10th World Congress on Structural and Multidisciplinary Optimization
May 19 - 24, 2013, Orlando, Florida, USA

Hybrid-Parallel Methods for Large-Scale Gradient-Based Structural Design Optimization

Graeme J. Kennedy[1] **and Joaquim R. R. A. Martins**[2]

University of Michigan, Department of Aerospace Engineering, Ann Arbor, MI, USA
[1]Postdoctoral Research Fellow, graeme.j.kennedy@gmail.com
[2]Associate Professor, jrram@umich.edu

1. Abstract

In this paper, we describe and evaluate the parallel implementation and performance of a hybrid parallel finite-element code for large-scale gradient-based optimization. Realistic high-fidelity structural design optimization problems for modern composite aircraft involve hundreds to thousands of load cases, tens of thousands of design variables and up to hundreds of thousands of constraints. For these optimization problems, the single most expensive computational operation is the evaluation of the constraint derivatives. This can be a significant bottleneck during the optimization, even when efficient gradient evaluation techniques are employed, such as the adjoint method. In this paper, we describe methods for analysis and gradient-evaluation that exploit the structure of these large-scale optimization problems to achieve optimal computational performance on machines built using clusters of multi-core CPUs. In this research we have extended the capabilities of our in-house parallel finite-element code, called the Toolkit for the Analysis of Composite Structures (TACS), to use a hybrid parallel architecture that combines the Message Passing Interface (MPI) and POSIX threads (Pthreads). This two-level hybrid scheme enables us to achieve finer-grain parallelism on performance-critical tasks required for gradient evaluation. In particular, we use this hybrid scheme to achieve better scalability when evaluating the partial derivatives required for the adjoint and when solving multiple adjoints simultaneously. We demonstrate the efficiency of the implementation of the new algorithms on a large-scale finite-element wing-box model.

2. Keywords

Large-scale optimization, high-performance computing, structural optimization

3. Introduction

The increasing use of high-performance parallel computers will enable the solution of large-scale, high-fidelity structural design optimization problems of increasing complexity [10]. Numerous authors have developed methods for a variety of large-scale structural design optimization problems including three dimensional topology optimization [1, 11], shape and sizing of aerospace structures [7, 8] and large-scale static aeroelastic design optimization of aircraft [6, 5, 3]. The vast majority of optimization methods for large-scale applications have employed gradient-based optimization methods with efficient analytic or semi-analytic gradient evaluation techniques. Gradient-based methods are preferred due primarily to the poor scalability of gradient-free methods with increasing numbers of design variables.

Most complex, large-scale structural design optimization problems require the simultaneous consideration of multiple design conditions to ensure that the structure can safely operate at all points within the design envelope. Frequently, the design problem will be formulated in such a manner that a new set of stress or failure constraints is added to the problem for each additional load case. Depending on the constraint formulation, this can quickly lead to design problems with thousands to tens of thousands of constraints, even for problems with a moderate number of loading conditions. As a result, the critical bottleneck in a structural optimization problem is often the evaluation of constraint gradients.

In this paper, we address the constraint gradient bottleneck by optimizing the performance of a finite-element code with large-scale gradient-based design optimization in mind. In particular, we utilize a hybrid parallel paradigm to extract better performance for gradient evaluation. Modern high-performance computers use shared memory symmetric multi-processors (SMPs). In this type of architecture, each SMP contains a group of processing cores with a local cache and shared memory hierarchy. The SMPs are typically connected through a low-latency interconnect that can be used to communicate between non-local SMP cores. As a result, there is non-uniform communication latency between any two processor cores. Furthermore, within each SMP there is a memory hierarchy where certain cores will be able to

access local memory faster than non-local memory. This non-uniform memory access (NUMA) is a result of the nature of the shared memory and local cache architecture for each processing core. In this type of computing environment, it is often beneficial, for memory and performance reasons, to implement algorithms that take into account the non-uniformity of memory access and communication latency.

