Parallel Global Aircraft Configuration Design Space Exploration

CHUCK A. BAKER, LAYNE T. WATSON, BERNARD GROSSMAN, WILLIAM H. MASON
Multidisciplinary Analysis and Design (MAD) Center for Advanced Vehicles
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0203
USA
ltw@cs.vt.edu http://www.aoe.vt.edu/mad/mads.html

RAPHAEL T. HAFTKA
Department of Aerospace Engineering, Mechanics and Engineering Science
University of Florida
Gainesville, Florida 32611-6250
USA
haftka@ufl.edu

Abstract: – The preliminary design space exploration for large, interdisciplinary engineering problems is often a difficult and time-consuming task. General techniques are needed that efficiently and methodically search the design space. This work focuses on the use of parallel load balancing techniques integrated with a global optimizer to reduce the computational time of the design space exploration. The method is applied to the multidisciplinary design of a High Speed Civil Transport (HSCT). A modified Lipschitzian optimization algorithm generates large sets of design points that are evaluated concurrently using a variety of load balancing schemes. The load balancing schemes implemented in this study are: static load balancing, dynamic load balancing with a master-slave organization, fully distributed dynamic load balancing, and fully distributed dynamic load balancing via threads. All of the parallel computing schemes have high parallel efficiencies. When the variation in the design evaluation times is small, the computational overhead needed for fully distributed dynamic load balancing is substantial enough so that it is more efficient to use a master-slave paradigm. However, when the variation in evaluation times is increased, fully distributed load balancing is the most efficient.

Key-Words: – Nonlinear programming, Global optimization, Parallel computation, Aerospace

1 Introduction

Previous work [1] has shown that the design space of the HSCT configuration is complex. Local minima occur because the feasible design domain is nonconvex. Running local optimizations from a sufficient number of starting points distributed throughout the design space requires a large number of function evaluations and still does not guarantee that the promising regions of the design space will be explored. A global optimizer is needed that is able to judiciously balance the local and global searches, insuring a complete space investigation, while keeping the number of function evaluations to a minimum.

Global optimization in high dimensional spaces requires many thousands of analyses, and this may not be possible without parallel computation. Fortunately, many global optimization algorithms can take advantage of analyzing many design points in parallel, thus allowing relatively simple course grain parallelization. However, the question of how to best manage the evaluation and distribution of points on parallel computers is unresolved for exploratory multidisciplinary engineering design studies. The objective of the present paper is to explore several options for distributing the work among the nodes of a parallel computer.

Section 2 describes the aircraft design problem, Section 3 gives the direct search global optimization algorithm, and Section 4 presents detailed pseudo code for the parallel version of the algorithm from Section 3. Sections 5 and 6 give parallel load balancing and termination detection strategies. Parallel performance results are presented and discussed in Section 7. Sections 8 and 9 describe a modified, more aggressive, global optimization algorithm and its parallel performance. All the results are for large SGI Origin 2000 systems. Section 10 summarizes the results.

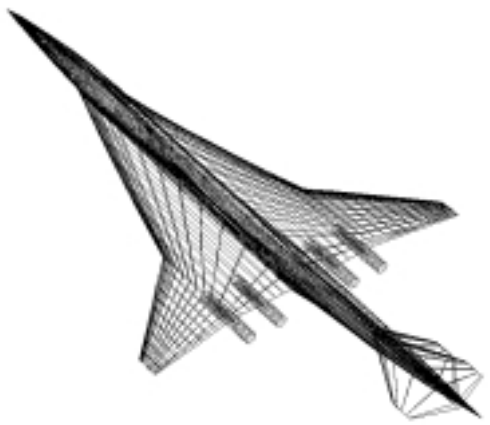

Fig. 1. Typical HSCT configuration.

2 HSCT Design Problem

The design problem considered is the optimization of a HSCT configuration [10], [11] to minimize takeoff gross weight (TOGW) for a range of 5500 nautical miles and a cruise Mach number of 2.4, while carrying 251 passengers. A typical HSCT configuration is seen in Figure 1. The choice of gross weight as the objective function directly incorporates both aerodynamic and structural considerations, in that the structural design directly affects aircraft empty weight and drag, while aerodynamic performance dictates drag and thus the required fuel weight.

To successfully perform aircraft configuration optimization, it is important to have a simple, but meaningful, mathematical characterization of the geometry of the aircraft. This paper uses a model that defines the HSCT design problem using the twenty-eight design variables listed in Table 1. Twenty-four of the design variables describe the geometry of the aircraft and can be divided into six categories: wing planform, airfoil shape, tail areas, nacelle placement, and fuselage shape. In addition to the geometric parameters, four variables define the idealized cruise mission: mission fuel, engine thrust, initial cruise altitude, and constant climb rate used in the range calculation.

