A Hierarchical Parallel Scheme for a Global Search Algorithm

J. He*, M. Sosonkina††, C. A. Shaffer*, J. J. Tyson†, L. T. Watson*, J. W. Zwolak*
Department of Computer Science*, Department of Biology†
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061
Ames Laboratory, Iowa State University††
236 Wilhelm Hall, Ames, IA 50011.
Contact E-mail: jihe@vt.edu

Abstract—**This paper presents a sophisticated and efficient parallel scheme for the DIRECT global optimization algorithm of Jones et al. (1993). Although several sequential implementations for this algorithm have been successfully applied to large scale MDO problems, few parallel versions of the DIRECT algorithm have addressed well algorithm characteristics such as a single starting point, an unpredictable workload, and a strong data dependency. These challenges engender many interesting design issues including domain decomposition, data access and management, and workload balancing. In the present work, a hierarchical parallel scheme has been developed to address these challenges at three levels. Each level is supported by a programming model to access shared data sets, distribute workload, or exchange messages. Results for numerical simulations and performance measurements on three test problems were obtained on a 200 node Linux cluster.**

Index Terms—**DIRECT (DIviding RECTangles) algorithm, global optimization, GPSHMEM (generalized portable shared memory), load balancing strategy, multidisciplinary design optimization**

1. Introduction

The optimization algorithm DIRECT (DIviding RECTangles) is a global search algorithm proposed by Jones et al. [11], designed as an effective approach to solve continuous optimization problems subject to simple constraints. In the past decade, DIRECT has been successfully applied to many modern large-scale multidisciplinary engineering problems ([1], [2], [9], and [18]). Recently, DIRECT has been used in global nonlinear parameter estimation problems in systems biology [14]. However, unnecessary overhead and complexity caused by inefficient implementation inside other software packages (e.g., Matlab) may obscure DIRECT's advanced features. Some computational biologists are attracted by its unique strategy of balancing global and local search, its selection rules for potentially optimal regions according to a Lipschitz condition, and its easy-to-use black-box interface. Like other global optimization approaches of [4] and [6], DIRECT is being challenged by high-dimensional ($\geq$ 50) problems including nonlinear models for parameter estimation. The present work applies DIRECT to a 154-parameter estimation problem for a budding yeast cell cycle model. Simulations were also conducted on two n-dimensional test functions — Griewank and quartic with $n = 50$ and $n = 100$.

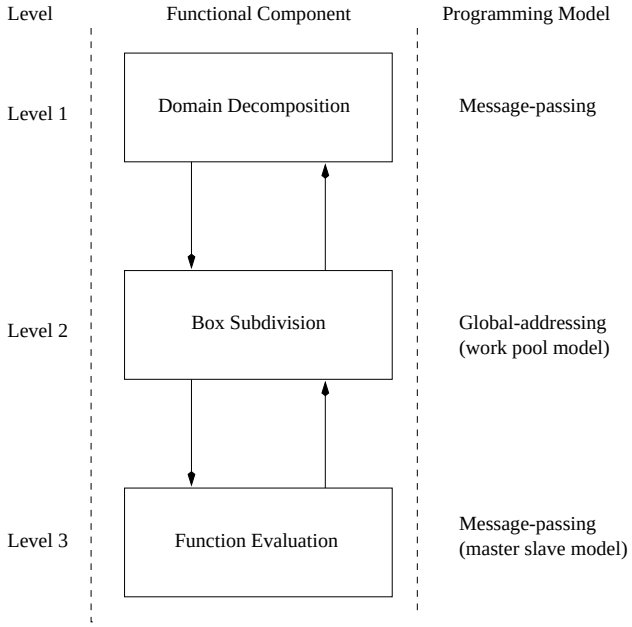

Figure 1.1. Three functional levels with a mixed programming paradigm.

As the scale of both problems and clusters of workstations grows, computational parallelism of optimization algorithms has become a very active research area. However, the nature of the DIRECT algorithm presents both potential benefits and difficulties for a sophisticated and efficient parallel implementation. Gablonsky [5] and Baker et al. [1] are among the few parallel DIRECT implementations known in the public domain. In [5], Gablonsky adopts a master-slave paradigm to parallelize the function evaluations, but little discussion is given to the issue of parallel performance and potential problems, such as load-balancing and interprocessor communication, both of which raise many challenging design issues.

