Algorithm XXX: VTDIRECT95: Serial and Parallel Codes for the Global Optimization Algorithm DIRECT

JIAN HE, LAYNE T. WATSON
Virginia Polytechnic Institute and State University
and
MASHA SOSONKINA
Ames Laboratory

VTDIRECT95 is a Fortran 95 implementation of D. R. Jones' deterministic global optimization algorithm called DIRECT, which is widely used in multidisciplinary engineering design, biological science, and physical science applications. The package includes both a serial code and a data-distributed massively parallel code for different problem scales and optimization (exploration vs. exploitation) goals. Dynamic data structures are used to organize local data, handle unpredictable memory requirements, reduce the memory usage, and share the data across multiple processors. The parallel code employs a multilevel functional and data parallelism to boost concurrency and mitigate the data dependency, thus improving the load balancing and scalability. In addition, checkpointing features are integrated into both versions to provide fault tolerance and hot restarts. Important algorithm modifications and design considerations are discussed regarding data structures, parallel schemes, error handling, and portability. Using several benchmark functions and real-world applications, the software is evaluated on different systems in terms of optimization effectiveness, data structure efficiency, parallel performance, and checkpointing overhead. The package organization and usage are also described in detail.

Categories and Subject Descriptors: J.2 [**Computer Applications**]: Physical Science and Engineering — *Mathematics*; G.4 [**Mathematics of Computing**]: Mathematical Software
General Terms: Algorithms, Design, Documentation
Additional Key Words and Phrases: DIRECT, global optimization, data structures, parallel schemes, checkpointing

This work was supported in part by Air Force Research Laboratory Grant F30602-01-2-0572, NSF Grants DMI-0422719 and DMI-0355391, Department of Energy Grant DE-FG02-06ER25720, and NIGMS/NIH Grant 5 R01 GM078989.
Authors' addresses: J. He, L. T. Watson, Departments of Computer Science and Mathematics, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061-0106, {jihe, ltw}@cs.vt.edu; M. Sosonkina, Ames Laboratory, Iowa State University, Ames, IA 50011, masha@scl.ameslab.gov.

1. INTRODUCTION

VTDIRECT95 is a FORTRAN 95 software package consisting of a dynamic data structure based serial implementation and a data-distributed massively parallel implementation of the DIRECT algorithm by Jones et al. [1993]. Jones et al. [1993] invented DIRECT (DIviding-RECTangles) as a Lipschitzian direct search algorithm for solving global optimization problems (Horst et al. [2000], Horst and Tuy [1996], Pinter [1996]) subject to bound constraints of the form

$$\min_{x \in D} f(x), \tag{1.1}$$

where $D = \left\{ x \in E^n \mid \ell \leq x \leq u \right\}$ is a bounded box in n-dimensional Euclidean space E^n, and $f : E^n \to E$ must satisfy a Lipschitz condition

$$|f(x_1) - f(x_2)| \leq L\|x_1 - x_2\|, \quad \forall\, x_1, x_2 \in D. \tag{1.2}$$

Although DIRECT can be used for local optimization, it was designed as an effective global method that avoids being trapped at local optima and intelligently explores "potentially optimal" regions to converge globally for Lipschitz continuous optimization problems. As a direct pattern search method, DIRECT produces deterministic results and is straightforward to apply without derivative information or the Lipschitz constant of the objective function. It has been used successfully in many multidisciplinary design optimization problems such as high speed civil transport aircraft design (Baker et al. [2000]), pipeline design (Carter et al. [2001]), aircraft routing (Bartholomew-Biggs et al. [2003]), surface optimization (Zhu et al. [2002]), wireless communication transmitter placement (He et al. [2004]), molecular genetic mapping (Ljungberg et al. [2004]), and cell cycle modeling (Zwolak et al. [2005] and Panning et al. [2006]).

Many global optimization problems require both supercomputing power and a great amount of memory to store intermediate data. For example, the parameter estimation problem for the budding yeast cell cycle has 143 parameters with 36 stiff ordinary differential equations. A reasonable solution entails tens of thousands of function evaluations, requiring days to weeks of computation on a single processor. This type of application motivated the massively parallel implementation in VTDIRECT95, which also distributes data among processors to share the memory burden imposed by such high dimensional problems.

Previous serial and parallel DIRECT implementations in the public domain include a FORTRAN 77 implementation by Gablonsky [2001] and a FORTRAN 90 implementation by Watson et al. [2001]. The data structures that they employ are static, thus inducing inefficiencies in handling an unpredictable memory requirement due to different problem structures and the nature of DIRECT's exploratory strategy. A pure master-slave paradigm is adopted in Gablonsky [2001] with no further enhancement for load balancing. Taking a step forward, Watson et al. [2001] designed dynamic load balancing schemes for a distributed control version of DIRECT. However, other load balancing issues such as a single starting point and a distributed data structure were not considered by Watson et al. [2001].

The design considerations absent from these earlier attempts have been addressed in VTDIRECT95. Advanced features (derived data types, pointers, dynamic memory allocation, etc.) supplied by FORTRAN 95 were used to design dynamic data structures that flexibly organize the data on a single machine, effectively reduce the local data storage, and efficiently share the data across multiple processors. Moreover, a multilevel functional and data parallelism is proposed to produce multiple starting points, mitigate the data dependency, and improve the load balancing. In addition, both the serial and parallel programs are equipped with checkpointing features to provide fault tolerance to power outage or hardware/memory failures, and enable hot restarts for large runs.

The paper is organized as follows. Section 2 reviews the DIRECT algorithm and the major algorithmic modifications that increase the program concurrency for the parallel implementation, tailor execution for different problem properties and optimization objectives, and offer more choices in stopping conditions. Section 3 outlines important design considerations and implementation details that potential users may be most interested in. Performance results for several artificial benchmark functions and real-world problems on two different parallel systems are presented in Section 4. Package organization and usage are explained in detail in Section 5.

2. DIRECT AND MODIFICATIONS

The DIRECT search is carried out through three essential operations: region selection, point sampling, and space division. Jones et al. [1993] describe the original algorithm in six detailed steps, which are regrouped and relabeled to highlight the basic operations as below.

Given an objective function f and the feasible set D, the steps are:

1. **Initialization.** Normalize the feasible set D to be the unit hypercube. Sample the center point c_i of this hypercube and evaluate $f(c_i)$. Initialize $f_{\min} = f(c_i)$, evaluation counter $m = 1$, and iteration counter $t = 0$.
2. **Selection.** Identify the set S of "potentially optimal" boxes that are subregions of D. A box is potentially optimal if, for some Lipschitz constant, the function value within the box is potentially smaller than that in any other box (a formal definition with parameter ϵ is given by Jones et al. [1993].)
3. **Sampling.** For any box $j \in S$, identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length. Sample the function at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the box and e_i is the ith unit vector.
4. **Division.** Divide the box j containing c into thirds along the dimensions in I, starting with the dimension with the lowest value of $w_i = \min\{f(c+\delta e_i), f(c-\delta e_i)\}$, and continuing to the dimension with the highest w_i. Update $f_{\min}$ and m.
5. **Iteration.** Set $S = S - \{j\}$. If $S \neq \emptyset$ go to **3**.

6. **Termination.** Set $t = t + 1$. If iteration limit or evaluation limit has been reached, stop. Otherwise, go to **2**.

A few modifications were made in VTDIRECT95 to meet the needs of various applications and improve the performance on large scale parallel systems. For **Initialization**, an optional *domain decomposition* step is added to create multiple subdomains, each with a starting point for a DIRECT search. Empirical results have shown that this approach significantly improves load balancing among a large number of processors, and likely shortens the optimization process for problems with asymmetric or irregular structures. The second step **Selection** has two additions. The first is an "aggressive" switch adopted from Watson et al. [2001], which generates more function evaluation tasks that may help balance the workload under the parallel environment. The second is an adjustable ϵ, which is recommended by Jones et al. [1993] to be within $(10^{-2}, 10^{-7})$, and "most naturally" 10^{-4} or the desired solution accuracy. The studies by Gablonsky [2001] show that $\epsilon = 0.0$ speeds up the convergence for low dimensional problems. In general, smaller ϵ values make the search more local and generate more function evaluation tasks. On the other hand, larger ϵ values bias the search toward broader exploration, exhibiting slower convergence. The value of ϵ is taken as zero by default, but can be specified by the user depending on problem characteristics and optimization goals.

To produce more tasks in parallel, new points are sampled around all boxes in S along their longest dimensions during **Sampling**. This modification also removes the step **Iteration**, thus simplifying the loop. In the serial version, **Sampling** samples one box at a time to eliminate unnecessary storage for new boxes. Another modification to **Sampling** is adding lexicographical order comparison between box center coordinates in both the serial and parallel versions. Since box center function values may be the same or very close to each other, the parallel **Sampling** may yield a different box sequence in each box column as the parallel scheme varies. As a consequence, boxes will be subdivided in a different order, thus destroying the deterministic property of DIRECT. Hence, lexicographical order comparison is added to keep the boxes in the same sampling sequence on the same platform. Unfortunately, the deterministic property is hard to preserve across machines/compilers that produce different numerical values, so the numerical results for the same problem may vary slightly on different systems.