For the optimizer used here, upper and lower bounds had to be set on all n of the design variables. These bounds form a n-dimensional rectangular shaped set in the design space, referred to as the design *box*. In order to ensure that a thorough design space exploration was being conducted, the bounds were chosen to include as wide a range of designs as realistically possible. The edges of the design box were set near the limits of physically impossible designs (overlapping geometries, negative chord lengths) or the assumptions of the numerical analyses being used.

Sixty-eight geometry, performance, and aerodynamic constraints, listed in Table 2, are included in the optimization. Aerodynamic and performance constraints can only be assessed after a complete analysis of the HSCT design; however, the geometric constraints can be evaluated using algebraic relations based on the 28 design variables.

Methods of varying fidelity are used for the aerodynamic and structural analyses in the constraint evaluations. The methods used to calculate the drag components used in the drag calculation and their corresponding ranges are described in [6], [7]. The aerodynamics calculations are based on the Mach box method [4], [3], and the Harris wave drag code [5]. A simple strip boundary layer friction estimate is implemented as in [7]. A vortex lattice method with vortex lift and ground effects included [2] is used to calculate landing angle of attack. Structural weights are calculated by the FLOPS [12] weight equations. Each of these analysis methods uses iterative loops or discretization methods that can cause differences in the computational time needed to evaluate (calculate the objective function and constraint values) different HSCT designs.

3 Lipschitzian Global Optimizer (DIRECT)

The global optimizer selected to explore the design space is a Lipschitzian unconstrained optimization algorithm that (effectively) uses all possible values of the Lipschitz constant [8]. By using different values of the constant, which can be viewed as an upper limit on the variation of the function, equal emphasis is

Table 1. HSCT configuration design variables.

Index	Description	Index	Description
1	Wing root chord (ft)	15	Fuselage restraint 2, x (ft)
2	Leading edge (LE) break point, x (ft)	16	Fuselage restraint 2, y (ft)
3	LE break point, y (ft)	17	Fuselage restraint 3, x (ft)
4	Trailing edge (TE) break point, x (ft)	18	Fuselage restraint 3, y (ft)
5	LE wing tip, x (ft)	19	Fuselage restraint 4, x (ft)
6	Wing tip chord (ft)	20	Fuselage restraint 4, y (ft)
7	Wing semi-span (ft)	21	Nacelle 1 location (ft)
8	Chordwise location of max. thickness	22	Nacelle 2 location (ft)
9	LE radius parameter	23	Vertical tail area (ft^2)
10	Airfoil t/c ratio at root, (%)	24	Horizontal tail area (ft^2)
11	Airfoil t/c ratio at LE break, (%)	25	Thrust per engine (lb)
12	Airfoil t/c ratio at LE tip, (%)	26	Flight fuel (lb)
13	Fuselage restraint 1, x (ft)	27	Starting cruise/climb altitude (ft)
14	Fuselage restraint 1, y (ft)	28	Supersonic cruise/climb rate (ft/min)

Table 2. HSCT optimization constraints.

Index	Constraint	Index	Constraint
1	Fuel volume $\leq$ 50% wing volume	35	C_L at landing speed ≤ 1
2	Wing root TE $\leq$ Tail LE	36–53	Section C_L at landing ≤ 2
3–20	Wing chord $\geq$ 7.0 ft	54	Landing angle of attack $\leq 12^\circ$
21	LE break within wing semi-span	55–58	Engine scrape at landing
22	TE break within wing semi-span	59	Wing tip scrape at landing
23	Root chord t/c ratio $\geq$ 1.5%	60	TE break scrape at landing
24	LE break chord t/c ratio $\geq$ 1.5%	61	Rudder deflection $\leq 22.5^\circ$
25	Tip chord t/c ratio $\geq$ 1.5%	62	Bank angle at landing $\leq 5^\circ$
26–30	Fuselage restraints	63	Tail deflection at approach $\leq 22.5^\circ$
31	Wing spike prevention	64	Takeoff rotation to occur $\leq V_{min}$
32	Nacelle 1 inboard of nacelle 2	65	Engine-out limit with vertical tail
33	Nacelle 2 inboard of semi-span	66	Balanced field length $\leq$ 11000 ft
34	Range $\geq$ 5500 nautical miles		
67–68	Mission segments: thrust available $\geq$ thrust required		

placed on the local and global search being performed by the optimizer. This algorithm is called DIRECT because the algorithm is a direct search technique and as an acronym for *di*viding *rect*angles, one of the primary operations in the procedure.