A major contribution in [1] is a distributed control version equipped with dynamic load balancing strategies. Nevertheless, that work did not fully address other design issues such as a single starting point and a strong data dependency.

The present work proposes a hierarchical parallel scheme (shown in Figure 1.1) to address design issues by three function components in different levels. Level 1 splits the entire search space to start the processing at multiple points, detects the stopping conditions, and merges the results at the end. This level transforms the original single-start sequential algorithm to a multistart parallel algorithm. Below Level 1, Level 2 uses GPSHMEM [15] to establish a global addressing space to ease the strong data dependency problem occurring in the algorithm step (refer to Section 2) that identifies a set of potentially optimal boxes to be subdivided at the next iteration. This globally shared data structure also forms a work pool paradigm [7] to apply a dynamic load balancing strategy to adjust the workload among processors at Level 2. Similar to [1], a master-slave paradigm is used between Level 2 and 3 for distributing function evaluation tasks. An assignment scheme for processors at each level is proposed in Section 3. However, the optimal numbers of processors for different levels are problem-dependent. This is an open research question in future work.

Both Levels 2 and 3 take advantage of dynamic process management in MPI-2 [8] so that processors are assigned to these two levels at run time. As shown in Figure 1.1, a mixed programming paradigm is constructed with a global addressing model and a message passing model. A similar style of subgrouping processors for multiple level parallelism (MLP) was called GSPMD (group single program multiple data) in [17]. By contrast, in [12] and [16], a hybrid parallel programming model involving thread-level parallelism (OpenMP) and message passing (MPI) was used to vary the number of threads and CPUs in order to simplify the dynamic load balancing strategy in MLP. Mixed parallel programming models may become a trend due to the recent deployment of many large scale clusters of shared memory multiprocessor workstations [12].

Section 2 describes the DIRECT algorithm and the parallel design issues. The proposed parallel implementation is described in Section 3, with two test functions and the budding yeast cell cycle model described in Section 4. Computational results and some performance analysis are included in Section 5.

2. Design Challenges

The sequential DIRECT algorithm can be described by the following six steps [11], given an objective function $f(x)$ and the n-dimensional design space $D = \{x \mid l \leq x \leq u\}$.

Step 1. Normalize the design space 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 $t = 0$.

Step 2. Identify the set S of potentially optimal boxes.

Step 3. Select any box $j \in S$.

Step 4. Divide the box j as follows:

(1) Identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.

(2) 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.

(3) 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.

Step 5. Set $S = S - \{j\}$. If $S \neq \emptyset$ go to Step 3.

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

Steps 2 to 6 form a processing loop controlled by two stopping criteria—limits on iterations and function evaluations. Starting from the center of the initial hypercube, DIRECT makes exploratory moves across the design space by probing the potentially optimal hyperboxes. "Potentially optimal" is precisely defined in [11], but roughly a hyperbox is potentially optimal if, for some Lipschitz constant, the objective function is potentially smaller in that hyperbox than in any other hyperbox. It is observed in Step 4 that multiple new hyperboxes are generated for each potentially optimal hyperbox. The multiple function evaluation tasks at each iteration give rise to a natural functional parallelism used both in [1] and [5].

In addition, a few design challenges are also observed here. First, the algorithm starts with one normalized domain, which produces simply one evaluation task for all the acquired processors. With a single starting point, load balancing is always an issue at an early stage, even though the situation will be improved as the algorithm progresses by subdividing the domain and generating multiple evaluation tasks. Second, the number of boxes subdivided at each iteration is unpredictable depending on the result

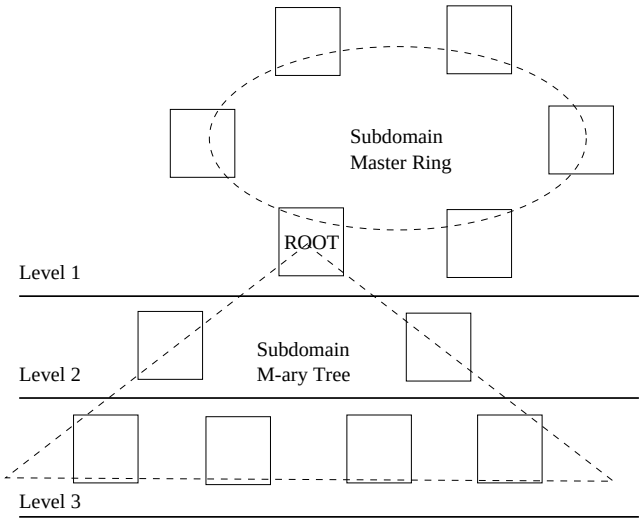

Figure 3.1. A 3-level hierarchical parallel scheme.

of identifying the potentially optimal boxes. For iterations that generate fewer new boxes, a load imbalance occurs with some processors sitting idle. Third, a strong data dependency exists throughout the algorithm steps. Only Step 2 and Step 4 can be parallelized, respectively, in terms of functional parallelism and data parallelism. Both involve shared data sets with a considerable growth rate, which challenges memory system performance on a single machine. Efficient data decomposition and access methods across multiple machines are the major issues here.