The last set of modifications, in **Termination**, is to offer more choices of stopping conditions. Jones et al. [1993] commented that the original stopping condition on a limit on iterations `MAX_ITER` or evaluations `MAX_EVL` is not convincing for many optimization problems. Two new stopping rules proposed in VTDIRECT95 are (1) minimum diameter `MIN_DIA` (exit when the diameter of the best box has reached the value specified by the user or the round off level) and (2) objective function convergence tolerance `OBJ_CONV` (exit when the relative change in the optimum objective function value has reached the given value).

3. DESIGN AND IMPLEMENTATION

One of the biggest design challenges of DIRECT is to break the "curse of dimensionality" first noted by its creators Jones et al. [1993]. The approach here is to design dynamic data structures that are easily extensible to store continuously generated data from point sampling/space division and to share the data storage among multiple processors. A detailed discussion of data structures for DIRECT appears in He et al. [2002]. Section 3.1 covers the data structure design at a high level and illustrates advanced features such as a memory reduction technique. Section 3.2 presents the parallel schemes that focus on **Selection** and **Sampling**, which are dependent on each other, but can proceed as individual steps. As an indispensable part of implementation, error handling and recovery features are discussed in Sections 3.3 and 3.4. Portability issues are discussed in the last section.

3.1 Data Structures

DIRECT keeps subdividing the design space and selects potentially optimal regions according to the sampling results. The divided subregions are called "boxes". The key information for a box is stored in a derived data type `HyperBox`, containing an array of center point coordinates c, an array of box sides $side$, center point function value val, and box diameter $diam$. To identify potentially optional boxes, all the boxes are organized ideally by the center function values and box diameters as shown in Figure 3.1, where the vertical sequences of boxes are called "box columns", which are sorted in the order of box diameters, while the boxes in each box column are sorted in the order of center function values. Jones et al. [1993] have proved that the potentially optimal boxes are those on the lower right convex hull of the scatter plot shown in Figure 3.1, so they are also called "convex hull boxes" in this paper. When $\epsilon > 0$, a line starting from $f^* = f_{\min} - \epsilon|f_{\min}|$ on the vertical axis of function values will screen out the boxes that may lead to insignificant improvement.

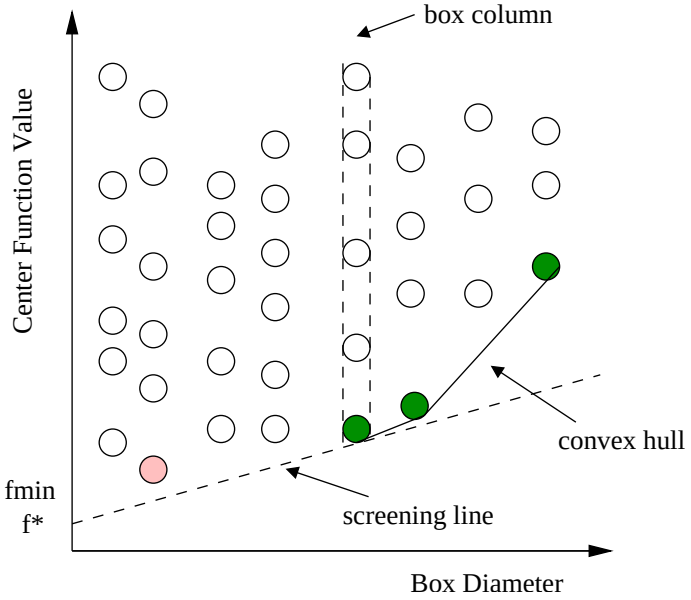


Fig. 3.1. An example of a box scatter plot.

The horizontal strict order for the box diameters must be maintained to facilitate the convex hull computation. However, the vertical strict order of box center function values (implemented in the serial version described in He et al. [2002]) is unnecessary for each box column, and it also incurs more operational cost such as shifting and sorting for box removal and insertion. In VTDIRECT95, a min-heap data structure implements a box column, so that the lowest box owns the smallest function value and every box has a smaller function value than its left and right children if they exist. It lays out the potentially optimal box candidate at the bottom of the scatter pattern. Once this candidate box is determined to be potentially optimal, it will be removed from the heap to be subdivided, and the last box in the heap will be put to the first position and sifted down, reordering the heap in $\mathcal{O}(\log n)$ operations instead of the $\mathcal{O}(n)$ shifting operations required by a strictly increasing order of center function values. Similarly for box insertion, a new box is inserted at the end of the heap and sifted up in $\mathcal{O}(\log n)$, reduced from $\mathcal{O}(n)$, comparisons. The complexity is improved considerably, especially for large box columns.

In addition to `HyperBox`, `BoxMatrix` and `BoxLink` are the other two derived data types for storing boxes. These three are called "box structures" in He et al. [2002]. `BoxMatrix` contains the following components:

(1) a two-dimensional array `M` of type `HyperBox`,
(2) an array of box counters `ind` for box columns,
(3) a pointer `child` that points to the next linked node of `BoxMatrix` needed when more box columns with new diameters are generated,
(4) an array `sibling` of pointers that point to linked nodes of type `BoxLink`, which are used to extend box columns beyond the storage afforded in `M`, and
(5) `id` to identify this box matrix among others.

Initially, a box matrix is allocated with empty box column structures (called "free box columns") according to the problem dimension and optimization scale. When currently allocated memory for a box column has been filled up, a new box link of derived type `BoxLink` is allocated dynamically adding a one-dimensional array of `HyperBox`es and associated components such as counter, pointers, and ID to extend the box column.

To maintain the strict order of box diameters for box columns, another set of "linked list structures" organizes box columns and recycles free box columns. The linked lists `setFcol` and `setInd` are of the type `int_vector`, containing a one-dimensional array *elements* of integers, an array *flags* for marking convex hull boxes, pointers for linking nodes, and the node ID. The array *flags* is only allocated and used for `setInd`. The third linked list `setDia` is derived from `real_vector` to hold an array of box diameters (real values), pointers, and ID. When a new box matrix is allocated, all the global IDs of its free box columns are computed based on its box matrix ID and inserted in `setFcol`. When a new box diameter is produced from **Division**, a free box column from `setFcol` is assigned to hold the box. An appropriate position for this box diameter will be found using a binary search in `setDia`, which is sorted in decreasing order. Then, the global ID of the newly

assigned box column is added in `setInd` at the corresponding position to that in `setDia`. The process is reversed when a box diameter disappears after the last box with this box diameter has been removed, so this box column becomes free. Subsequently, its diameter value and the global ID will be taken out from `setDia` and `setInd`, respectively. Finally, the free box column is recycled back to `setFcol` for later use.

Another practical issue is the sequence of dividing convex hull boxes. Because diameters are sorted in decreasing order in `setDia`, the serial version needs to start from the end of `setDia` and subdivide the box with the smallest diameter first, so that sifting up newly generated boxes would not override any existing convex hull boxes. Obviously, this potential overriding problem does not exist for the parallel version, which buffers all new boxes from subdivided convex hull boxes and inserts them all at one time. Hence, the parallel version starts from the beginning of `setDia`, thus avoiding the unnecessary cost of chasing the linked nodes of `setDia`.

VTDIRECT95 adds a new feature to output multiple best boxes (MBB), which are found by searching through the box structures that hold all the information on the space partition. This feature is very useful for global optimization problems with complex structures, where local optimum points are far away from each other, thus demanding a large amount of space exploration and slowing convergence. In such a case, DIRECT is often used as a global starter to find good regions. Then, a local optimizer is applied to each region to efficiently find multiple local optimum points. Examples of this are presented in Zwolak et al. [2005] and Panning et al. [2006]. To activate the MBB option, an empty array `BOX_SET` of type `HyperBox` allocated with a user-desired size needs to be specified in the input argument list. Optionally, two more arguments `MIN_SEP` (minimal separation) and `W` (weights) can be given to specify the minimal weighted distance between the center points of the best boxes returned in `BOX_SET`. By default, `MIN_SEP` is half the diameter of the design space and `W` is taken as all ones. When the desired number of best boxes can not be found conditioned on `MIN_SEP` and `W`, the output argument `NUM_BOX` is returned as the actual number of best boxes in `BOX_SET`. The following pseudo code illustrates the MBB process. For interested readers, it also demonstrates a typical scenario of VTDIRECT95 manipulating the dynamic data structures.