The algorithm begins by scaling the design box to a n-dimensional unit hypercube. The center point of the hypercube is evaluated and then points are sampled at one-third the cube side length in each coordinate direction from the center point. Depending on the direction with the smallest function value, the hypercube is then subdivided into smaller rectangles, with each sampled point becoming the center of its own n–dimensional rectangle or box. All boxes are identified by their center point and their function value at that point.

From there the algorithm loops in a procedure that subdivides each of the boxes in the set in turn until termination or convergence. By using different values of the Lipschitz constant, a set of potentially optimal boxes is identified from the set of all boxes. These potentially optimal boxes are sampled in the directions of maximum side length, to prevent boxes from becoming overly skewed, and subdivided again based on the directions with the smallest function value. If the optimization continues indefinitely, all boxes will eventually be subdivided meaning that all regions of the design space will be investigated. The algorithm [8] is as follows:

1. Normalize the search space to be the unit hypercube. Let c_1 be the centerpoint of this hypercube and evaluate $f(c_1)$.
2. Identify the set S of potentially optimal rectangles (those rectangles defining the bottom of the convex hull of a scatter plot of rectangle diameter versus $f(c_i)$ for all rectangle centers c_i) as in Figure 2.

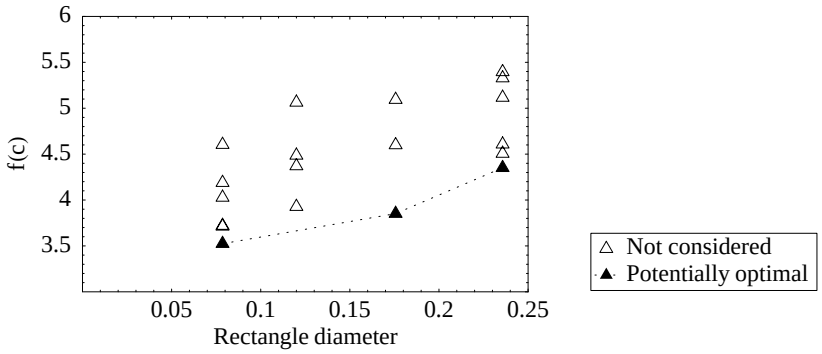

Fig. 2. DIRECT point selection.

3. For all rectangles $j \in S$:
   3a. Identify the set I of dimensions with the maximum side length. Let δ equal one-third of this maximum side length.
   3b. Sample the function at the points $c \pm \delta e_i$ for all $i \in I$, where c is the center of the rectangle and e_i is the ith unit vector.
   3c. Divide the rectangle containing c into thirds along the dimensions in I, starting with the dimension with the lowest value of $f(c \pm \delta e_i)$ and continuing to the dimension with the highest $f(c \pm \delta e_i)$.
4. Repeat 2.–3. until stopping criterion is met.

Two important issues in using the algorithm are how to determine convergence and incorporate constraint values. For this study, the algorithm was run for a fixed number of loops or iterations. Since the purpose of the optimization was to identify promising regions of the design space, it was unnecessary to tightly converge to a global optimum. Constraints were accounted for through the use of a simple penalty function, as follows. Let x be the 28-dimensional design vector, $f(x)$ the TOGW, and $g_i(x) \leq 0$ the constraints in Table 2. The constrained optimization problem

$$\min f(x) \quad \text{subject to } g_i(x) \leq 0, \ i = 1, ..., 68,$$

is converted to the unconstrained optimization problem

$$\min f(x) + 10 \sum_{i=1}^{68} \max\{0, g_i(x)\}.$$

4 Parallel DIRECT

Since DIRECT generates large sets of points to be evaluated at each iteration, parallel computers can easily be used to reduce the computation time. The parallel implementations that were studied employ a variety of methods to control processor work distribution, to generate the point set, and to balance the function evaluation work load. Below is pseudo code for a parallel implementation of DIRECT using N_p processors, incorporating fully distributed control (DLBDC), which is described in the next section.