3. Parallel Scheme and Implementation

A hierarchical parallel scheme is proposed here to address the above mentioned design challenges. It consists of three logical levels as shown in Figure 3.1. Each level deals with different design aspects including domain decomposition, load balancing, and task or data parallelism.

Level 1 holds a ring of processors, each of which is responsible for a DIRECT search in an assigned subdomain. A root processor is noted as the leading node on the ring. The problem domain D is decomposed by each processor on the ring into

$$S^2 \leq N_1 = \lfloor \sqrt{P} \rfloor \tag{3.1}$$

subdomains, where P is the total number of processors. The parallel scheme described here requires $P \geq 16$ for the proposed processor assignment scheme. When the number of available processors is below 16, P should be set as 16, the total number of processes, some of which may run on the same physical processor. The decomposition is accomplished in two phases. In phase one, each processor on the ring finds the longest dimension of the domain and subdivides it into $S = \lfloor \sqrt{N_1} \rfloor$ partitions, each of which will be processed by S processors. In phase two, inside each of the resulting partitions, the currently longest dimension is subdivided into S parts. The total number of subdomains S^2 equals the number of processors needed for Level 1. As a result, exactly one processor is in charge of one subdomain.

For each subdomain, a process is spawned by MPI to be the subdomain master starting a DIRECT search. Subsequently, a logical ring is formed by the spawned processes to adopt a termination detection strategy depending on different stopping criteria. The overall termination condition is to keep every subdomain active until all subdomains have satisfied the specified stopping criteria. This rule results in more computational cost in terms of total number of iterations, function evaluations, or degree of subdivision. Thus, the stopping condition for the proposed 3-level parallel scheme is effectively a lower bound on the computational cost, while for the sequential algorithm, it is an exact limit for computation cost. To provide exact stopping criteria, Level 1 is an optional layer, which can be removed to form a 2-level single-start parallel algorithm with a simpler control framework.

A ring topology naturally fits the equal relationship among subdomain masters at Level 1. Moreover, it represents the dependency of the stopping process of each subdomain on other subdomains on the ring. A subdomain terminates search activities only after all other subdomains have reached their specified stopping criteria. This discentralizes the termination control among the ring, thus avoiding the bottleneck at the root subdomain master. On the other hand, the communication latency on a ring is higher than on some other topologies, such as a tree. To reduce the communication time on the ring and improve the performance at Level 1, future work can consider a tree topology instead of a ring. The stopping process for the entire domain is controlled by a token passed in the ring that consists of subdomain masters. It is originated from the root subdomain master R_0 and passed around the rest of the ring. After the local stopping criteria are met, each subdomain master R_i checks if a token has arrived at each iteration. If not, DIRECT proceeds. If the token was received, the token is passed along in the ring. After the token is passed back to R_0, a termination message is sent to all R_i. Lastly, R_0 will collect the final results and report the total number of evaluations and the range of number of iterations as well as minimum diameters.

For Levels 2 and 3, there are $P - S^2$ processors available. Each subdomain master dynamically

spawns $\lfloor M \rfloor$ mini master processes at Level 2 for box subdivision tasks, where

$$M = \sqrt{\frac{P - S^2}{S^2}},$$

derived from $M^2 \times S^2 = P - S^2$. Similarly, each mini master spawns $\lfloor \sigma \rfloor$ or $\lceil \sigma \rceil$ slaves for function evaluation tasks, where

$$\sigma = \frac{P - S^2(1 + \lfloor M \rfloor)}{S^2 \lfloor M \rfloor}.$$

A $\lfloor M \rfloor$-ary tree structure (in Figure 3.1) is rooted at a subdomain master, which is at the boundary of Levels 1 and 2. Therefore, a subdomain master plays two roles—one for communicating with the processes on the subdomain ring, and the other for managing the search in the subdomain $\lfloor M \rfloor$-ary tree. Subdomain management tasks here include updating current search results, and detecting local stopping conditions. Pseudo code 3.1 shows the interactions between the root mini master M_0 and other M_js ($j = 1, 2, \ldots, \lfloor M \rfloor - 1$) in a subdomain i ($i = 0, 1, \ldots, S^2$), which is managed by a subdomain master R_i. To reduce the control overhead, no handshakings are involved between R_i and M_0 before the local stopping criteria are met.