---

cc: the current best box center
c_b: the counter for best boxes stored in `BOX_SET`
c_m: the counter for marked boxes
$f_{\min}$: minimum function value
f_i: the current function value to be compared
i, j, k: loop counters
n_b: the desired number of best boxes
n_c: the number of columns allocated in M
n_e: the number of function evaluations
n_r: the number of rows allocated in M
p_b: the pointer to a box matrix

p_c: the pointer to a box
p_l: the pointer to a box link
sep: weighted separation between *cc* and a candidate box
x_0: minimizing vector scaled in the original design space
x_1: normalized x_0 in the unit design space

Store the first best box centered at x_0 with value $f_{\min}$;
Initialize *cc* and f_i
$c_b := 1$
`BOX_SET`(c_b)%`val` := $f_{\min}$
`BOX_SET`(c_b)%`c` := x_0
$cc := x_1$
$f_i :=$ a very large value
OUTER: **do** $k := 1$, $n_b - 1$
  Initialize p_b to point to the head of box matrices
  $c_m := 0$
  INNER1: **do while** (p_b is not NULL)
    INNER2: **do** $i := 1$, n_c
      INNER3: **do** $j := 1$, ((p_b%`ind`(i)-1) **mod** n_r) + 1
        Locate the best box with x_0 and $f_{\min}$ in the first pass.
        if ($k = 1$) **then**
          if (x_1 is the same as p_b%`M`(j, i)%`c`) **then**
            Found the best box and fill in `BOX_SET`;
            Assign the first best box with scaled p_b%`M`(j,i);
            Mark off the box at p_b%`M`(j,i);
            $c_m := c_m + 1$
            cycle
          end if
        end if
        if (box at p_b%`M`(j,i) is not marked) **then**
          Compute *sep*
          if (*sep* < `MIN_SEP`) **then**
            mark off the box at p_b%`M`(j,i)
            $c_m := c_m + 1$
          else
            if (box at p_b%`M`(j,i)%`val`< f_i)
              $f_i := p_b$%`M`(j,i)%`val`
              p_c points to the box at p_b%`M`(j,i)
            end if
          end if
        else
          $c_m := c_m + 1$
        end if
      end do INNER3

```
      if (any box link exists for this box column) then
        p_l points to the first box link
        do while (p_l is not NULL)
          Repeat above steps in INNER3 loop for all boxes in p_l
          p_l points to the next box link
        end do
      end if
    end do INNER2
    p_b points to the next box matrix
  end do INNER1
  if (p_c is not NULL) then
    if (p_c is not marked) then
      Found the next best box at p_c; Scale it back to the original design
      space and store it in BOX_SET
      c_b := c_b + 1
      BOX_SET(c_b) := scaled box at p_c
      Mark it off
      c_m := c_m + 1
      Update cc
      cc := p_c%c
    end if
  else
    exit OUTER since the next best box is not available
  end if
  Exit when all evaluated boxes have been marked
  if (c_m ≥ n_e ) exit OUTER
end do OUTER
```

Pseudocode 3.1.

The MBB option is available for both serial and parallel versions except for parallel runs with multiple masters, because the communication and computation complexity of implementing MBB across multiple processors is fairly high. Also, the problem scale of locating good regions is usually much smaller than finding the global optimum, so a single master should be able to hold all the information for box subdivision.

To reduce the memory requirement, VTDIRECT95 is enhanced with the limiting box columns (LBC) technique. Recall that every iteration, DIRECT divides at most one box from each box column, thus each box column only needs to have at most $L = I_{\max} - I_c + 1$ boxes with the smallest function values, where $I_{\max}$ is the iteration limit and I_c is the current iteration number. With LBC, box columns are scanned to be squeezed to length L after all convex hull boxes are subdivided and all new boxes are inserted. Although the extra operations of removing boxes with the largest function values are expensive (deleting the box with the largest value in a min-heap has $\mathcal{O}(n)$ complexity), the memory requirement is reduced greatly as

a result. Therefore, it is highly recommended to enable LBC for large scale/high dimensional problems that more likely encounter memory allocation failures than small scale/low dimensional problems.

LBC is enabled under three conditions: (1) the specified iteration limit $I_{\max}$ (`MAX_ITER` in the code) is positive, (2) the evaluation limit L_e (`MAX_EVL` in the code) is not specified or is sufficiently large—$L_e \times (2N + 2) > 2 \times 10^6$, and (3) the MBB option is off. Without Condition (1), LBC would not be able to decide on the number of boxes to remove. Condition (2) is to turn off LBC to save operations for small scale runs with little concern for box storage; 2×10^6 is the threshold obtained from an empirical study. The last condition is also necessary since the MBB process demands that all boxes stay in the memory.

3.2 Parallel Schemes

The functional flow of DIRECT exposes its inherent sequential nature as seen in Section 2. The data dependency among the algorithm steps suggests multilevel parallelism for **Selection** and **Sampling**. The parallel scheme for **Selection** concentrates on distributing data among multiple masters to share the memory burden. Moreover, the data-distributed scheme naturally parallelizes the convex hull computation by merging multiple local convex hulls to a global one. Differently for **Sampling**, functional parallelism distributes function evaluation tasks to workers. Nevertheless, function evaluations should be computed locally on masters if the evaluation cost is cheaper than the communication round trip cost. This is called the "horizontal scheme" (multiple masters, no workers) to contrast with the "vertical scheme" (one master and multiple workers). He et al. [2007a] and He et al. [2007b] present thorough performance studies on different parallel schemes under various problem configurations and computing systems. Here, the parallel schemes for **Selection** and **Sampling** are described based on assumed reasonable problem and system parameters under normal circumstances.

The overall hierarchy of the parallel scheme is shown in Figure 3.2. On the top level, n subdomain masters (SMs) are grouped for each of m subdomain (SDs) to collaborate on **Selection**, update intermediate results, and detect stopping conditions in parallel. On the bottom, k workers (Ws) are shared in a global pool to request function evaluation tasks from all the subdomain masters to accomplish **Sampling**. SD_i denotes subdomain i, $SM_{i,j}$ denotes subdomain master j in SD_i, and W_k is the worker k that works for all the SMs in "active" SDs. When a SD finishes all its work, it becomes inactive. SD_1 is called the "root" SD. When a nonroot SD becomes inactive, and at least one SD is still active, $SM_{1,1}$ will send a message to convert all SMs in that inactive SD to workers that are going to perform function evaluation tasks for the remaining active SDs.

Standard MPI library functions are called to group, synchronize, and communicate between the involved processors in their different roles. Because any MPI-based execution only needs to be initialized (`MPI_INIT()`) and finalized (`MPI_FINALIZE()`) once, two separate subroutines encapsulate these two MPI function calls, so that users have an option to exclude the latter to avoid conflicting with existing MPI

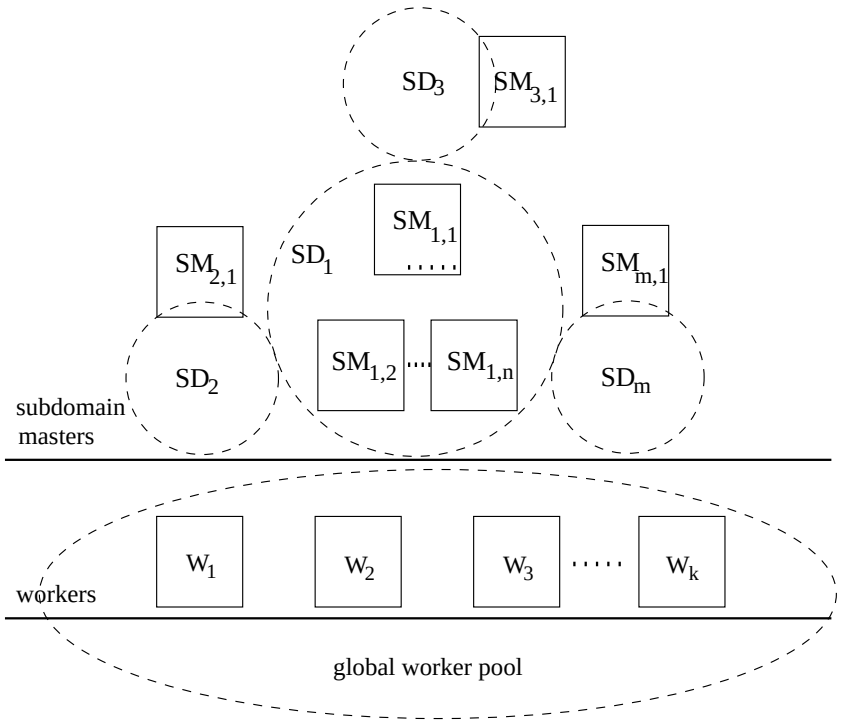


Fig. 3.2. The parallel scheme.

initialization and finalization calls in their local parallel environment. To ease the required collective communication among SMs in the same SD, sub-communicators are created at the beginning. Depending on a processor's global rank and specified scheme parameters (i.e., n and m), the processor is assigned a role as a master or a worker and executes the corresponding part of the code. A master has a global rank that identifies it among all processors, and a local rank that establishes it as the root or a nonroot SM for that SD.

When multiple SDs ($m > 1$) are used, the original feasible set delimited by upper (U_b) and lower (L_b) bounds is decomposed into m parts, each of which will be normalized to a unit box to start a DIRECT search. Theoretically, the original unscaled box is subdivided into $s = \sqrt{m}$ parts along the longest scaled dimension $D_1 = \max_i w_i(U_b - L_b)_i$, then each of these s boxes is subdivided into s boxes along the longest scaled dimension D_2 (the second longest overall). The $w_i > 0$ are user supplied component weights (dimension scalings), all one by default. In practice, s may not be an integer, so the decomposition needs to determine two reasonable divisors s_1 and s_2, where (1) $s_1 \times s_2 = m$ and (2) $s_1/s_2 \approx D_1/D_2$. The second condition on the ratio of divisors prevents the resulting subdomains from being out of proportion. For example, if $m = 12$, the acceptable divisors are (a) $s_1 = 12$, $s_2 = 1$, or (b) $s_1 = 6$, $s_2 = 2$, or (c) $s_1 = 4$, $s_2 = 3$. Whichever divisors best satisfy (2) are chosen, which best preserves the original weights on dimension bounds given by the user.