$iteration := 1$
while $iteration \leq$ maximum iteration
  find potentially optimal point set, local C_{opt}, from previously evaluated box center points, C_{eval}
  if P_0 **then**
    gather C_{opt} from all processors
    find global C_{opt} from local C_{opt} sets
    broadcast global C_{opt} set to all processors
  end if
  remove points in local C_{opt} not in global C_{opt}
  if number of points in global $C_{opt} \leq N_p \log N_p$, **then**
    total tasks $N_{tasks} :=$ number of points in local C_{opt}; task counter $i_{task} := 1$

```
        while termination not detected
            if i_task ≤ N_tasks then
                sample around all C_opt(i_task) to create C_new(i_task)
                evaluate function at all C_new(i_task)
                i_task := i_task + 1
            else
                if outgoing work request is not pending, then
                    generate random processor number, P_rand; send work request to P_rand
                end if
            end if
            process message of each type (incoming work request, outgoing work request reply,
                token pass, etc.) received;
            if outgoing work request reply received then increment N_tasks by number of tasks received;
            if work request received then
                if N_tasks − i_task > 1 then
                    send ⌊(N_tasks − i_task)/2⌋ tasks to requesting processor;
                    decrement N_tasks by ⌊(N_tasks − i_task)/2⌋ tasks;
                else
                    send 0 tasks to requesting processor;
                end if
            end if
        end while
    else
        sample around all C_opt to create C_new
        total tasks N_tasks := number of points in C_new; task counter i_task := 1
        while termination not detected
            if i_task ≤ N_tasks then
                evaluate function at C_new(i_task)
                i_task := i_task + 1
            else
                if outgoing work request is not pending, then
                    generate random processor number, P_rand; send work request to P_rand
                end if
            end if
            process message of each type (incoming work request, outgoing work request reply,
                token pass, etc.) received;
            if outgoing work request reply received then increment N_tasks by number of tasks received;
            if work request received then
                if N_tasks − i_task > 1 then
                    send ⌊(N_tasks − i_task)/2⌋ tasks to requesting processor;
                    decrement N_tasks by ⌊(N_tasks − i_task)/2⌋ tasks;
                else
                    send 0 tasks to requesting processor;
                end if
            end if
        end while
    end if
    if P_0 then
        gather C_new from all processors
        sort C_new points by parent processor rank
        scatter each C_new point to its parent processor
    end if
    set new box side lengths for C_new and its parent C_opt points; append all C_new to C_eval
    iteration := iteration + 1
end while
```

5 Load Balancing Strategies

As the potentially optimal boxes are sampled in their respective directions during the DIRECT optimization, a typically large set of new design points, or tasks, that need to be evaluated is created. It is these tasks in this set of designs that are load balanced. Processor communications were performed in the optimization algorithm through the use of the Message Passing Interface (MPI) [13], a message passing standard. MPI was chosen because, as a communications protocol, it is platform independent, thread-safe, and a widely accepted standard.

In the master-slave implementation of dynamic load balancing, one processor, the master, makes all of the calculations for box manipulation in DIRECT and controls the distribution of tasks to be evaluated by the HSCT code on the slave processors. The master processor begins with the set of all boxes, finds the potentially optimal boxes, and then samples inside of these boxes to generate the set of tasks. It then distributes one task to each slave processor. When a slave processor completes the evaluation of its task it returns the function value back to the master and receives another task, if available. The biggest potential drawback to using this method is that there is a chance for a communication bottleneck caused by slave processors simultaneously requesting work from the master. To investigate this effect, a version of the master-slave implementation was also used that distributes the tasks in bins of 10.

For the static load balancing case, the processors only communicate with each other when finding the set of potentially optimal boxes and initially distributing the tasks. At the start of a DIRECT loop each processor finds its own local set of potentially optimal boxes. The root processor, P_0, gathers all of the local potentially optimal sets from the other processors and finds the global set of potentially optimal boxes. This processor creates the set of new tasks from the global set of potentially optimal boxes. The new tasks are equally distributed to all of the processors and the individual processors evaluate every task in their set of new tasks. The problem inherent to static load balancing is that differences in evaluation times can cause some processors to finish their tasks early and sit idle, while other processors continue to work on their tasks.

The interprocessor communications used for the DIRECT box manipulation by the fully distributed version of dynamic load balancing are the same as those performed by the static version, with the added capability of task migration to processors that have finished their tasks. The dynamic load balancing algorithm is based on that of previous work [9], employing random polling for the redistribution of tasks and token passing to terminate the load balancing process. Once task evaluation is started by a processor, it evaluates a single task and then processes any messages received during the evaluation of the task. The cycle of evaluating and communicating is continued until the processor runs out of work, in which case it begins sending work requests to a randomly selected processor either until work is found or the termination is detected. If a work request is received by a processor, half of its remaining tasks are transferred to the requesting processor.

A dynamic load balancing strategy is also implemented that uses threads in the fully distributed version. Multi-threading in the distributed version is based on the POSIX (pthreads) package. In this implementation, one thread is a worker responsible for evaluating tasks and sitting idle when no tasks are available. A second thread handles all of the message passing and processing. By exploiting concurrency at the processor level, messages can be processed at the same time as a task is being evaluated, instead of the purely sequential operations used by the distributed version without threads.