In the past, many advanced high-performance scientific applications have exclusively employed the Message Passing Interface (MPI) to achieve parallelism. However, the MPI standard treats communication amongst any given pair of processors uniformly and may not take full advantage of the hierarchical SMP architecture of many modern computers. In order to increase computational efficiency on very large scale problems, many researchers are employing a hybrid approach in which groups of SMP cores implement parallelism using a shared-memory model, while MPI is used to communicate between groups. The advantage of this approach is that local cores can access the same memory and avoid communication overhead. On the other hand, algorithms must now be developed or adapted to reflect this change. In this work, we implement a hybrid approach for large-scale structural analysis and optimization problems using POSIX threads, or Pthreads. The Pthreads library is a standard interface developed for UNIX systems for initiating and controlling threads within a shared memory environment.

The goal of this work is to develop parallel methods to solve large-scale structural optimization problems that make effective use of available high-performance parallel computing resources. We focus on large-scale structural optimization problems of the form:

$$\begin{aligned} \min \quad & f(\mathbf{x}, \mathbf{u}_1) \\ \text{w.r.t.} \quad & \mathbf{x} \\ \text{governed by} \quad & \mathbf{K}\mathbf{u}_i - \mathbf{f}_i = 0 \qquad i = 1, \ldots, N \\ \text{such that} \quad & \mathbf{c}_i(\mathbf{x}, \mathbf{u}_i) \geq 0 \qquad i = 1, \ldots, N \end{aligned} \tag{1}$$

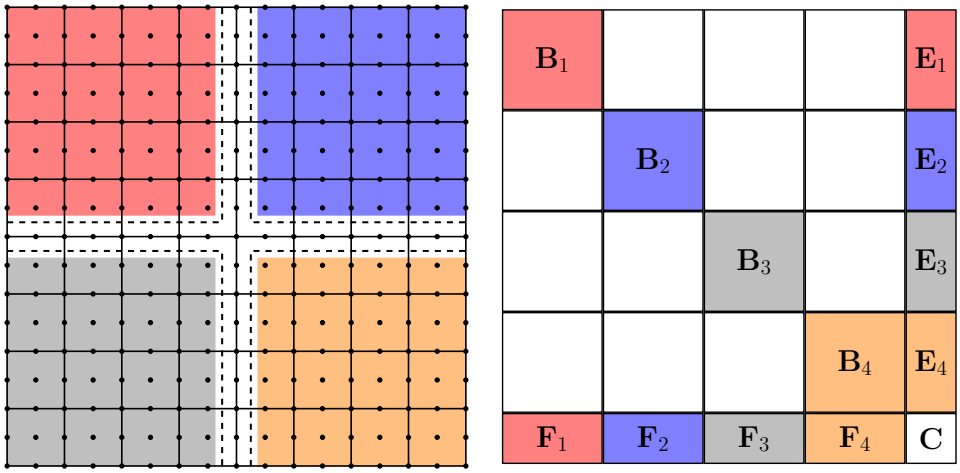
where $\mathbf{x}$ are the design variables, $\mathbf{f}(\mathbf{x}, \mathbf{u}_1)$ is the objective function, $\mathbf{u}_i$ are the state variables for each load case $i = 1, \ldots, N$, $\mathbf{K}$ is the stiffness matrix, $\mathbf{c}_i$ are the stress or failure constraints for each load case, and $\mathbf{f}_i$ is the load vector for each load case. We refer to Problem (1) as a large-scale structural optimization problem due to the size of the design space, the number of constraints, and the number of degrees of freedom in the governing equations. In particular, we focus on structural optimization problems with thousands of design variables and constraints and millions of structural state variables.

The remainder of the paper is outlined as follows: In Section 4 we outline the hybrid MPI/Pthreads scheme employed in this study including how the approach is adapted to direct factorization methods and derivative computation. In Section 5, we present results from two problems: a simple clamped square plate and a large-scale wing-box problem. Finally, in Section 6 we draw conclusions from the results presented herein.