When multiple masters ($n > 1$) are used, the parallel **Selection** is implemented as follows in SD_i:

1. $SM_{i,j}$, $j = 1, \ldots, n$ identify local convex hull box sets $S_{i,j}$, $j = 1, \ldots, n$.
2. $SM_{i,1}$ gathers the $S_{i,j}$ from all the $SM_{i,j}$.
3. $SM_{i,1}$ merges the $S_{i,j}$ by box diameters and finds the global convex hull box set S_i.
4. All the $SM_{i,j}$ receive the global set S_i and find their portion of the convex hull boxes.

The above scheme takes advantage of the geometrical fact that a box on a global convex hull must be in the union of all the local convex hull box sets. The amount of communication is greatly reduced since $SM_{i,1}$ does not gather all the lowest boxes from each $SM_{i,j}$. Also, the convex hull computation is shared by all SMs. However, the costly synchronization and communication involved in such a scheme still behooves users to use as small a number of masters as possible. Depending on the potential memory requirement of a run, users can estimate the number of masters required to achieve a particular stopping condition. If the function evaluation cost is high, but the memory requirement is hard to assess, the checkpointing feature can be enabled to log evaluations on existing masters and recover the run with more masters if memory allocation failure occurs.

Following **Selection**, each master samples new points within its own portion of convex hull boxes and stores them in a buffer. If workers are not used, function values are computed locally on each master. Otherwise, parallel **Sampling** is carried out in the following manner, assuming $k \geq 2mn$ workers are in the pool so that there are at least two workers per master.

1. A worker W_i sends a "nonblocking" request to a randomly selected $SM_{i,j}$. (The term "nonblocking" means that the request receiving master will not block the worker in the queue).
2. $SM_{i,j}$ sends a task (if any) to a worker that is in the queue or that has sent a "nonblocking" request. Each task contains $N_b \leq$ `BINSIZE` number of points, where `BINSIZE` is an optional input argument taken as one by default. If $SM_{i,j}$ has no more tasks, it sends a "no point" message. If it has no more iterations, it sends an "all done" message.
3. If W_i receives a task to evaluate the objective function at some point(s), it sends back the function value(s). If it receives a "no point" message, it marks $SM_{i,j}$ as idle and checks with other masters that may have tasks, and if none have tasks, sends a "blocking" request and waits. If W_i receives an "all done" message, it removes $SM_{i,j}$ from the master list and checks with the remaining masters, if all reply "all done", it terminates.
4. If $SM_{i,j}$ receives the function values back, it puts them in the buffer and sends another task. If it has no more tasks, it sends a "no point" message. If it receives a "blocking" request, when multiple subdomains are involved, $SM_{i,j}$ tracks the number of "blocking" requests from this worker during this iteration; it sends a "no point" message again if this is the first "blocking" request from this worker, or blocks this worker in the queue if the worker has sent two "blocking" requests during this cycle. For a single domain, the

feature of tracking "blocking" requests is disabled so that $SM_{i,j}$ queues up the worker upon each "blocking" request.

The workers are shared by all masters no matter how many subdomains exist. The random master selection gives every master in each subdomain a fair chance of being served by workers. If the problem structure in a particular subdomain yields more tasks than others, workers will be dynamically appointed more often for that subdomain. Also observe that the masters in different subdomains work independently, meaning no communication or synchronization is required, except for result merging and processor termination at the end. The asynchronous property of multiple subdomains improves load balancing among workers, who are more likely to obtain tasks than those with a single subdomain. Therefore, the feature of tracking "blocking" requests described in (4) is designed to encourage a worker to seek tasks one more time under the multiple subdomain scenario. The final point in **Sampling** concerns choosing a reasonable `BINSIZE`. It needs to be set greater than one to pack several points in a single task only if (1) there are an extremely large number of function evaluations, and (2) each evaluation costs less than the communication round trip. Otherwise, the load becomes imbalanced and communication overhead increases, thus degrading the parallel efficiency. It is recommended to compute function values locally on masters if condition (2) is satisfied, but condition (1) is not. He et al. [2007a] discuss this issue in great detail with convincing experimental results.

3.3 Error Handling

Program robustness requires error handling that anticipates, detects, and resolves errors at run time. The highest level of error handling capability is fault tolerance that attempts to recover from hardware or operating system failures if possible, and if not, terminates the program gracefully. The tradeoff for fault tolerance is increased program complexity. The errors encountered in using VTDIRECT95 come from several sources, including input parameters, memory allocation, files, MPI library, and hardware/power failure, etc. The error handling strategies here aim at balancing potential computation loss with implementation complexity. Therefore, simple fault tolerance features are considered only for recovering from some of the input parameter errors. The remaining errors are regarded as fatal errors, which are handled by checkpointing to save the computation as much as possible for later recovery.

Input parameter errors: Input parameter errors—for instance, the given lower bounds are not less than the upper bounds or none of the four stopping rules is specified—are recognized in the initialization phase. The function `sanitycheck` verifies all input parameters and assigns values to the derived local variables. Some input parameter errors are recoverable when the parameters are also in the output list. In this case, the default parameter values are set or the desired features are disabled, and the revised parameter values will be reported to the user upon return. Examples of such errors include nonpositive values of `MAX_ITER`, `MAX_EVL`, or `MIN_DIA` for stopping conditions. Also, if the box structures in (the subroutine

argument) BOX_SET are not allocated, the missing pointers are recovered by allocating them with the correct problem dimension. For an irrecoverable error, the error code is returned in STATUS, which is an integer in the serial version, or an array of integers to hold return statuses for all subdomains in the parallel version. All masters and workers will check the sanity of input parameters and handle such errors in the same way.

MPI errors: The MPI function calls in the parallel version may also return errors at run time. By default, any error that MPI encounters internally for the global communicator MPI_COMM_WORLD is set as MPI_ERRORS_ARE_FATAL whose default action aborts the entire program. In VTDIRECT95, MPI_ERRORS_RETURN is set in place of the default error handler to notify the user of errors during the initialization phase, and reset to the default one to reduce the overhead after all processors have passed the initialization. MPI_ALLTOALL is used to collect initialization status on each processor from all others. If a fatal error occurs on a subset of processors during initialization, every processor is notified that the initialization failed. Then, the program terminates gracefully with a defined error code. The fatal errors here include those related to MPI and also all the irrecoverable errors discussed previously.

Memory allocation errors: The next source of errors is memory allocation that aborts the program when the virtual memory is exhausted. The behavior of the program depends on the virtual memory management under a particular operating system. It may simply quit or may become intolerably slow because of the heavy disk paging. The ultimate solution for this type of error is checkpointing (see the next section).

3.4 Checkpointing

VTDIRECT95 adopts a user level and nontransparent checkpointing method that records/recovers function evaluation logs via file I/O. Plank [1997] categorizes such a method as "user level" and "nontransparent" because it is visible in the source code and is implemented outside the operating system without using any system level utilities. It requires more programming effort than simply applying system level transparent tools (e.g., MPICH-V by Bouteiller et al. [2006], FT-MPI by Fagg et al. [2001], LAM-based MPI-FT by Louca et al. [2000], or model based fault tolerance MPI middleware by Batchu et al. [2004]), but it is flexible and precise in choosing what to save, instead of dumping all the relevant program and even system data. Another drawback of using fault tolerance enhanced tools in a parallel program is the dependence on a particular implementation of the MPI standard. For MPI based programs, Gropp et al. [2004] also recommend "user-direct" checkpointing with which it is easier to extract all the necessary state information than with "system-direct" methods. In the present work, function data points $(x, f(x))$ are chosen as the checkpointing state information for both serial and parallel versions.

The checkpointing switch RESTART can be 0 ("off"), 1 ("saving"), or 2 ("recovery"). During checkpointing, the errors are mainly related to the file in the

process of opening, reading, writing, verifying the file header, or finding checkpoint logs. For "saving", the program will report an opening error if the default checkpoint file already exists, in order to prevent the saved checkpoint logs from being overwritten. Hence, an old checkpoint file should be either removed or renamed before starting another "saving" run. The opening error also occurs when "recovery" can not find the needed checkpoint file. Note that the checkpoint file has a fixed name (`vtdirchkpt.dat`) in the serial version, while in the parallel version, the file name on each master is tagged with its subdomain ID and master ID (i.e., `pvtdirchkpt.000.001` is saved by $SM_{1,2}$ in SD_1).

Each checkpoint file has a header containing important parameters that must be validated in the recovery run to ensure that **Sampling** will produce the same sequence of logs as in the file. For the serial version, the header includes the problem dimension, upper and lower bounds, ϵ, and the aggressive switch. Changing any of these parameters will result in different point sampling. However, other input parameters such as `MAX_ITER` or `BOX_SET` can be modified for the recovery run. Some applications may use checkpointing as a convenient probing tool to find a good stopping condition or a reasonable set of best boxes. In the parallel version, m (the number of subdomains) and n (the number of masters per subdomain) are added in the header. m must be the same in the recovery run, but n is permitted to be changed in order to adjust the number of masters. This makes it possible to recover a crashed run due to memory allocation failure.