In the subsequent discussion, these load balancing strategies are referred to as static (STATIC), dynamic load balancing with the master-slave paradigm—bin size 1 (DLBMS01), dynamic load balancing with the master-slave paradigm—bin size 10 (DLBMS10), dynamic load balancing with fully distributed control (DLBDC), dynamic load balancing with fully distributed control using pthreads (DLBDCT).

6 Termination Detection

The termination detection scheme used for DLBDC and DLBDCT is the standard token wave algorithm used in [9]. Suppose there are P processors. Each processor keeps track of its state in a local flag *idle*. Initially, the flag *idle* is set to false if a processor has work or true otherwise. If at any time a processor receives work, the *idle* flag is set to false. A token is passed around, in ring fashion, to all processors. If a processor with $idle = true$ receives the token, the token is less than P, and there are no pending requests for incoming work, the token value is incremented and sent to the next processor in the ring; if the token

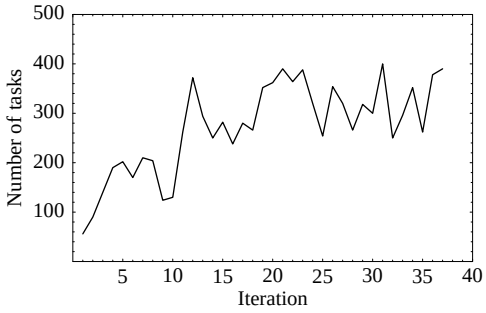

Fig. 3. History of tasks per iteration.

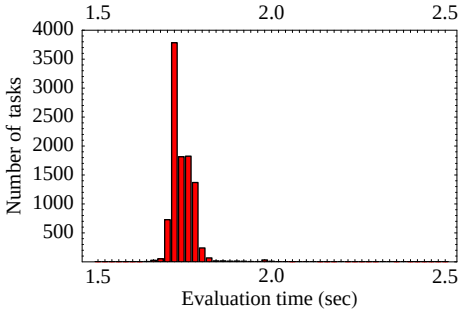

Fig. 4. Time distribution for all 10,077 tasks evaluated.

received is equal to P, then that processor terminates, and broadcasts a termination message to all other processors. If a processor with $idle = false$ receives the token, the token is set to zero. When that processor finishes its work, it passes the (zero) token along and sets $idle = true$. After all the tasks on all the processors have been completed, the token makes two complete circuits of the ring of processors, terminating at the end of the second circuit.

7 Parallel Performance, DIRECT

The parallel runs were conducted on an SGI Origin 2000 with a total of 256 CPUs. Runs were made on 4, 8, 16, 32, and 64 processors for each of the five load balancing methods. The DIRECT optimizer was terminated after 37 iterations, performing 10,077 function evaluations. The history of total tasks for each iteration is shown in Figure 3. The figure illustrates the amount of work that had to be distributed to the processors during the load balancing. Figure 4 is a histogram of the evaluation times for the 10,077 tasks. The variation in the evaluation times is relatively small, with most of the tasks taking around 1.75 seconds to complete.

The parallel efficiencies for the runs are plotted in Figure 5. Efficiency is calculated relative to a serial implementation of DIRECT. With static load balancing, the efficiency starts high (0.97) for 4 processors and then linearly decreases to 0.83 with all 64 processors. The master-slave organization (DLBMS01) of dynamic load balancing starts with a low efficiency, and then the efficiency gradually increases to be the highest of the load balancing schemes for 64 processors. The initial low values of efficiency are because, even though four processors are used, only the three slave processors are evaluating tasks. As the number of processors increases, the increased number of slave processors minimizes this effect. The master-slave organization with a bin size of 10 (DLBMS10) initially has a low efficiency like DLBMS01 then it peaks at 0.80 for 8 processors. From then as the number of processors used increases, the efficiency plateaus at 0.57. The fully distributed version with dynamic load balancing performs the best up to 32 processors and then the efficiency drops to 0.84 when using 64 processors, slightly below that of DLBMS01 and slightly above that of STATIC. This is attributable to both the short average time per task and the relatively small amount of total work assigned to each of the 64 processors. Also, a peculiarity was observed in that DLBDC either ran at an efficiency of 0.84 or 0.78 (shown on plot). The distributed version with threads performs the worst of all the methods, rapidly decreasing in efficiency as the number of processors used is increased. This behaviour was not observed for pthreads on the Intel Paragon reported in [9], and thus is more likely a reflection of the SGI pthreads implementation than of an inherent characteristic of pthreads.