4. Parallel performance

In this work, we enhance the parallelism of an existing finite-element code with a hybrid MPI/Pthreads scheme. For this study, we use the finite-element code called the Toolkit for the Analysis of Composite Structures (TACS), which uses domain decomposition to achieve parallelism for finite-element matrix and residual assembly, direct matrix factorization, and gradient-evaluation. While TACS is highly optimized for parallel performance, there are two important areas in which further improvements are possible: memory usage and matrix-factorization. As the number of domains in the domain decomposition increases, the memory requirements grow in an absolute sense such that the memory usage per domain decreases, but the total memory usage increases. Typically, the additional memory requirements do not pose a significant problem for machines with a large amount of memory per core. However, this trend may be a significant issue for more memory-limited computer architectures. The second potential improvement lies in the direct matrix factorization. As the number of domains increases, the number of arithmetic operations required to perform the direct factorization often increases as well. Furthermore, with increasing numbers of domains, the load balancing may deteriorate as the amount of time spent in each operation in the matrix factorization decreases. The goal of the hybrid scheme is to extract additional parallelism from a given domain decomposition, thereby enabling the use of fewer, larger domains leading to a reduction in memory requirements and a potential increase in the performance of time-critical tasks within the code.

For this study, we have implemented a funneled or master-only approach in which the parallel communication amongst MPI tasks is funneled through a master MPI thread which spawns all the threaded operations using the Pthreads API. We describe below in greater detail the implementation of the hybrid



(a) Domain decomposition (b) Matrix storage format

Figure 1: The domain decomposition and corresponding matrix for a domain decomposition of a third order finite-element mesh with four domains.

parallel algorithms. There are many tasks that are required for parallel finite-element analysis and design optimization. Within this paper, we focus on the tasks that consume the most computational time during analysis and design optimization: the direct factorization of the stiffness matrix, matrix and residual assembly, and the evaluation and assembly of partial derivative terms required for gradient evaluation. In the following sections, we outline the implementation of the domain decomposition-based finite-element code and describe how the threaded scheme is added to attain additional parallelism.

4.1 Direct matrix factorization

In this section, we describe the implementation of the direct matrix factorization scheme in TACS and outline how we have enhanced the scheme with a threaded implementation. In TACS, the direct matrix factorization uses a substructuring approach based on the Schur complement. In this approach, the unknowns in the finite-element problem are divided in to two separate groups: interface unknowns which lie directly on the domain boundaries, and internal unknowns which are in the domain interior. Figure 1a shows the domain decomposition where the coloured regions denote four separate domains, and the interface unknowns are illustrated in white. The internal unknowns for each processor are denoted as $\mathbf{x}_i$, for $i = 1, \ldots, n_d$, and the global vector of interface unknowns is denoted as $\mathbf{y}$. The contributions to the global system of equations from each domain are stored a local matrix partitioned as follows:

$$\mathbf{A}_i = \begin{bmatrix} \mathbf{B}_i & \mathbf{E}_i \\ \mathbf{F}_i & \mathbf{C}_i \end{bmatrix}, \qquad i = 1, \ldots, n_d, \tag{2}$$

where the $\mathbf{B}_i$ and $\mathbf{C}_i$ blocks represent the diagonal contributions to the internal and interface unknowns, respectively. The global system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, can be then be written as follows:

$$\begin{aligned} \sum_{i=1}^{n_d} (\mathbf{B}_i \mathbf{x}_i + \mathbf{E}_i \mathbf{y}) &= \mathbf{b}_i, \\ \sum_{i=1}^{n_d} (\mathbf{F}_i \mathbf{x}_i + \mathbf{C}_i \mathbf{y}) &= \mathbf{d}, \end{aligned} \tag{3}$$

where $\mathbf{b}_i$ and $\mathbf{d}$ represent the right hand sides for each local processor and the interface unknowns, respectively. The structure of the global system of equations is illustrated in Figure 1b.