In the serial code, a checkpoint log consists of the current iteration number t, a vector c of point coordinates, and the function value val at c. The "saving" run records each evaluation as a checkpoint log in the file. Assuming the computing platform is the same, the points are sampled in the same sequence for the same number of iterations/evaluations, because of the deterministic property of DIRECT. Therefore, the recovery run loads all the checkpoint logs, or those that are within the iteration limit if specified. These logs are stored in a list in the same order as in the file, and will be recovered in that order as the program progresses. Recall that in the serial version, **Sampling** samples around one convex hull box at a time, but in the parallel version, it samples around all the convex hull boxes to produce as much work for the workers as possible. As the serial program generates new points, N_t (the number of points at iteration t) is unknown. Therefore, t is required for each checkpoint log in the serial code. However, N_t is known under the parallel version. Hence, the checkpoint file has a different form for the parallel code—in addition to a file header, a subheader consisting of t and N_t is followed by N_t logs, each with c and val.

When the number of masters n is the same as the "saving" run, the recovery run proceeds on each master similarly as in the serial version. If n is changed, the masters in the recovery run read in the checkpoint logs from all the files generated by all the masters during the saving run. Since the total number of logs aggregated from all masters may become very large, the masters load the logs only for the current iteration. The original deterministic sequence on a single machine breaks

into pieces on multiple masters, so it is better to organize the logs for easy searching. In the present work, these logs are sorted in lexicographical order of the point coordinates, which are looked up using a binary search to retrieve the corresponding function values. When the checkpoint file is corrupted or is from a different platform, some point coordinates may be missing—a fatal error that aborts the recovery run.

3.5 Portability Issues

The module `REAL_PRECISION` from HOMPACK90 by Watson et al. [1997] is used to define real arithmetic for "precision portability" across different systems. In the `REAL_PRECISION` module, `R8` is the selected `KIND` value corresponding to real numbers with at least 13 decimal digits of precision, covering 60-bit (Cray) and 64-bit (IEEE 754 Standard) real arithmetic.

Another portability issue arises under the parallel computing environment. Although MPI is well known for its portability across machines, its latest standard has not proposed a portable way of matching the data types specified with Fortran 95 `KIND` values. The `REAL (KIND=R8)` real number may be considered as double precision on one system but as single precision on another system. This data type matching problem is addressed here by calling `INQUIRE` to obtain the byte size for the `R8` type and using `MPI_BYTE` to transfer a buffer holding `R8` values, assuming the same byte ordering on all the involved machines. No performance degradation has been observed for this approach.

4. PERFORMANCE

In this section, VTDIRECT95 is evaluated in terms of optimization effectiveness, data structure efficiency, parallel performance, and checkpointing overhead. The performance tests here focus on practical concerns and new features, summarizing important test results in the earlier performance studies by He et al. [2002], He et al. [2007a], and He et al. [2007b] to present the reader with a complete picture.

Five benchmark functions (also provided in the package) and two real-world applications are listed in Table 4.1. The problem dimension N and function evaluation cost T_e in seconds can be adjusted for benchmark functions to suit the different test purposes, while N and T_e are not adjustable for the real-world problems FE and BY. The problem FE has 16 parameters ($N = 16$) and costs about 3 seconds ($T_e \approx 3.0$) per function evaluation. For the problem BY, $N = 143$ and $T_e \approx 11.0$. Located at Virginia Tech, an Apple Xserve G5-based system (System X) with 2200 processors and an AMD Opteron-based system (Anantham) with 400 processors are used in the studies.

4.1 Optimization Effectiveness

The convergence speed is considered here for measuring the optimization effectiveness. It certainly depends on the problem structure, but the parameter ϵ also plays an important role as reported by Jones et al. [1993], Finkel et al. [2004], and

Table 4.1. Test functions.

Name	Description
GR	Griewank: $f = 1 + \sum_{i=1}^{N} {x_i}^2/500 - \prod_{i=1}^{N} \cos(x_i/\sqrt{i})$), $-20.0 \le x_i \le 30.0$, $f(0,\ldots,0) = 0.0$
QU	Quartic: $f = \sum_{i=1}^{N} 2.2 \times (x_i + 0.3)^2 - (x_i - 0.3)^4$, $-2.0 \le x_i \le 3.0$, $f(3,\ldots,3) = -29.816N$
RO	Rosenbrock's Valley: $f = \sum_{i=1}^{N} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$, $-2.048 \le x_i \le 2.048$, $f(1,\ldots,1) = 0$
SC	Schwefel: $f = -\sum_{i=1}^{N} x_i \sin(\sqrt{\lvert x_i \rvert})$, $-500 \le x_i \le 500$, $f\big(420.9(1,\ldots,1)\big) \approx -418.9N$
MI	Michalewicz: $f = -\sum_{i=1}^{N} \sin(x_i) \times \sin(\frac{ix_i^2}{\pi})^{20}$, $0 \le x_i \le \pi$, $f(\bar{x}) = 0$ for $\bar{x} \in \{0,\pi\}^N$
FE	Frog egg parameter estimation (Zwolak et al. [2005])
BY	Budding yeast parameter estimation (Panning et al. [2006])

Gablonsky [2001]. The following tests demonstrate how ϵ affects the convergence speed on the benchmark functions. The convergence is defined as when both the global optimum value $f_{\min}$ and global optimum solution x_0 are achieved within less than 0.1% error, thus the desired accuracy is α = 1.0E-03. Jones et al. [1993] recommend ϵ to be the desired solution accuracy to find good optimization results with a reasonable amount of work. Table 4.2 lists the number of iterations (N_I) and evaluations (N_e) needed to converge to the solution for the benchmark functions with ϵ values in (0.0, 1.0E-02).

Table 4.2. The number of iterations N_I and evaluations N_e required for convergence with ϵ varying in (0.0, 1.0E-02). An asterisk prefixing certain entries indicates that they are lower bounds on the actual N_I and N_e.

	GR		**QU**		**RO**		**SC**		**MI**	
ϵ **value**	N_I	N_e	N_I	N_e	N_I	N_e	N_I	N_e	N_I	N_e
1.0E-02	259	3561	$^*12 \cdot 10^3$	$^*10^5$	151	6567	33	285	892	16771
1.0E-03	25	295	57	563	146	6883	22	151	312	10890
1.0E-04	15	143	57	587	146	7217	21	157	318	14559
1.0E-05	14	135	57	613	146	7423	21	157	319	17629
1.0E-07	14	135	57	637	146	7485	21	157	319	23059
0.0	14	135	57	679	146	7485	21	173	—	—

When ϵ is between 1.0E-03 and 1.0E-07, all benchmark function optimizations converge with a reasonable number of iterations and evaluations. $\epsilon = \alpha$ yields the smallest number of evaluations for QU, SC, and MI, and the second smallest for RO. The problem GR prefers ϵ as small as possible to minimize the amount of

work. However, observe that the MI optimization fails to converge when $\epsilon = 0.0$, because the search is biased to be so local that the search has to stop since the minimum box diameter 1.26E-15 has been reached after 235 iterations and 25301 evaluations. Moreover, using $\epsilon = 1.0\text{E-}02 > \alpha$ increases the number of evaluations a thousand fold for the problem QU. Therefore, the desired accuracy is proved to be a reasonable choice for ϵ, unless the optimization goal is to find a local solution, in which case $\epsilon = 0.0$ can improve local convergence, or to broadly explore the feasible set, when $\epsilon > \alpha$ is appropriate.

4.2 Data Structure Efficiency

The dynamic data structures were introduced in the original 2002 version of the VTDIRECT95 serial code. In terms of the execution time and memory usage, two other implementations using static data structures were compared empirically with the serial version described by He et al. [2002], which has demonstrated its strength in dealing with unpredictable memory requirements. The next improvement on data structures was constructing box columns as heaps instead of sorted lists. In addition, lexicographical order of box center coordinates is enforced in the heap for maintaining determinism. Table 4.3 compares the execution time of an earlier version with sorted lists (SL), the current version with lexicographically ordered heaps (HL), and a version (HNL) that was built without the lexicographical order comparisons. Clearly, using heaps is much more efficient than using sorted lists. Also, the lexicographical order comparison accounts for a very tiny portion of the entire operational cost.

Table 4.3. Execution time (in seconds) of the versions with sorted lists (SL), with lexicographically ordered heaps (HL), and heaps without lexicographical order comparison (HNL). The evaluation limit is 10^5 for all test functions.

#	SL	HL	HNL
GR	233.21	10.60	10.57
QU	840.43	56.70	56.38
RO	226.52	8.12	7.92
SC	273.58	11.97	11.86
MI	468.65	29.69	29.32

The last important improvement on data structures is limiting box columns (LBC). The experimental results in He et al. [2007a] show that LBC reduces the memory usage by 10–70% for selected high dimensional test problems. The following experiments investigate the added computational cost of LBC. Figure 4.1 compares the growth of the execution time with LBC or without LBC (NON-LBC) as the number of function evaluations N_e increases for the 2-dimensional problem GR and 4-dimensional problem RO.