To provide insight into why the distributed versions of the code were not performing as well as expected for a large number of processors, a plot of the individual processor load for a complete optimization was made (Figure 6) for the 64 processor case. From this plot it is clear that the master-slave organization (DLBMS) does the best job of load balancing, the curve being nearly horizontal. The load distribution for the distributed version without threads (DLBDC) falls directly on top of the curve for the static load balancing case (STATIC). This is due to the variation in function evaluation times being small enough that no tasks get transferred between processors, so DLBDC effectively becomes static load balancing. This effect does not appear when the number of processors is small because each processor has a larger set and with a large set the differences in evaluation times are magnified enough to where dynamic load balancing does take place. The time spent evaluating tasks for the threaded code DLBDCT is almost double that of all other methods. It was found that having the communicator thread running continuously sufficiently

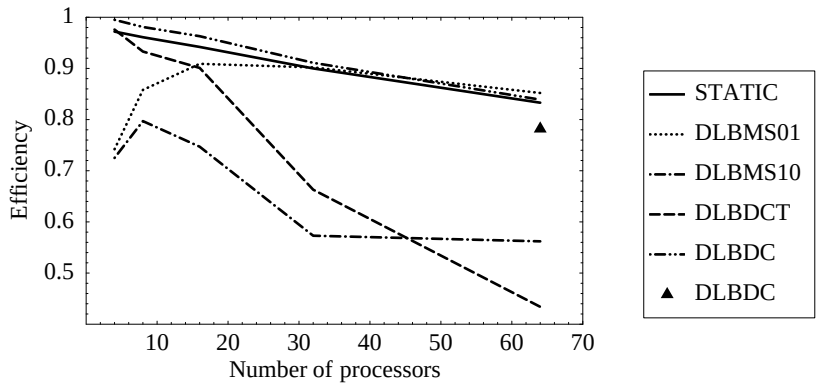

Fig. 5. Parallel efficiencies for 4, 8, 16, 32, and 64 processor cases.

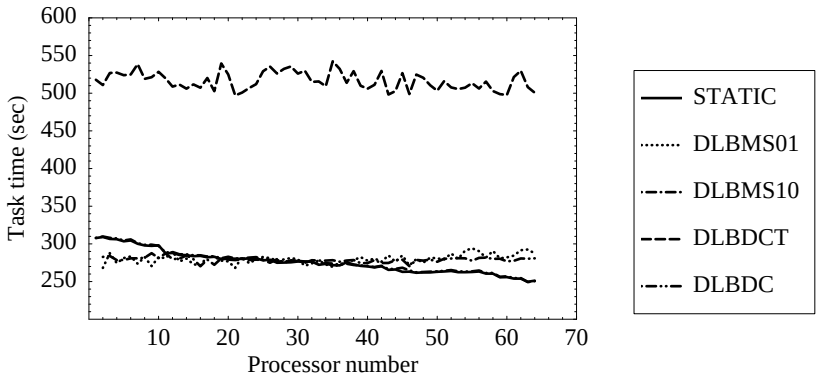

Fig. 6. Individual processor load, 64 processor case.

impeded the performance of each processor on the Origin to cause this noticeable rise in function evaluation times.

An investigation of the two discrete run times for DLBDC revealed that the cause was the way that memory is assigned for the Origin. The operating system assigns processes to memory banks (MLDs) and to CPUs in nodes. Processes can migrate between nodes searching for free CPUs. For the slow runs, about half the MPI processes have almost totally nonlocal memory allocation—the memory used by those migrated processes is still allocated over on another node. The detrimental effect of nonlocal memory access is apparent and significant.

8 Aggressive DIRECT

To observe the effect of larger sets of tasks for a large number of processors, a more aggressive version of the DIRECT algorithm is implemented. For the aggressive DIRECT, the idea of using the Lipschitz constants is discarded and the box with the smallest objective function for each box size existing is deemed potentially optimal and subsequently subdivided. Consequently, for the example shown in Figure 2 there will be a total of four potentially optimal boxes, instead of the three for the standard DIRECT algorithm. This change in the algorithm typically results in a much larger set of new tasks to be evaluated and load balanced at each iteration.