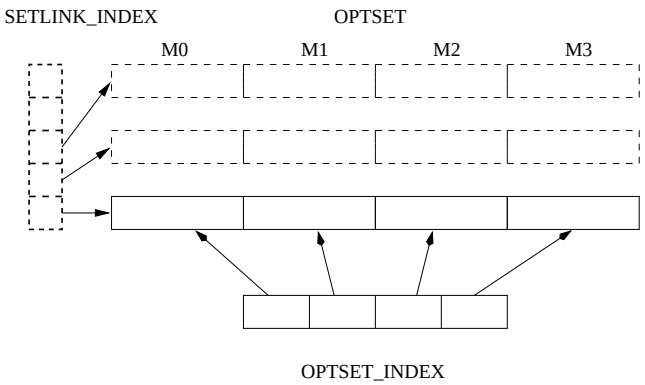

Figure 3.1. Data structures in GPSHMEM.

M_0 receives DIRECT parameters (problem size N, domain D, and stopping conditions C_{stop}) from R_i;
broadcast DIRECT parameters to all M_j;
done :=FALSE;
while TRUE
  if M_0 **then**
    if *done* =FALSE **then**
      run one DIRECT iteration (reduce intermediate results);
        if C_{stop} satisfied **then**
          done :=TRUE;
          send *done* to R_i;
        end if
        continue;
    else
      receive a message from R_i;
      if not a termination message **then**
        send a handshaking message to R_i;
        broadcast a message to keep M_js working;
        run one DIRECT iteration (reduce intermediate results);
        continue;
      else
        broadcast a termination message to all M_js;
        terminate slaves at Level 3;
        store the reduced results;
        break;
      end if
    end if
  else
    M_js receive a message from M_0;
    if not a termination message **then**
      run one DIRECT iteration (reduce intermediate results);
    else
      break;
    end if
  end if
end while
M_0 sends the final results to R_i;

Pseudo code 3.1

At Level 2, mini masters collaborate on identifying the potentially optimal box set on a shared data structure in a global addressing space based on GPSHMEM [15]. This algorithm step is one of most interesting challenges in the parallelization of DIRECT. Two sets of global shared data structures `OPTSET` and `OPTSET_INDEX` (structures in solid lines shown in Figure 3.1) are used. The structures in dashed lines (`SETLINK_INDEX` and additional `OPTSET`s) will be implemented in the next version to enlarge `OPTSET` at run time. Basically, `SETLINK_INDEX` holds a list of pointers to dynamically allocated `OPTSET`s in the global addressing space. For the maximum flexibility in adding `OPTSET`s, `SETLINK_INDEX` can be implemented as a local linked list on each processor at Level 2.

Both structures `OPTSET` and `OPTSET_INDEX` are allocated and distributed across all the mini masters, which use one-sided communication operations such as "put" and "get" to access shared data. These one-sided operations provide a direct access to remote memory with less interaction between communicating parties. At each iteration, M_j puts all the boxes with the smallest function values for different box sizes to