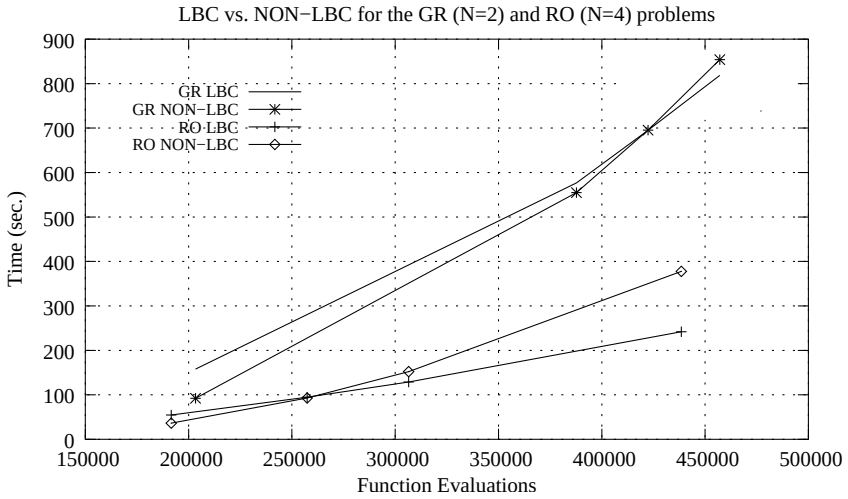


Fig. 4.1. Growth of execution time with LBC or NON-LBC as N_e increases for the 2-dimensional problem GR and the 4-dimensional problem RO.

Regardless of the different problem structures, LBC performs slower than NON-LBC until N_e reaches a certain "crossover point", where NON-LBC begins to run slower due to depleting memory resources and more expensive operations on the box columns longer than those in LBC, while LBC keeps the box columns as short as possible. Observe that the crossover point for the 2-dimensional problem GR is approximately twice the crossover point for the 4-dimensional problem RO. This observation inspired the next set of experiments to find the approximate number of evaluations N_x at the crossover points for all five benchmark functions, and define a condition to turn off LBC if

$$L_e(2N+2) < \overline{L_x}, \tag{4.1}$$

where N is the problem dimension, $2N+2$ is the number of real values in a `Hyperbox`, L_e is the user specified limit on evaluations (`MAX_EVL` in the code), and $\overline{L_x} \approx 2 \times 10^6$ is the average of the five $N_x(2N+2)$ values shown in Table 4.4. The condition (4.1) is checked only when the user specifies stopping conditions based on both `MAX_ITER` and `MAX_EVL`.

Table 4.4. The crossover point N_x and the threshold L_x for all five test functions.

#	GR	QU	RO	SC	MI
N	2	3	4	2	5
$N_x/10^3$	422	190	258	442	129
$L_x/10^6$	2.5	1.5	2.5	2.6	1.5

4.3 Parallel Performance

Comprehensive analytical and experimental results in He et al. [2007c], He et al. [2007a], and He et al. [2007b] regarding the parallel performance are reviewed in

this section. In addition, several new experiments were conducted to measure the scalability in terms of the average overhead per function evaluation and the scaled speedup.

Objective function cost is one of the key parameters that affects the parallel performance under different parallel schemes. For expensive functions, evaluation tasks should be distributed in the smallest possible chunks ($N_b = 1$) for better load balancing, which is a major reason behind improved parallel efficiency. Much better load balancing is achieved with $N_b = 1$ than $N_b = 5$, 10, or 20 for the 150-dimensional problem GR with $T_e = 0.1$ in an experiment running on 100 processors under the vertical scheme (He et al. [2007a]). If the objective function cost is low, either a vertical scheme with function evaluation stacking ($N_b > 1$) or a horizontal scheme with a small number of masters can be used to achieve some speedup. It has been shown by He et al. [2007b] that a horizontal scheme using three, four, and five masters yields better parallel efficiency than the corresponding vertical scheme on the same number of processors for the same problem GR when $T_e \leq$ 2.5E-04 on System X and $T_e \leq$ 1.0E-03 on Anantham.

As the isoefficiency analysis in He et al. [2007b] concluded, different parallel system characteristics (e.g., communication round-trip cost) matter only for cheap functions. The higher the function cost, the better the scalability of the vertical scheme, which is more scalable than the horizontal scheme in general. The overhead due to processor idleness and communication grows faster in the horizontal scheme when more processors are used. For expensive functions, the biggest performance impact comes from problem-dependent factors such as the number of evaluation tasks per iteration, which determines the degree of concurrency. Decoupling **Sampling** and **Selection** is the first step taken in VTDIRECT95 to enhance the program concurrency. In future research, a promising solution would be to generate a sufficient number of tasks for idle workers by pre-fetching boxes that may become potentially optimal in later iterations (speculative evaluation).

Here, new scalability tests were done on all five benchmark functions with a fixed cost $T_e = 0.1$, a growing problem dimension $N = 2^i$ ($i = 2, \ldots, 6$), and an increasing number of processors $p = 10 \cdot 2^{i-1}$ ($i = 1, \ldots, 5$). Since T_e is maintained approximately constant by using a microsecond precision utility function `gettimeofday` in C, the increase from 0.1 to the average cost per evaluation $\overline{T_e} = pT_p/N_e$ can be considered as the average overhead per function evaluation as both N and p increase, where T_p is the parallel execution time with p processors and N_e is the actual number of evaluations.

Figure 4.2 plots how $\overline{T_e}$ changes as N and p grow. For most problems, $\overline{T_e}$ grows slowly until N reaches 32 and p reaches 80. The larger increase in overhead for the QU problem could be related to its special structure that causes more worker idleness and more computation during **Selection** and **Division**. Additionally, doubling p every time that N doubles is not guaranteed to maintain efficient performance, since the growth in N may not produce more concurrent evaluation tasks proportionally. On the other hand, load balancing can be improved greatly for a

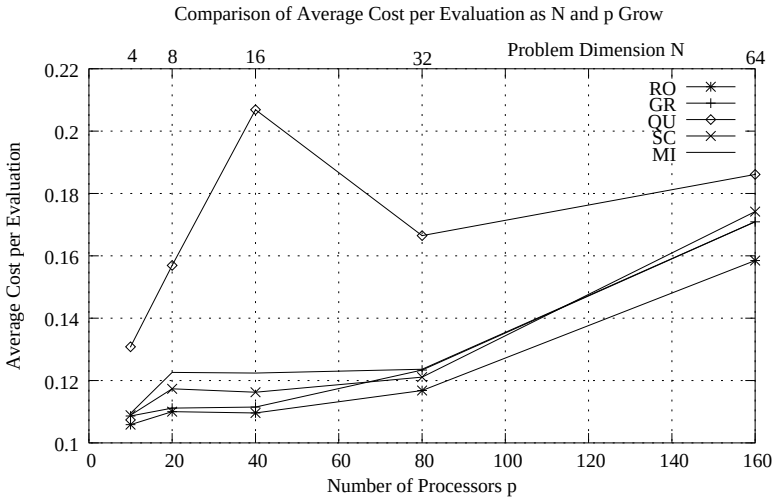


Fig. 4.2. The growth of $\bar{T}_e$ as N and p increase for all five benchmark functions.

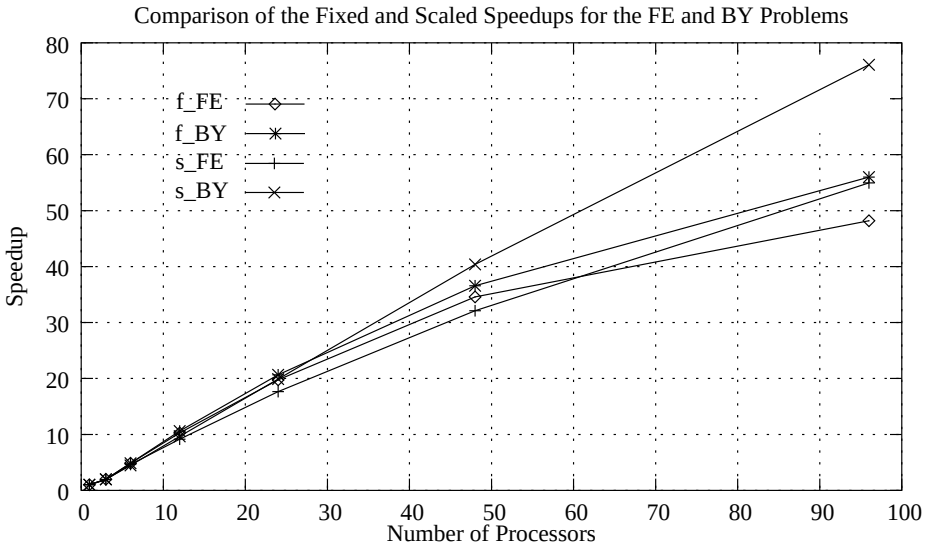
fixed p, when other parameters, such as N, the iteration limit $I_{\max}$, or the evaluation limit L_e, grow. An experiment (He et al. [2007a]) has shown that the parallel efficiency curves for 100 processors reach more than 90%, 80%, and 70% when $I_{\max} = 30$, 20, and 10, respectively, as N increases from 10 to 50, 100, and 150 for the problem RO with the same cost $T_e = 0.1$.