The direct matrix factorization proceeds in two stages: first, the computation of the local contributions to the global Schur complement, followed by the assembly and factorization of the global Schur complement. The first stage consists of a series of computations that are independent and can be performed concurrently on all processors without communication, while the second stage consists of a matrix assembly and factorization that requires coordination amongst all processors.

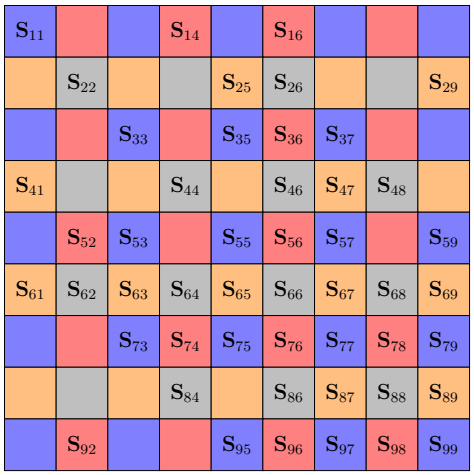


Figure 2: An illustration of the sparse 2D block cyclic matrix format on four processors.

In the first stage of the matrix factorization, the diagonal block corresponding to the internal unknowns is factored such that $\mathbf{B}_i = \mathbf{L}_i\mathbf{U}_i$. After this factorization is completed, the contribution to the global Schur complement are formed on for each domain within the mesh as follows:

$$\mathbf{S}_i = \mathbf{C}_i - \mathbf{F}_i\mathbf{U}_i^{-1}\mathbf{L}_i^{-1}\mathbf{E}_i, \tag{4}$$

where $\mathbf{S}_i$ is Schur complement contribution from the i^{th} domain. Once all processors have computed $\mathbf{S}_i$, the global Schur complement is assembled such that:

$$\mathbf{S} = \sum_{i=1}^{n_d} \mathbf{S}_i, \tag{5}$$

where $\mathbf{S}$ is the global Schur complement. Finally, the global Schur complement is factored on all processors such that $\mathbf{S} = \mathbf{L}_S\mathbf{U}_S$.

The local matrices for each domain in Eq. (2) are stored using a Block Compressed Sparse Row (BCSR) format in which the block matrices for the unknowns for each node are stored contiguously in memory [9]. In TACS we have implemented optimized block-specific code that is designed to increase the number of arithmetic operations per memory access. The global Schur complement (5) is stored in a sparse 2D block-cyclic matrix format in order to facilitate parallel factorization. In this format, the matrix is divided into a regular block structure and the blocks are assigned to processors in a cyclic pattern that is repeated until all blocks are assigned. The sparse 2D block cyclic format for a four processor case is illustrated in Figure 2, where only non-zero blocks are labeled.

In order to develop an efficient threaded implementation of the direct matrix factorization technique described above, we have focused on implementing threaded versions of two types of operations that are required during the factorization: matrix-matrix products and sparse lower and upper triangular back-solves. In both cases, the matrices involved in these operations are stored in the BCSR format as described above. In order to achieve good parallel performance, we have employed a task-assignment paradigm in which a queue of tasks is generated and assigned to each thread that completes its previous task. In order to guarantee correct execution order, the tasks are only added to the queue once all dependencies have been completed. In this manner, we ensure that the correct sequence of operations is performed without prescribing in advance the threads that will complete a given operation.

Algorithm 1 shows the pseudo-code used to compute a sparse matrix-matrix product: $\mathbf{C} = \mathbf{AB}$. Here, we use an event-driven task assignment routine, get_mult_sched_job(), that returns the starting row for the next part of the matrix-matrix multiplication. The variable row_group_size is selected to achieve a better memory access pattern such that fewer threads attempt to access the same locations in main memory at the same time.

Algorithm 2 shows the pseudo-code used to compute a sparse lower-triangular solve: $\mathbf{C} = \mathbf{L}^{-1}\mathbf{B}$. Again, we use an even-driven, task assignment paradigm, but due to the dependencies within the calculation, the job scheduler must resolve what task, if any, can be performed before other tasks finish.