its own portion in OPTSET and updates its index in OPTSET_INDEX. Next, M_0 gets all boxes in OPTSET and merges the boxes with the same size. After M_0 finds all potentially optimal boxes, it balances the number of boxes for each M_j's portion in OPTSET. Detailed algorithm steps are in Pseudo code 3.2. Finally, each M_j gets its portion of the workload, removes some boxes (if any) that are assigned to other mini masters, and starts processing each potentially optimal box. Each box is tagged with a processor ID and other indexing information to be tracked by its original owner. To minimize the number of local box removals and additions, M_0 restores the boxes back to their contributors before it starts load adjustment. This also guarantees maximum data locality on each processor. On each processor, a set of local data structures for storing and processing boxes are reused from the sequential DIRECT implementation described in [10].

```
copy M_0's portion in OPTSET to LOCALSET;
merge boxes from M_j's portion in OPTSET
to LOCALSET;
find N_box potentially optimal boxes in LOCALSET;
inLOCALSET, restore boxes given by M_j to its
portion in OPTSET;
avgload := ⌈(N_box/N_proc)⌉;
i := 0;
while TRUE
  if i = N_proc − 1 break;
  if OPTSET_INDEX(i+1)< avgload then
    i_1 := i;
    while TRUE
      underload := avgload− OPTSET_INDEX(i+1);
      i_1 := (i_1 + 1)%N_proc;
      if i_1 = i break;
      if OPTSET_INDEX(i_1+1) > avgload then
        overload := OPTSET_INDEX(i_1+1) −avgload;
        if overload ≥ underload then
          shift enough load over;
          OPTSET_INDEX(i+1) := avgload;
          OPTSET_INDEX(i_1+1) :=
              OPTSET_INDEX(i_1+1) −underload;
          break;
        else
          shift some and look for more;
          OPTSET_INDEX(i+1) :=
              OPTSET_INDEX(i+1)+overload;
          OPTSET_INDEX(i_1+1) := avgload;
        end if
      end if
    end while
  end if
  i := i + 1
end while
```

Pseudo code 3.2

As shown in Pseudo code 3.2, a centralized dynamic load balancing strategy is applied at Level 2 with shared data structures in GPSHMEM. At Level 3, workload balancing of processors is also centralized with a master-slave model. For Level 2, the workload is box subdivision on the subdomain assigned at Level 1. The root mini master adjusts the workload in the globally shared structure, and each mini master subdivides its share of potentially optimal boxes and distributes the function evaluation tasks down to its slaves at Level 3. Workload is spread from the subdomain master to mini masters at Level 2 and their slaves at Level 3. In some way, this is similar to a sender-initiated strategy in MLB (multilevel load balancing) defined in [13], where workload is sent down to a $\lfloor M \rfloor$-ary tree structure. Although the control mechanism is simple, this strategy suffers a common bottleneck problem with other centralized methods. A distributed control version will be considered in future work.

4. Test Problems

Two test functions used in the present work are the n-dimensional Griewank function and a quartic function (one-dimensional instances are plotted in Figures 4.1 and 4.2). It is observed that both functions have multiple local minima. The n-dimensional Griewank function

$$f(x) = 1 + \sum_{i=1}^{n} \frac{{x_i}^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \qquad (4.1)$$

where $d > 0$ is a constant to adjust the noise, has a unique global minimum at $x = 0$, and numerous local minima. The larger the value of d, the deeper the minima values are. The numerical results in the next section are for an initial box $[-10, 90]^n$ and $d = 500$. The second test function is an n-dimensional quartic function

$$f(x) = \sum_{i=1}^{n} [2.2(x_i + 0.3)^2 - (x_i - 0.3)^4]. \qquad (4.2)$$

The quartic function is considered in the box $[-2, 3]^n$, $n \geq 2$; the global minimum occurs at a vertex of this box.

The third test problem is a parameter estimation problem for a budding yeast cell cycle (ordinary differential equation) model in systems biology. The budding yeast start and finish of mitosis is modeled. The start of mitosis model is described in [3], but the finish transition has not been published. This model

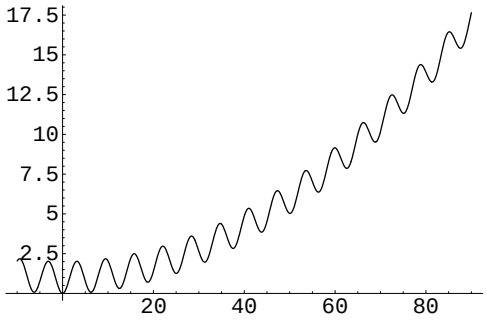

Figure 4.1. One-dimensional Griewank function with parameter $d = 500$.

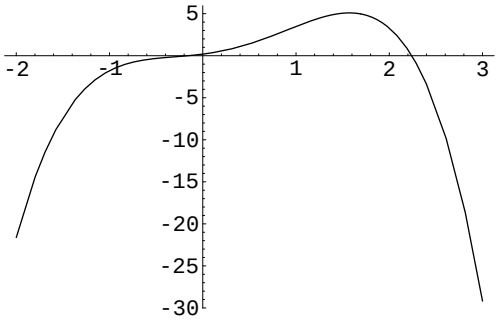

Figure 4.2. One-dimensional quartic function.

is of great significance to theoretical biologists. The model contains 154 parameters and 116 experimental data points. This problem is under specified since the number of parameters is greater than the number of experimental data points. More data exists for this model and will be used for production runs in future work. However, this version is adequate for testing the parallel DIRECT implementation. Numerical solution of the ODE model to obtain all the data for comparison to experiments takes approximately 37 seconds. During that time the ODE system is solved once for each experimental datum with the parameters modified to match the experimental setup. The result from a simulation is viable, if the budding yeast lives, and inviable, if the budding yeast dies. The objective function is the number of mismatches between the model predicted and experimentally recorded viability.