The next scalability study varies the evaluation limit L_e as p grows for the problems FE and BY. The comparisons are also done with the number of iterations/evaluations fixed as p grows. The fixed speedup and scaled speedup are plotted in Figure 4.3. The fixed speedup follows the conventional definition $S_f = T(W,1)/T(W,p)$, where W is the fixed work load, $T(W,1)$ is the execution time on a single processor, and $T(W,p)$ is the parallel execution time on p processors. The scaled speedup is usually calculated as $S_s = pT(W_s,1)/T(pW_s,p)$, where W_s is the base work for a single processor, $T(W_s,1)$ is the execution time with a single processor on the work W_s, and $T(pW_s,p)$ is the parallel execution time with p processors and the linearly increased work pW_s. (See, e.g., Quinn [2003] for a discussion of the relationship between S_s and S_f.) For DIRECT, S_s needs to be redefined because the stopping condition $N_e > L_e$ is checked after all convex hull boxes have been sampled and subdivided, meaning that the number of evaluations may not grow exactly linearly as DIRECT iterates. Therefore, define

$$S_s = \frac{pT(W_s,1)}{T(W_p,p)(pW_s/W_p)},$$

where W_s represents the base number of evaluations on a single processor and $W_p > W_s$ is the increased number of evaluations as p grows. Hence, $T(pW_s,p)$ is approximated with $T(W_p,p)(pW_s/W_p)$.

In Figure 4.3, the fixed speedup is obtained with $W = 3251$ for the problem FE and $W = 1699$ for the problem BY. For the scaled speedup, $W_s = 449$ for the problem FE and $W_s = 287$ for the problem BY. The evaluation limits with p processors are listed in the table under the plot for both problems FE and BY.



p	1	3	6	12	24	48	96
W_p(FE)	449	941	2465	5413	11339	23187	47051
W_p(BY)	287	571	1699	3095	7263	19059	40045

Fig. 4.3. Comparison of the fixed and scaled speedups for the problems FE and BY as L_e and p grow. The actual work W_p is shown in the table below the plot.

The scaled speedup is slightly worse than the fixed speedup when $p \leq 24$ for the problem BY and when $p \leq 48$ for the problem FE. Note also that the speedup for the problem BY is better than that for the problem FE, because the problem dimension and evaluation cost of the problem BY are higher.

When a large number of processors are involved as in some of the experiments above, domain decomposition should be considered to improve the load balancing and scalability. Comparison runs between a single domain and four subdomains using four masters and 196 workers were done on five 150-dimensional benchmark functions and the problem BY in He et al. [2007a]. The search with four subdomains gave narrower workload ranges for workers, thus better load balancing than that with the single domain. Better optimization solutions may also be discovered earlier with decomposed subdomains than with a single domain for problems with irregular structures. For the problems FE and BY, the solution found with a single domain search was worse than that with a four subdomain search, since with the same $I_{\max}$ more function evaluations were generated across multiple subdomains.

4.4 Checkpointing Overhead

The following experiments measure the checkpointing overhead under the serial and parallel running environments. The evaluation limit is 10^5 and the original cost $T_e \approx 0.0$ is used for the five test functions. Table 4.5 reports the execution time without checkpointing T_{nc}, the time for "saving" T_{sv}, and the time for "recovery" T_r. First, note that T_{sv} is always greater than T_{nc}, but T_r is sometimes less than T_{nc}. This means the recovery overhead is very small even for cheap functions.

Second, the saving overhead depends heavily on the number of iterations, because the checkpoint logs are flushed to the file at the end of each iteration. The average saving overhead per iteration is approximately 0.003 second.

Table 4.5. Comparison of serial checkpointing overhead (in seconds) for five test functions. I is the number of iterations upon termination with the stopping rule $L_e = 10^5$, T_{nc} is the execution time without checkpointing, T_{sv} is the execution time with saving, and T_r is the execution time for recovery.

#	I	T_{nc}	T_{sv}	T_r
GR	3057	10.58	18.85	11.50
QU	12238	56.70	87.38	57.28
RO	1198	8.12	11.61	9.22
SC	3637	11.97	21.43	12.96
MI	1968	29.69	34.61	24.05

In parallel environments, all processors are masters since the function evaluation cost is too low to justify distribution to workers. Table 4.6 shows the timing results of saving and recovering the checkpoint logs saved by a single master and recovered using multiple masters, and both saving and recovery with multiple masters. The checkpointing overhead on a single master in the parallel version is slightly more than that in the serial version. Recovering with multiple masters costs more than that with a single master, but the overhead does not grow dramatically as the number of masters doubles. In some cases, the recovery overhead even drops with more masters. The saving and recovery overhead with three masters is also comparable to that with the single master. In summary, checkpointing overhead is very insignificant compared to the benefit of saved computation for expensive function evaluations.

Table 4.6. Comparison of parallel checkpointing overhead (in seconds) when saving with a single master (m1) and saving with three masters (m3) for five test functions. The stopping rule is evaluation limit $L_e = 10^5$. T_{nc} is the execution time without checkpointing, T_{sv} is the execution time with saving, and $T_r(m)$ is the execution time for recovery with m masters.

#	T_{nc}	T_{sv}	$T_r(1)$	$T_r(2)$	$T_r(3)$	$T_r(4)$	$T_r(5)$	$T_r(7)$	$T_r(8)$
GR m1	13.22	21.06	14.43	26.11	–	22.76	–	–	27.86
GR m3	11.95	21.07	–	–	12.71	–	23.49	27.24	–
QU m1	55.87	109.15	57.42	76.27	–	86.51	–	–	104.47
QU m3	68.58	83.04	–	–	49.52	–	95.78	107.46	–
RO m1	9.33	12.80	10.70	16.89	–	13.74	–	–	14.50
RO m3	6.61	10.47	–	–	7.02	–	13.28	13.89	–
SC m1	14.44	23.38	15.37	30.05	–	26.11	–	–	30.39
SC m3	14.49	25.59	–	–	13.68	–	34.40	29.87	–
MI m1	30.31	35.58	23.30	21.57	–	18.87	–	–	20.68
MI m3	14.40	17.84	–	–	11.74	–	18.28	20.03	–

5. ORGANIZATION AND USAGE

The README file distributed with the package describes the physical organization of the package into files, and includes the basic instructions for compiling, testing, and running the installed serial and parallel codes. This section describes the organization and usage of the key modules, driver subroutines, and test programs.

5.1 Package Organization

Figure 5.1 shows the high level organization of VTDIRECT95. The module `VTdirect_MOD` declares the user called driver subroutine `VTdirect` for the serial code. Correspondingly, the module `pVTdirect_MOD` declares the user called parallel driver subroutine `pVTdirect`, the subroutine `pVTdirect_init` for MPI initialization, the subroutine `pVTdirect_finalize` for MPI finalization, as well as the data types, parameters, and auxiliary functions used exclusively in the parallel code.

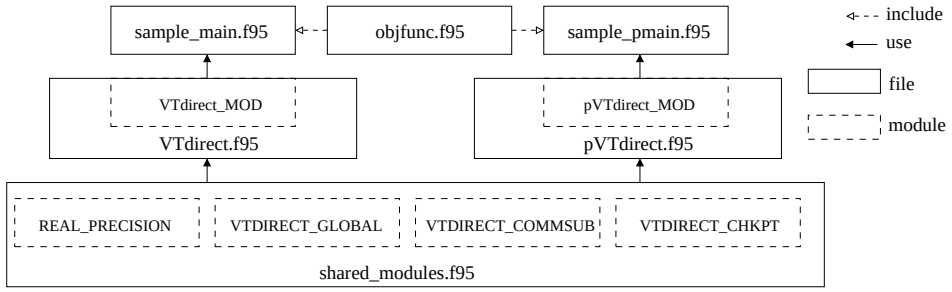


Fig. 5.1. The module/file dependency map.

The two driver subroutines `VTdirect` and `pVTdirect` share the modules: (1) `REAL_PRECISION` from HOMPACK90 (Watson et al. [1997]) for specifying the real data type, (2) `VTDIRECT_GLOBAL` containing definitions of derived data types, parameters, and module procedures, (3) `VTDIRECT_COMMSUB` containing the subroutines and functions common to both the serial and parallel versions, and (4) `VTDIRECT_CHKPT` defining data types and module procedures for the checkpointing feature. These shared modules are merged in the file `shared_modules.f95` as shown in Figure 5.1. `sample_main` and `sample_pmain` are sample main programs that call `VTdirect` and `pVTdirect`, respectively, to optimize five test objective functions defined in `objfunc.f95` and verify the installation. The dependencies between the package components are depicted in Figure 5.1.

In the sample serial main program `sample_main` each test objective function illustrates a different way of calling the driver subroutine `VTdirect`. The calls illustrate the four different stopping rules—maximum number of iterations `MAX_ITER`, maximum number of function evaluations `MAX_EVL`, minimum box diameter `MIN_DIA`, and minimum relative decrease in objective function value `OBJ_CONV`. For the last objective function, a multiple best box (MBB) output is illustrated. Details of the arguments are in comments at the beginning of the subroutine `VTdirect`. Different parallel schemes are used in the test cases for `pVTdirect`, called by the sample parallel main program `sample_pmain`. Both sample main programs print to standard

out the stopping rule satisfied, the minimum objective function value, the minimum box diameter, and the number of iterations, function evaluations, and the minimum vector(s). In addition, the test output for pVTdirect lists the number of masters per subdomain and the number of subdomains.

Different computation precision and different compiled code on different systems may require different numbers of iterations or evaluations to reach the desired solution accuracy (1.0E-03) specified in the test programs. If a test program fails to locate the optimum value or the optimum point given the stopping conditions in the supplied namelist input file, the stopping conditions can be adjusted accordingly.