9 Parallel Performance, Aggressive DIRECT

The parallel runs were conducted on the same SGI Origin 2000 as the standard DIRECT. Runs were made on 8, 16, 32, 64, and 128 processors for each of the five load balancing methods. Due to the large number of points generated, the DIRECT optimizer was terminated after 20 iterations and performing 48,577 function evaluations. The history of total tasks for each iteration is shown in Figure 7. The increase in number of points at each iteration that the aggressive version provides is clearly illustrated in the figure, from an average of 272 evaluations per iteration for the standard DIRECT to 2,429 for the aggressive version. After

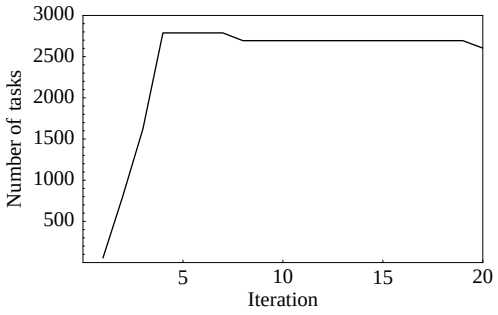

Fig. 7. History of tasks per iteration.

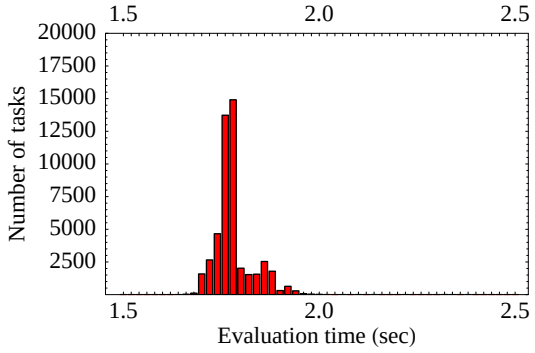

Fig. 8. Time distribution for all 48,577 tasks evaluated.

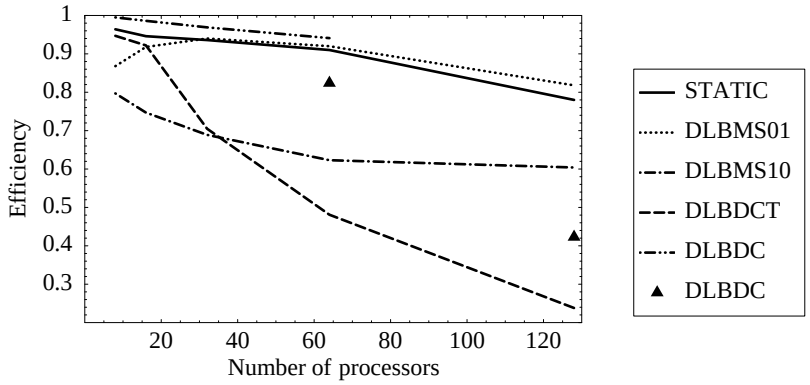

Fig. 9. Parallel efficiencies for 8, 16, 32, 64, and 128 processor cases.

four iterations the number of different box sizes becomes saturated; a new box size is being created while another is being eliminated, resulting in a plateau in the number of new tasks. Figure 8 is a histogram of the evaluation times for the 48,577 tasks. It can be seen that the variation in evaluation time has been increased as well as the number of tasks with aggressive DIRECT. The aggressive version was also able to find a better optimum HSCT design in fewer iterations than standard DIRECT, although of course the total number of evaluations and aggregate CPU time are more (the aggressive case used 86,374 seconds of serial CPU time versus 17,642 seconds for the standard DIRECT).

The parallel efficiencies for the runs using the aggressive DIRECT are plotted in Figure 9. All the load balancing methods implemented exhibit similar trends as when used with the standard DIRECT except that their efficiencies have been slightly improved. Due to the increase of variation in evaluation time, DLBDC is now the most efficient method to 64 processors, where its efficiency is 0.94. The improvement in load balancing of DLBDC over the other methods is also shown in Figure 10. The task time for DLBDCT, not shown in Figure 10, hovered around 2300 seconds, well above all the other times. The memory problems experienced with the standard DIRECT were experienced again here (Figure 9) and a valid run using 128 processors for DLBDC was not attained.

10 Conclusion

A variety of parallel load balancing strategies were successfully integrated into a global design space exploration method applied to a meaningful, complex aircraft design problem. The load balancing methods implemented ranged from simple static load balancing to fully distributed dynamic load balancing via threads. It was observed that the master-slave load balancing method was the most efficient for a large number of processors, when the variation in function evaluation times for the test problem was small. When the variation in function evaluation times is significant, as is the case for the aggressive DIRECT algorithm or inherently in other aircraft design problems [9], or as here when using a small number processors, the fully distributed dynamic load balancing method is most efficient. The use of pthreads greatly facilitates programming, but the execution efficiency of pthreads varies greatly between system implementations—from nearly invisible on the Intel Paragon to a factor of two slower on the SGI Origin.