5. Simulation and Results

All the simulations were run on a 200-node Linux cluster. The list of assigned processors can be sorted to map the grouped processes (at each level) to adjacent processors. The LAM/MPI package was chosen for its support of dynamic process management in the MPI-2 standard.

Three groups of simulations were designed to evaluate the present work. The first group validated

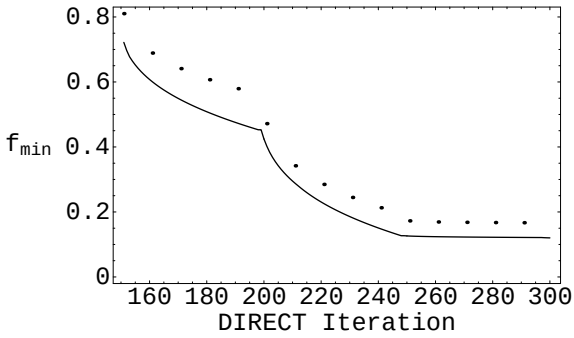

Figure 5.1. Comparison of optimization results generated by single-start DIRECT (dotted curve) and by multistart parallel DIRECT (solid curve) for the n-dimensional Griewank function, $n = 50$.

the optimization results obtained for the two test functions described in the previous section. The second group measured the parallel efficiency of the present work on 16, 32, and 64 processors. The last group measured the parallel efficiency for the budding yeast cell cycle model with 2, 36, 71, and 141 processors.

Figures 5.1 and 5.2 show the optimization progress for the original single-start DIRECT (dotted line) and for the transformed multistart parallel DIRECT (solid line), where f_{min} was reduced from subdomain masters at each iteration. For the 50-dimensional Griewank function, the objective function decreases faster in the case of the multistart parallel DIRECT algorithm as the number of iterations grows. In contrast, the quartic function progresses similarly for the single-start and the multistart parallel algorithm, since the global minimum occurs at a vertex of the bounding box. Therefore, the multistart algorithm performs better than the single-start algorithm for objective functions with many local minima. For problems with unknown structures (as most engineering design problems are), the multistart algorithm searchs subdomains in parallel, thus providing a higher probability of locating the global optimum.

Speedup is defined in [7] as "the ratio of the time taken to solve a problem on a single processing element to the time required to solve the same problem on a parallel computer with p identical processing elements". In the present work, speedup is computed relative to a *base*, the smallest number of processors required for the hierarchical parallel scheme. Because the 3-level parallel scheme does not have an exact limit for computation cost (as mentioned in Section 3), the time was measured for the lower two levels of a single subdomain tree,

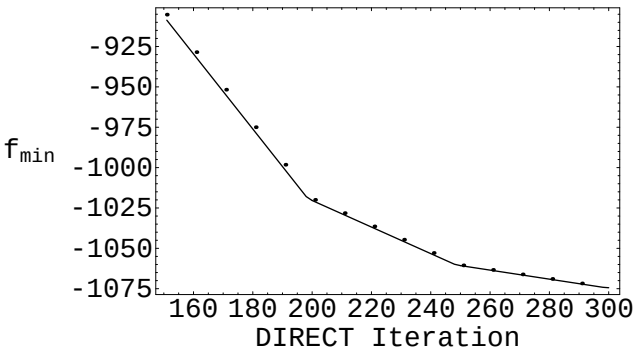

Figure 5.2. Comparison of optimization results generated by single-start DIRECT (dotted curve) and by multistart parallel DIRECT (solid curve) for the n-dimensional quartic function, $n = 50$.

neglecting any interaction with the ring at Level 1. The formulas for S^2, M, and σ still apply, but only the processors assigned to one subdomain tree (working on the entire problem domain) are used for performance evaluation. When total processor number $P \in [16, 64]$, the 3-level framework splits the domain into four parts according to Equation 3.1. Different ways of splitting the problem domain result in different search problems, so using 16, 32, and 64 processors guarantees the same search problem for the efficiency test. Therefore, the 2-level subdomain tree has $p = 3$, 6, and 15 processors accordingly. The smallest number of processors possible for a subdomain tree is $base = 3$. The parallel efficiency E is defined as

$$E = \frac{S_r}{p/base}, \tag{5.1}$$

where $S_r = \text{Time}_{base}/\text{Time}_p$ is the relative speedup.