5.2 Using VTDIRECT95

One of the virtues of DIRECT, shared by VTDIRECT95, is that it only has one tuning parameter (ϵ) beyond the problem definition and stopping condition. Using VTDIRECT95 basically takes three simple steps. First, define the objective function with an input argument for the point coordinates (`c`), an output argument for evaluation status (`iflag`), and an output variable for the returned function value (`f`). A nonzero return value for `iflag` is used to indicate that `c` is infeasible or `f` is undefined at `c`. The user written objective function is a FUNCTION procedure that must conform to the interface:

```
INTERFACE
  FUNCTION Obj_Func(c, iflag) RESULT(f)
    USE REAL_PRECISION, ONLY: R8
    REAL(KIND = R8), DIMENSION(:), INTENT(IN)::  c
    INTEGER, INTENT(OUT)::  iflag
    REAL(KIND = R8)::  f
  END FUNCTION Obj_Func
END INTERFACE
```

Second, allocate the required arrays and specify appropriate input parameters to call one of the driver subroutines. In the parallel case, the MPI initialization and finalization subroutines need to be called before and after calling the parallel driver subroutine (pVTdirect), unless MPI is initialized and finalized elsewhere in the same application. The required arrays include the input lower (`L`) and upper (`U`) bounds, and an output array for the optimum vector (`X`). Additionally, in the parallel version, the return status is also an array, required to be allocated beforehand, to hold statuses returned from subdomains, even if only one domain exists, in which case the size of the status array is one. If the user desires to specify the optional input argument `BOX_SET`, an array of boxes must be allocated and an optional weight array `W` for dimensional scaling may also be allocated.

All other input parameters specified in the argument list of the driver subroutine are conveniently read in from a NAMELIST file, as illustrated in the sample main programs. Using namelist files is an elegant way of varying input parameters as needed, without recompiling the program. The namelist file `pdirectRO.nml` shown below is to test `pVTdirect` for optimizing the 4-dimensional problem RO. The parameters are grouped into four categories (NAMELISTs): parallel scheme

PSCHEME, problem configuration PROBLEM, optimization parameters OPTPARM, and checkpointing option CHKPTOP. This example uses two subdomains and two masters per subdomain, and the stopping condition is when the minimum box diameter reaches 1.0E-05. The checkpointing feature is activated when chkpt_start equals 1 (saving) or 2 (recovery). The program will terminate if the checkpoint file errors occur as explained in the section on error handling (cf. Section 3.4). It is the user's responsibility to maintain the checkpoint files, including renaming or removing old files.

```
&PSCHEME n_subdomains=2 n_masters=2 bin=1 /

&PROBLEM N=4
    LB(1:4)=-2.048,-2.048,-2.048,-2.048
    UB(1:4)=2.048,2.048,2.048,2.048 /

&OPTPARM iter_lim=0 eval_lim=0 diam_lim=1.0E-5 objf_conv=0.0
    eps_fmin=0.0 c_switch=1 min_sep=0.0 weight(1:4)=1,1,1,1
    n_optbox=1 /

&CHKPTOP chkpt_start=0 /
```

Finally, the last step is to interpret the return status, collect the results, and deallocate the arrays as needed. The return status consists of two digits. The tens digit indicates the general status: 0 for a successful run, 1 for an input parameter error, 2 for a memory allocation error or failure, and 3 for a checkpoint file error. The stopping condition for a successful run is further indicated in the units digit, which also points to the exact source of error if a nonzero status is returned. For example, a return status of 33 means the checkpoint file header does not match with the current setting. All the error codes and interpretations can be found in the source code documentation. A successful run returns the optimum value and vector(s) in the user-prepared variables and arrays. In order to receive a report on the actual number of iterations/evaluations, or minimum box diameter, these optional arguments must be present in the argument list. The final results of calling pVTdirect are merged on processor 0 (the root master), so proc_id is returned to designate the root to report the results. VTDIRECT95 is designed so that the optimization results may be directly fed to another procedure or process, the typical situation in large scale scientific computing.

ACKNOWLEDGMENTS

The authors are indebted to Paul Boggs, John Dennis, and Donald Jones for many discussions and suggestions.

BIBLIOGRAPHY

Baker, C.A., Watson, L.T., Grossman, B., Haftka, R.T., and Mason, W.H. 2000. Parallel global aircraft configuration design space exploration. High Performance Computing Symposium 2000, A. Tentner (Ed.), Soc. for Computer Simulation Internat, San Diego, CA, 101–106.

Bartholomew-Biggs, M.C., Parkhurst, S.C., and Wilson, S.P. 2003. Global optimization approaches to an aircraft routing problem. *EUR J. Operational Research 146*, 417–431.

Batchu, R., Dandass, Y.S., Skjellum, A., and Beddhu, M. 2004. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing 7*, 303–315.

Bouteiller, A., Herault, T., Krawezik, G., Lemarinier, P., and Cappello, F. 2006. MPICH-V Project: A multiprotocol automatic fault-tolerant MPI. *International of High Performance Computing and Applications 20*, 319–333.

Carter, R.G., Gablonsky, J.M., Patrick, A., Kelly, C.T., and Eslinger, O.J. 2001. Algorithms for noisy problems in gas transmission pipeline optimization. *Optimization and engineering 2*, 139–157.

Fagg, G.E., Bukovsky, A., and Dongarra, J.J. 2001. HARNESS and fault tolerant MPI. *Parallel Computing 27*, 1479–1495.

Finkel, D.E. and Kelly, C.T. 2004. An adaptive restart implementation of DIRECT. CRCS-TR04-30, Center for Research in Scientific Computation, North Carolina State University, Raleigh, NC, USA.

Gablonsky, J.M. 2001. Modifications of the DIRECT algorithm. Ph.D. thesis, Department of Mathematics, North Carolina State University, Raleigh, NC, 2001.

Gropp, W. and Lusk, E. 2004. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications 18*, 363–372.

He, J., Verstak, A., Watson, L.T., Stinson, C.A., Ramakrishnan, N., Shaffer, C.A., Rappaport, T.S., Anderson, C.R., Bae, K., Jiang, J., and Tranter, W.H. 2004. Globally optimal transmitter placement for indoor wireless communication systems. *IEEE Transactions on Wireless Communications 3*, 1906–1911.

He, J., Verstak, A., Sosonkina, M., and Watson, L.T. 2007b. Performance modeling and analysis of a massively parallel DIRECT: Part 2. Technical Report TR-07-02, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA.

He, J., Verstak, A., Watson, L.T., and Sosonkina, M. 2007a. Performance modeling and analysis of a massively parallel DIRECT: Part 1. Technical Report TR-07-01, Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA.

He, J., Verstak, A., Watson, L.T., and Sosonkina, M. 2007c. Design and implementation of a massively parallel version of DIRECT. *Computational Optimization and Applications*, to appear.

He, J., Watson, L.T., Ramakrishnan, N., Shaffer, C.A., Verstak, A., Jiang, J., Bae, K., and Tranter, W.H. 2002. Dynamic data structures for a direct search algorithm. *Computational Optimization and Applications 23*, 5–25.

Horst, R., Pardalos, P.M., and Thoai, N.V. 2000. *Introduction to Global Optimization*. Kluwer, Boston.

Horst, R. and Tuy, H. 1996. *Global Optimization: Deterministic Approaches*. Springer-Verlag, Berlin.

Jones, D.R., Pertunen, C.D., and Stuckman, B.E. 1993. Lipschitzian optimization without the Lipschitz constant. *J. Optimization Theory and Applications 79*, 157–181.

Ljungberg, K., Holmgren, S., and Carlborg, Ö. 2004. Simultaneous search for multiple QTL using the global optimization algorithm DIRECT. *Bioinformatics (Oxford, England) 20*, 1887–1895.

Louca, S., Neophytou, N., Lachanas, A., and Evripidou, P. 2000. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters 10*, 371–382.

Panning, T.D., Watson, L.T., Allen, N.A., Chen, K.C., Shaffer, C.A., and Tyson, J.J. 2006. Deterministic global parameter estimation for a model of the budding yeast cell cycle. *J. of Global Optimization*, to appear.

Pinter, J.D. 1996. *Global Optimization In Action*. Kluwer Academic Publishers, Boston.

Plank, J.S. 1997. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, TN.

Quinn, M.J. 2003. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education (ISE Editions), New York.

Watson, L.T. and Baker, C.A. 2001. A fully-distributed parallel global search algorithm. *Engineering Computations 18*, 155–169.

Watson, L.T., Sosonkina, M., Melville, R.C., Morgan, A.P., and Walker, H.F. 1997. Algorithm 777: HOMPACK90: A suite of FORTRAN 90 codes for globally convergent homotopy algorithms. *ACM Transactions on Mathematical Software 23*, 514–549.

Zhu, H. and Bogy, D.B. 2002. DIRECT algorithm and its application to slider air-bearing surface optimization. *IEEE Transactions on Magnetics 38*, 2168–2170.

Zwolak, J.W., Tyson, J.J., and Watson, L.T. 2005. Globally optimised parameters for a model of mitotic control in frog egg extracts. *IEE Systems Biology 152*, 81–92.