Algorithm 1: Sparse matrix-matrix multiplication using an event-driven approach.

```
Compute C = AB;
init_mult_sched(); // Initialize the job scheduler
while completed_rows < nrows(C) do
    row = get_mult_sched_job(); // Retrieve the starting row for the next job
    for i = row to row + row_group_size do
        for k ∈ nz(A_i*) do
            for j ∈ nz(B_k*) do
                C_ij ← C_ij + A_ik B_kj;
    completed_rows ← completed_rows + row_group_size; // Update the task counter
```

Algorithm 2: Solution of a lower triangular system using an event-driven approach.

```
Compute C = L^-1 B;
Copy C ← B;
init_lower_sched();// Initialize the job scheduler
while completed_rows < nrows(C) do
    row, kstart, kend = apply_lower_sched_job(); // Retrieve the information for the job
    for i = row to row + row_group_size do
        for j ∈ nz(C_i*) do
            for k = kstart to kend do
                C_ij ← C_ij − L_ik C_kj;
    apply_lower_mark_completed(row, kstart, kend);// Mark the task as completed
```

This dependency graph complicates the algorithm considerably, however, specifying the execution path in advance typically results in slower execution times.

4.2 Matrix and derivative assembly operations

In this section, we briefly outline the methods we use to parallelize the computation and assembly of the finite-element stiffness matrix and terms required for gradient evaluation. In both cases, the contributions from individual elements are nearly independent and only a single mutual exclusion variable is required to ensure that the same memory location is not written to at the same time by two threads. In these operations, we assign element-based computations to a thread in a dynamic fashion to try to minimize idle time.

For the gradient computation, we use the adjoint method in which the total derivative of a function of interest, $\nabla_{\mathbf{x}} f$, is obtained by first solving the following adjoint system of equations:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{q}}^T \boldsymbol{\psi} = \frac{\partial f}{\partial \mathbf{q}}^T ,$$

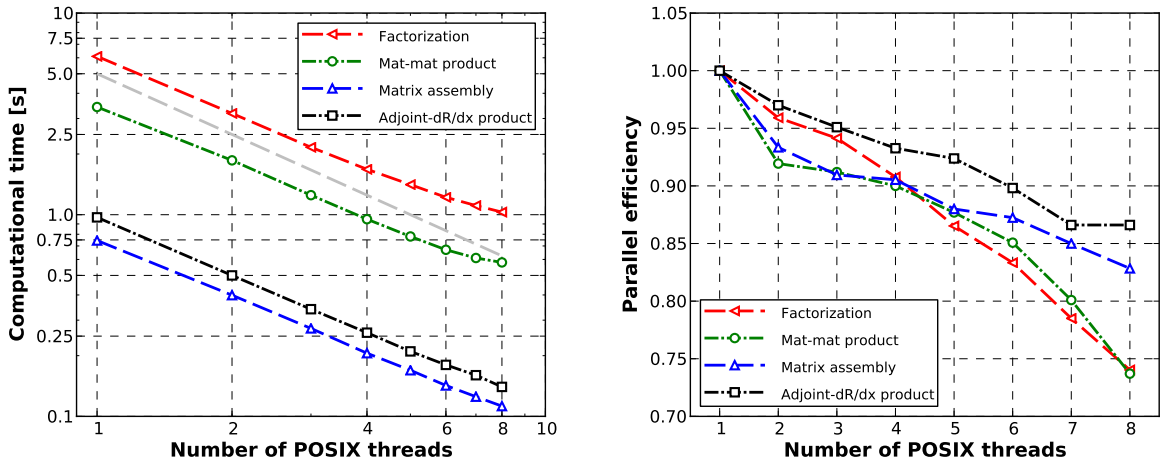
and then evaluating the total derivative:

$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial x} - \boldsymbol{\psi}^T \frac{\partial \mathbf{R}}{\partial \mathbf{x}}. \tag{6}$$

In the following discussion, we focus on the computational cost of the inner product of the adjoint vector with the derivative of the residuals with respect to the design variables: $\boldsymbol{\psi}^T \partial \mathbf{R}/\partial \mathbf{x}$. For this computation, we evaluate the contributions, element-by-element and add them to a local array stored by each thread. Again, we assign element-based evaluation tasks in a dynamic fashion. Once completed, the threads add the total contribution to the result on the master MPI thread. Finally, the master thread performs a global reduction across all MPI threads to compute the final result.