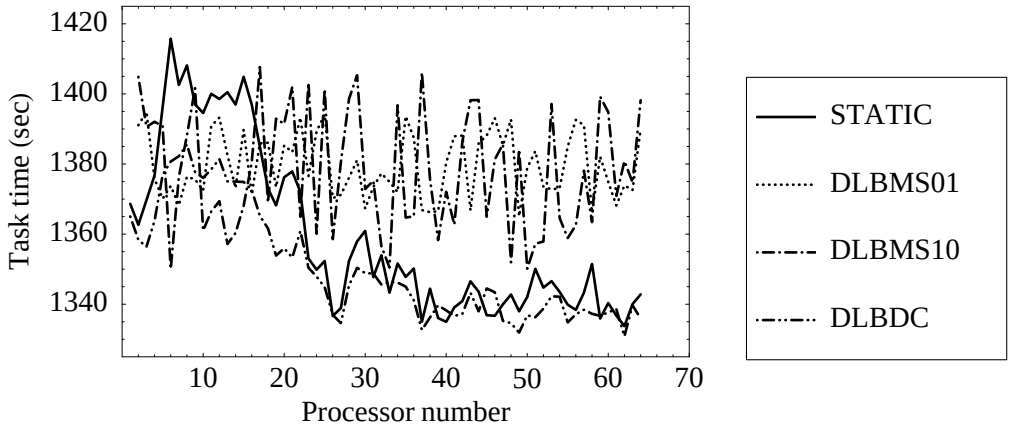

Fig. 10. Individual processor load, 64 processor case.

References:

[1] C.A. Baker, B. Grossman, R.T. Haftka, W.H. Mason, and L.T. Watson, HSCT configuration design space exploration using aerodynamic response surface approximations, in *Proceedings of 7th AIAA/-USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, Saint Louis, MO, 1998, pp. 769–777.

[2] J. Bertin and M. Smith, *Aerodynamics for Engineers*, Prentice Hall, 1989.

[3] H. Carlson, R. Mack, and R. Barger, Estimation of attainable leading edge thrust for wings at subsonic and supersonic speeds, Technical Report NASA TP-1500, 1979.

[4] H. Carlson and D. Miller, Numerical methods for the design and analysis of wings at supersonic speeds, Technical Report NASA TN D-7713, 1974.

[5] R. Harris Jr., An analysis and correlation of aircraft wave drag, Technical Report NASA TM X-947, 1964.

[6] M.G. Hutchison, W.H. Mason, R.T. Haftka, and B. Grossman, Aerodynamic optimization of an HSCT configuration using variable-complexity modeling, AIAA 31st Aerospace Sciences Meeting and Exhibit, Reno, NV, AIAA Paper 93-0101, 1993.

[7] M.G. Hutchison, E.R. Unger, W.H. Mason, B. Grossman, and R.T. Haftka, Variable-complexity aerodynamic optimization of a high-speed civil transport wing, *Journal of Aircraft*, Vol. 31, No. 1, 1994, pp. 110–116.

[8] D.R. Jones, C.D. Perttunen, and B.E. Stuckman, Lipschitzian optimization without the Lipschitz constant, *Journal of Optimization Theory and Application*, Vol. 79, No. 1, 1993, pp. 157–181.

[9] D.T. Krasteva, C. Baker, L.T. Watson, B. Grossman, W.H. Mason, and R.T. Haftka, Distributed control parallelism in multidisciplinary aircraft design, *Concurrency: Practice and Experience*, Vol. 11, 1999, pp. 435–459.

[10] P. MacMillin, O. Golovidov, W. Mason, B. Grossman, and R. Haftka, Trim, control, and performance effects in variable-complexity high-speed civil transport design, Technical Report MAD 96-07-01, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1996.

[11] P.E. MacMillin, O.B. Golovidov, W.H. Mason, B. Grossman, and R.T. Haftka, An MDO investigation of the impact of practical constraints on an HSCT optimization, AIAA 35th Aerospace Sciences Meeting and Exhibit, Reno, NV, AIAA Paper 97-0098, 1997.

[12] L.A. McCullers, Aircraft configuration optimization including optimized flight profiles, in *Proceedings of a Symposium on Recent Experiences in Multidisciplinary Analysis and Optimization*, NASA CP-2327, 1984, pp. 395–412.

[13] M. Snir, S. Otto, S. Huss-Lederman, D.W. Walker, and J. Dongarra, *MPI: The Complete Reference*, MIT Press, 1996.