Table 5.1 shows some preliminary parallel performance data for the Griewank function and quartic function using $p = 3$, 6, and 15 processors in a 2-level subdomain tree, which is a part of $P = 16$, 32, and 64 processors under the 3-level parallel scheme. As the maximum number of iterations I grows, the parallel efficiency E increases. Superlinear speedups ($E > 1$) (highlighted values) imply that the data generated by DIRECT may be too large to fit into the memory of the *base* number of processors. This certainly degrades its performance due to paging operations from disk. When $P = 16$, a single mini master holds all the data on a processor and sends the function evaluations to two slave processes. In the case of $P = 32$ and $P = 64$, more mini masters are spawned to hold the data, thereby sharing the huge memory burden.

TABLE 5.1. PARALLEL TIMING (IN SECONDS) AND EFFICIENCY RESULTS FOR DIFFERENT NUMBERS I OF DIRECT ITERATIONS FOR n-DIMENSIONAL GRIEWANK FUNCTION AND QUARTIC FUNCTION WITH $n = 100$ IN A 2-LEVEL SUBDOMAIN TREE ON 6 AND 15 PROCESSORS WITH $base = 3$ PROCESSORS.

	Griewank Function				
I	3	6	$E(6)$	15	$E(15)$
20	41.81	33.75	0.62	28.74	0.29
60	131.65	97.60	0.67	80.59	0.33
100	719.34	283.20	**1.27**	137.63	**1.05**

	Quartic Function				
I	3	6	$E(6)$	15	$E(15)$
50	55.22	51.00	0.54	50.82	0.22
100	652.60	168.11	**1.94**	159.36	0.82
200	3742.65	776.33	**2.41**	248.45	**3.01**

The budding yeast parameter estimation problem generates a large amount of work (function evaluations) per iteration, so a different processor assignment scheme is applied to put more processors at Level 3. For this efficiency test, the stopping criterion is the number of function evaluations instead of the number of iterations for it took approximately 17 hours to finish 5 iterations with 238397 evaluations on 141 processors (1 mini master and 140 slaves). Figure 5.3 plots the parallel efficiency for $N_{eval} = 309$ function evaluations running on a 2-level subdomain tree with a single mini master and W (1, 35, 70, 140) slave(s). The efficiency decreases as the number of processors increases and a superlinear speedup is observed for $W = 35$. This indicates that (i) more function evaluations are needed to balance the load on a larger number of processors and (ii) an optimal W value may be found in the range [60,70], where $E \approx 1$. When N_{eval} is not large enough, a load imbalance occurs and processor time is spent mostly in waiting rather than productively computing. Thus, the processor assignment scheme is crucial to achieve a good parallel efficiency. In future work, a strategy of isolating problem-dependent factors will be developed to find an optimal processor assignment scheme.

6. Conclusion and Future Work

Key contributions of the present work are i) the transformation from single-start to multistart, ii) the mixed programming model, and iii) the dynamic processor assignment.

In future work, a tree topology will be considered in place of the current ring implementation at Level 1. At Level 2, a new set of data structures `SETLINK_INDEX` and its associated `OPTSET`s will be

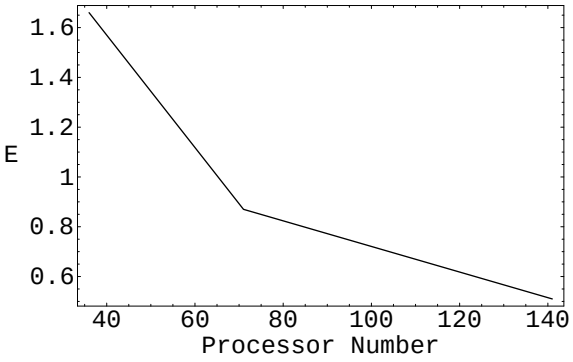

Figure 5.3. Parallel efficiency results for $N_{eval} = 309$ function evaluations in the budding yeast cell cycle model on 2, 36, 71, and 141 processors ($base = 2$).

implemented to add flexibility for enlarging the global addressing space dynamically, as the number of box sizes is unpredictable. Second, a distributed control version for load balancing, desirable to eliminate the bottleneck at the master, will be developed. Third, a queue with adjustable size entries can be used to hold multiple function evaluation tasks on processors at Level 3 to reduce communication overhead. The size of the queue entries depends on the problem granularity. When the ratio of computation to communication is higher, the entry size in the queue is smaller. Finally, an optimal processor assignment scheme will be developed to achieve a better parallel efficiency.