5. Results

In this section we present results from two finite-element models: a fully clamped square plate subject to pressure loading, and a transport aircraft wing-box model subject to maneuver loads. We first examine



(a) Parallel speedup

(b) Parallel efficiency

Figure 3: The parallel speed up and efficiency for the Pthreads-only results for the fully clamped plate subject to a pressure load.

the performance of TACS on the clamped plate to illustrate the scalability of the threaded implementation. Next, we demonstrate the hybrid MPI/Pthreads scheme for various numbers of MPI and threaded processes. Finally, we demonstrate the performance of the hybrid MPI/Pthread scheme on a large-scale design optimization problem with millions of degrees of freedom.

All cases presented here were run on the General Purpose Cluster (GPC) at SciNet [4]. Each node of the GPC is an Intel Xeon E5540 with a clock speed of 2.53GHz, with 16GB of dedicated RAM and 8 processor cores. In these comparisons, we only use nodes connected with non-blocking 4x-DDR InfiniBand.

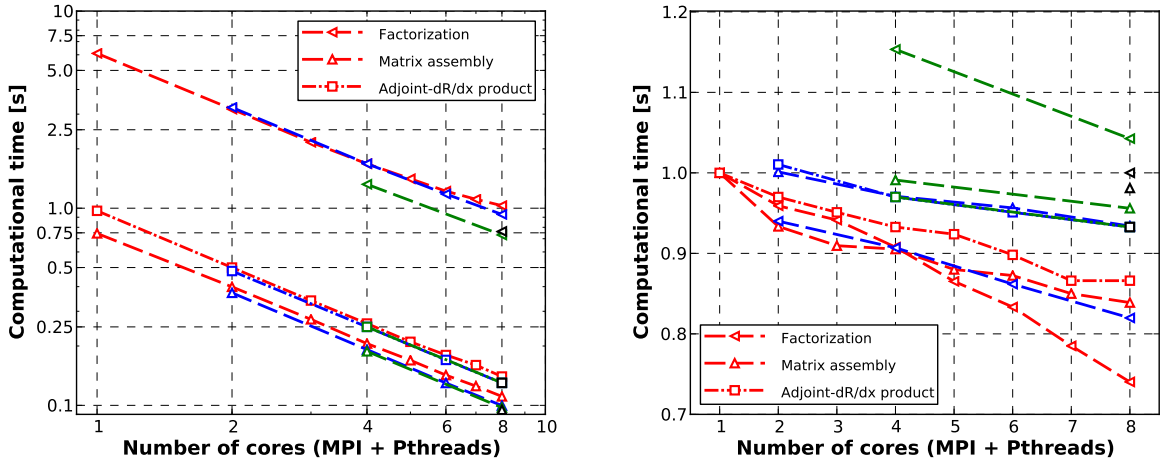
5.1 Fully clamped plate

In this section, we examine the parallel performance of the hybrid implementation for a relatively small structural problem on a single 8 core node. In this case we analyze a fully clamped, square plate subjected to a uniform surface pressure load. We discretize the problem using 64×64 third order MITC9 shell elements [2]. This results in a finite-element mesh with 16 641 nodes, 4 096 elements and just less than 100 000 degrees of freedom.

In order to isolate the performance of the threaded implementation, we first examine the performance of several finite-element assembly and factorization tasks with only the use of POSIX threads. Figure 3 shows the parallel speedup and efficiency of the factorization, matrix-matrix product, matrix assembly and adjoint-$\partial\mathbf{R}/\partial\mathbf{x}$ product. In order to account for the use of shared resources on the system, we average the computational times over several runs.

