information is printed where NUM is one of:

  0: Output final answer

  1: And new minimums and no spawn

  2: And status information on agents, cache, etc. (Default)

  3: And function return information

  4: And cache, no go, and insufficient decrease notices

  5: And convergence messages

  6: And contract, merge, and switch messages

  7: And function evaluation start and break messages

  8: And detailed job map calculations

  9: And positive basis convergence calculations

  10: And #2 for all agents

- `--profile=NUM`

  Where NUM is 0 (on) or 1 (off). For each individual search direction, profiling will print out the wall clock time, the wall clock idle time, the total number of function evaluations, the number of successful and failed function evaluations, and the number of aborted function evaluations. Default: 1.

- `--precision=NUM`

  Specify the output precision for real numbers. Default: 2.

- `--inc[=BOOL], --noinc`

  Specifies whether or not the step length can increase after multiple successful steps in the same direction. Default: true.

- `--contract=REAL`

  Specify the amount by which the step length is contracted during a contraction. Must be between 0 and 1. Default: 0.5.

- `--step=REAL`

  Specify the initial step length. Default: 1.0.

- `--min=REAL`

  Specify the minimum step length after a successful iteration. Default: twice the convergence tolerance.

- `--max=REAL`

  Specify the maximum allowable step length. Default: 1.0e+5.

- `--intw=NUM`

  Specify output width for integer search ids. Default: 2.

- `--hexw=NUM`

  Specify the output width for hex values used in host and taskids. Default: 8.

- `--nnls=REAL`

  Specify the tolerance for NNLS positive basis test. Default: 1.0e-8.

- `--eps=REAL`

  Specify the maximum of the machine epsilons for all machines participating in the computation. Default: 1.0e-8.

B Running APPS from the PVM Console

Type `pvm` to launch the PVM console. At the prompt type:

```
pvm> setopt PvmShowTids 0
```

to turn off display of the TID number with output. Then, to run APPS, type

```
pvm> spawn -> apps_pvm_generic [$OPTIONS] $PARAMS
```

To spawn APPS on a particular host, type

```
pvm> spawn -HOST -> apps_pvm_generic [$OPTIONS] $PARAMS
```

Type `help` in the PVM console for more options.

Warning: The PVM console occasionally core dumps when there is lots and lots of output. If this happens to you, try running from the command line instead.

C APPS Timings

In this section we present some numerical results on the performance of APPS. The test problems used are the six four-dimensional test problems from Example 3; see §6.3. The test problems were run on a cluster of quad 200 MHz Intel Pentium Pro processors, using a convergence tolerance of 10^{-3} and a stall factor of 200.

Because APPS is an asynchronous algorithm, the path to the solution may be different every time, even if the starting conditions are the same. Therefore we ran each problem 25 times and averaged the results. The same tests were performed with APPS-PVM in multi-agent mode, APPS-PVM in single-agent mode, and APPS-MPI. For the MPI tests, MPI/Pro (see Appendix D), a commercially available version of MPI, was used.

To demonstrate the difference between APPS's performance with and without the cache, we ran these tests both without the cache (Table 3) and with the cache (Table 4) In Tables 3 and 4, "Procs" refers to the number of processors (the same as the number of search directions) the problem was run on. "Total Time" refers to the longest time, in seconds, required for any search direction to finish. "Function Evaluations" is the average number of function evaluations, not counting cached function values, performed per search direction.

Table 5 gives information about the number of times a cached function value was used and the amount of time required for a cache lookup, successful or not. "% Reuses" is the reduction, as a percentage, in the number of function evaluations APPS had to perform. "Avg Lookup Time" is the average time required by the cache, including communication between hosts, to perform a lookup in the cache.

Because of the way the runs were timed, timings for MPI do not include the time needed to start all the processes before the program begins running (usually about one second). Timings for PVM *do* reflect the time needed to spawn all processes, since the spawning takes place after APPS-PVM has started.

D Obtaining MPI

There are several free implementations of MPI available for download. We have used two of these, LAM (Local Area Multicomputer), and MPICH (MPI Chameleon).

LAM can be downloaded from:
http://www.mpi.nd.edu/lam/

MPICH can be downloaded from:
http://www-unix.mcs.anl.gov/mpi/mpich/index.html

There are also commercial implementations of MPI. The one we have used is MPI/Pro. More information about MPI/Pro can be found at:
http://www.mpi-softtech.com/

Problem Name	Procs	Total Time			Function Evals		
		PVM	PVM Single	MPI/ Pro	PVM	PVM-S Single	MPI/ Pro
broyden2a	8	56.00	44.14	51.85	45.30	40.46	40.14
	16	39.75	35.84	38.98	35.14	31.35	32.18
	24	46.34	34.99	33.64	41.96	30.98	27.68
broyden2b	8	55.69	44.59	52.02	45.21	41.29	39.24
	16	39.65	34.28	38.79	35.86	31.35	31.66
	24	45.41	33.61	34.14	41.01	30.05	27.85
chebyquad	8	59.73	54.91	59.80	55.05	51.14	47.66
	16	51.18	38.20	49.38	47.16	35.27	40.70
	24	41.52	36.42	40.71	37.78	32.74	33.67
epowell	8	287.10	254.08	281.08	255.79	247.95	225.30
	16	172.08	164.94	160.32	164.76	154.86	132.27
	24	187.42	138.84	76.95	179.46	127.13	64.20
toint_trig	8	57.91	49.70	55.96	50.01	46.72	44.52
	16	47.46	41.65	48.10	43.69	38.40	39.58
	24	46.23	41.70	41.72	42.17	37.68	34.55
vardim	8	174.12	140.17	150.99	145.35	131.17	121.15
	16	37.16	35.12	89.25	34.19	32.16	73.11
	24	48.19	28.46	32.48	44.19	25.46	26.70

Table 3: Results on a collection of four-dimensional test problems, without the function value cache.