Acknowledgments

This work was supported in part by AFRL Grant F30602–01–2–0572. The authors gratefully acknowledge technical assistance provided by Dr. Calvin J. Ribbens on the 200 node Linux cluster at the Laboratory for Advanced Scientific Computing and Applications (LASCA).

References

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

[2] M. C. Bartholomew-Biggs, S. C. Parkhurst, and S. P. Wilson, "Using DIRECT to solve an aircraft routing problem", *Computational Optimization and Applications*, vol. 21, no. 3, pp. 311–323, 2002.

[3] Katherine C. Chen, Attila Csikasz-Nagy, Bela Gyorffy, John Val, Bela Novak, and John J. Tyson, "Kinetic analysis of a molecular model of the budding yeast cell cycle", *Molecular Biology of the Cell*, vol. 11, pp. 369–391, January, 2000.

[4] W. R. Esposito and C. A. Floudas, "Global optimization in parameter estimation of nonlinear algebraic models via the Error-In-Variables approach", *Ind Eng. Chemistry and Research*, vol. 37, pp. 1841–1858, 1998.

[5] J. M. Gablonsky, "Modifications of the DIRECT algorithm", PhD thesis, Department of Mathematics, North Carolina State University, Raleigh, NC, 2001.

[6] C. Gau and M. A. Stadtherr, "Nonlinear parameter estimation using interval analysis", in *AIchE Symposium*, vol. 94, no. 320, pp 445-450, 1999.

[7] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Pearson Education Limited, 2nd Edition, 2003.

[8] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced features of the message-passing interface*, The MIT Press, Cambridge, Massachusetts, London, England, 1999.

[9] J. He, A. Verstak, L. T. Watson, T. S. Rappaport, C. R. Anderson, N. Ramakrishnan, C. A. Shaffer, W. H. Tranter, K. Bae, and J. Jiang, "Global optimization of transmitter placement in wireless communication systems", in *Proc. High Performance Computing Symposium 2002*, A. Tentner (ed.), Soc. for Modeling and Simulation International, San Diego, CA, pp. 328–333, 2002.

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

[11] D. R. Jones, C. D. Perttunen, and B. E. Stuckman, "Lipschitzian optimization without the Lipschitz constant", *J. Optim. Theory Appl.*, vol. 79, no. 1, pp. 157–181, 1993.

[12] D. J. Mavriplis, "Parallel performance investigation of an unstructured mesh Naiver-Stokes solver", *Internat. J. High Performance Computing Appl.*, vol. 16, no. 4, pp. 395–407, 2002.

[13] V. Kumar, A. Y. Grama, and N. R. Vempaty, "Scalable load balancing techniques for parallel computers", *J. Parallel Distributed Computing*, vol. 22, pp. 60–79, 1994.

[14] C. G. Moles, P. Mendes, and J. R. Banga, "Parameter estimation in biochemical pathways: a comparison of global optimization methods", *Genome Res.*, vol. 13, pp. 2467–2474, 2003.

[15] K. Parzyszek, J. Nieplocha, and R. A. Kendall, "A generalized portable SHMEM library for high performance computing", in *12th IASTED International Conference Parallel and Distributed Computing and Systems (PDCS)*, pp. 401–406, 2000.

[16] R. Rabenseifner, and G. Wellein, "Communication and optimization aspects of parallel programming models on hybrid architectures", *Internat. J. High Performance Computing Appl.*, vol. 17, no. 1, pp. 49–62, 2003.

[17] M. Ruiz, O. Lopera, and C. de la Plata, "Component-based derivation of a parallel stiff ODE solver", *Internat. J. Parallel Programming*, vol. 30, no. 2, pp. 99–148, 2002.

[18] L. T. Watson and C. A. Baker, "A fully-distributed parallel global search algorithm", *Engineering Computations*, vol. 18, no. 1/2, pp. 155–169, 2001.