Overall the Pthreads-only implementation exhibits good parallel efficiency for cases with between 1 and 8 threads. All tasks exhibit excellent parallel efficiency for the cases with between 1 and 4 threads. However, the matrix-matrix products and matrix factorization do not maintain the same level of efficiency for the cases with between 5 and 8 threads. Nevertheless, the matrix assembly and adjoint-$\partial\mathbf{R}/\partial\mathbf{x}$ product tasks maintain good scalability all the way to the 8 thread case.

Next, we compare the performance for the hybrid MPI/Pthreads implementation. For these examples, we omit results from the matrix-matrix products which are only required for single-domain operations with the parallel matrix factorization. Here, we examine the computational performance of the matrix factorization, matrix assembly and adjoint-$\partial\mathbf{R}/\partial\mathbf{x}$ product. Figure 4 shows the parallel efficiency of the hybrid code for between 1 and 8 cores. Note that the red, blue, green, and black denote the 1, 2, 4, and 8 MPI-process cases, respectively. The hybrid MPI/Pthread implementation achieves excellent performance when using between 1 and 4 threads. For this problem the MPI-only cases achieve marginally better performance than the hybrid MPI/Pthread cases. Note that the higher-than-ideal performance of the four processor MPI case is due to a fortuitous ordering for the simple square clamped plate problem.



(a) Parallel speedup

(b) Parallel efficiency

Figure 4: The parallel speed up and efficiency for the fully clamped plate subject to a pressure load, with hybrid MPI/Pthreads. The red, blue, green and black denote the 1, 2, 4, and 8 MPI-process cases, respectively.