Problem Name	Procs	Total Time			Function Evals		
		PVM	PVM Single	MPI/ Pro	PVM	PVM Single	MPI/ Pro
broyden2a	8	41.45	38.14	48.19	43.21	39.79	34.28
	16	35.80	34.38	40.06	35.87	33.14	29.37
	24	40.06	32.08	33.30	40.02	30.08	24.99
broyden2b	8	39.84	37.66	47.41	41.42	38.92	33.83
	16	34.30	33.54	39.39	34.78	32.59	28.72
	24	40.40	31.70	32.59	40.45	29.94	24.51
chebyquad	8	52.01	48.48	57.01	56.28	51.81	39.43
	16	49.10	42.70	45.44	51.77	43.31	32.94
	24	43.38	38.68	40.56	45.53	38.59	30.55
epowell	8	289.52	251.71	359.96	315.08	273.05	257.26
	16	241.64	218.17	218.24	253.14	216.75	162.99
	24	183.45	211.16	72.03	193.96	205.27	54.01
toint_trig	8	47.46	45.56	53.59	50.50	48.49	38.67
	16	43.84	41.10	48.74	44.95	40.67	35.79
	24	40.86	40.46	41.26	42.18	39.28	30.65
vardim	8	153.56	139.23	128.33	169.98	152.50	92.22
	16	52.90	49.46	81.92	55.25	50.16	60.27
	24	56.84	36.84	36.30	59.72	35.54	26.73

Table 4: Results on a collection of four-dimensional test problems with the function value cache.

Problem Name	Procs	% Reuses			Avg Lookup Time		
		PVM	PVM Single	MPI/ Pro	PVM	PVM Single	MPI/ Pro
broyden2a	8	14.9	14.4	17.0	$2.4 \cdot 10^{-2}$	$2.6 \cdot 10^{-2}$	$4.1 \cdot 10^{-2}$
	16	12.1	11.6	12.4	$4.1 \cdot 10^{-2}$	$3.1 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$
	24	11.3	12.3	11.5	$4.9 \cdot 10^{-2}$	$2.6 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$
broyden2b	8	14.6	14.2	15.7	$2.6 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	$4.4 \cdot 10^{-2}$
	16	12.9	11.4	12.8	$4.1 \cdot 10^{-2}$	$3.2 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$
	24	11.3	12.0	11.6	$4.9 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$
chebyquad	8	18.4	16.7	20.3	$2.5 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	$4.2 \cdot 10^{-2}$
	16	15.4	14.7	15.6	$4.2 \cdot 10^{-2}$	$3.2 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$
	24	15.4	15.3	15.0	$4.4 \cdot 10^{-2}$	$2.5 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$
epowell	8	16.1	15.5	16.0	$2.5 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	$4.3 \cdot 10^{-2}$
	16	12.5	11.9	14.5	$4.5 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$
	24	13.3	12.9	13.1	$5.7 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$
toint_trig	8	15.2	15.9	15.9	$2.7 \cdot 10^{-2}$	$2.7 \cdot 10^{-2}$	$4.3 \cdot 10^{-2}$
	16	12.5	12.1	14.1	$4.3 \cdot 10^{-2}$	$3.2 \cdot 10^{-2}$	$2.9 \cdot 10^{-2}$
	24	13.6	13.6	13.9	$4.4 \cdot 10^{-2}$	$2.6 \cdot 10^{-2}$	$2.3 \cdot 10^{-2}$
vardim	8	17.2	16.3	17.8	$2.7 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	$4.3 \cdot 10^{-2}$
	16	13.4	14.5	15.6	$4.3 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$	$3.0 \cdot 10^{-2}$
	24	14.7	13.0	12.1	$4.5 \cdot 10^{-2}$	$2.5 \cdot 10^{-2}$	$2.4 \cdot 10^{-2}$

Table 5: Performance of the function value cache on a collection of four-dimensional test problems.

References

[1] T. H. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, The MIT Press, 1997.

[2] A. Geist, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine*, The MIT Press, 1994.

[3] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing, 22 (1996), pp. 789–828.

[4] W. D. Gropp and E. Lusk, *User's Guide for* `mpich`*, a Portable Implementation of MPI*, Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[5] P. D. Hough, T. G. Kolda, and V. J. Torczon, *Asynchronous parallel pattern search for nonlinear optimization*, Tech. Rep. SAND2000-8213, Sandia National Laboratories, Livermore, CA, January 2000. Submitted to SISC.

[6] MPI Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997. http://www-unix.mcs.anl.gov/mpi/index.html.

[7] Ohio Supercomputer Center, *MPI Primer / Developing with LAM*, November 1996.

[8] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufman Publishers, Inc., 1997.

[9] D. Sleator and R. Tarjan, *Self-adjusting binary search trees*, J. ACM, 32 (1985), pp. 652–686.