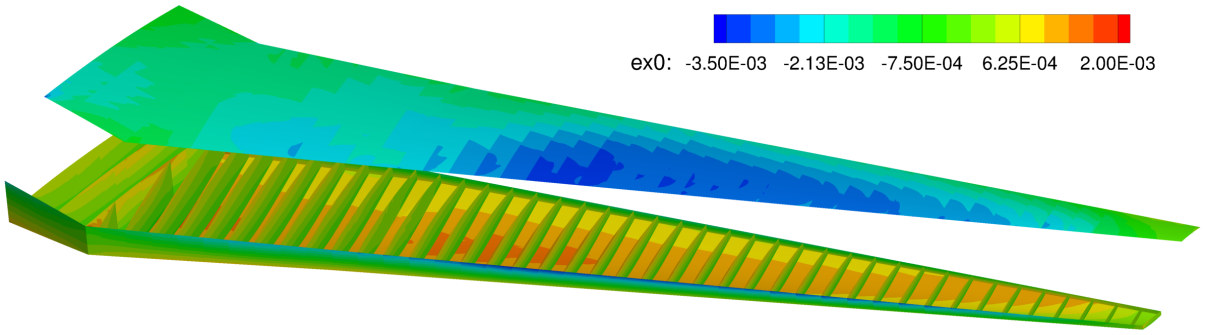


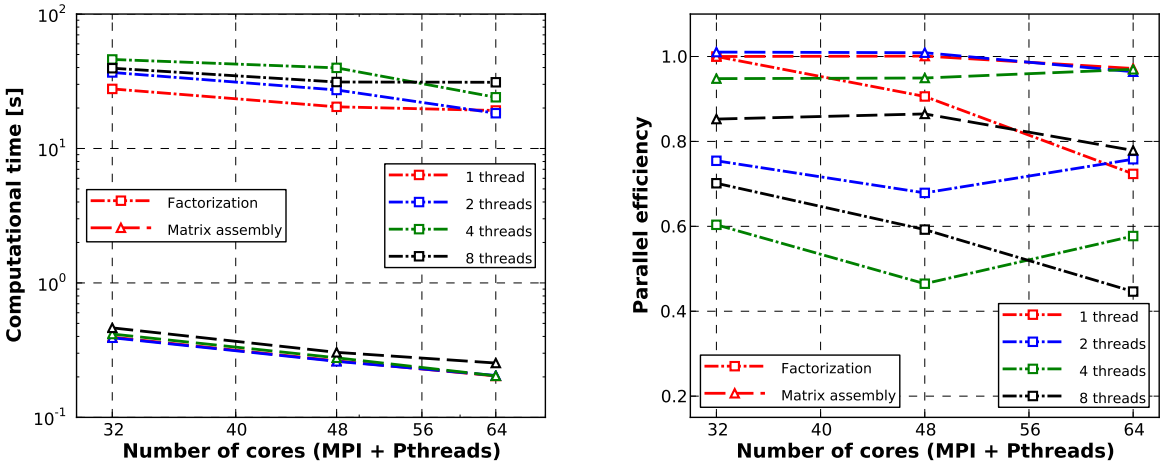
Figure 5: The composite wing box model used for the large-scale optimization study with 1.58 million degrees of freedom.

5.2 Wing box

In this section we examine the computational performance of the hybrid MPI/Pthreads implementation for a large-scale finite-element model with just over 1.58 million degrees of freedom. The finite-element model consists of 263 605 nodes and a total of 66 948 third order MITC9 shell elements. Figure 5 shows the strain distribution within the model when subjected to aerodynamic forces from a 2.5g maneuver condition. The wing box is 70 m long and consists of a two-spar configuration with 46 ribs.

In the following cases we ensure that the number of cores is equal to the number of threads per MPI process times the number of MPI processes. As a result, the SMPs are not oversubscribed and we do not rely on hyper-threading explicitly. For the cases presented here, we have found that oversubscription neither improves or harms the computational performance of the present implementation. Therefore for this study we limit the total number of threads to the available number of SMP cores.

Figure 6 shows the parallel speedup and efficiency for the wing case on 32, 48, and 64 cores for cases with 1, 2, 4, and 8 threads per MPI process. Note that the parallel efficiency is measured against the 32 core MPI-only case. The matrix computation and assembly scales well for all MPI/Pthread combinations. The 1, 2, and 4 thread cases attain better than 90% efficiency for all cases, while the 8 thread case drops below 80% efficiency for the 64 core case. The matrix factorization, however, does not scale as well. In particular, the cases with 4 and 8 threads only achieve between 50% and 70% efficiency. The matrix factorization for the MPI-only cases drops to 70% efficiency for the 64 core case, while the 2 thread hybrid code improves to 75% efficiency for the 64 core case.



(a) Parallel speedup

(b) Parallel efficiency

Figure 6: The parallel speed up and efficiency for the wing-box model with 1.58 million degrees of freedom. Results are shown for the 32, 48 and 64 core cases with 1, 2, 4, or 8 POSIX threads.

6. Conclusions

In this paper, we have presented a hybrid parallel MPI/Pthreads finite-element code implemented with large-scale gradient-based design optimization in mind. We have demonstrated that the hybrid MPI/Pthreads implementation achieves excellent parallel efficiency for assembly operations required for matrix, residual, and gradient evaluation. These are performance-critical tasks that must be repeated thousands of times within an optimization. While the implementation does not achieve ideal efficiency for some matrix factorization tasks, this is offset by improved algorithmic efficiency due to a reduction in the number of domains within the domain decomposition. Overall the implementation achieves excellent parallel efficiency and is well suited to large-scale structural and multidisciplinary design optimization applications.

References

[1] T. Borrvall and J. Petersson. Large-scale topology optimization in 3D using parallel computing. *Computer Methods in Applied Mechanics and Engineering*, 190(4647):6201 – 6229, 2001.

[2] M. L. Bucalem and K.-J. Bathe. Higher-order MITC general shell elements. *International Journal for Numerical Methods in Engineering*, 36:3729–3754, 1993.

[3] G. K. W. Kenway, G. J. Kennedy, and J. R. R. A. Martins. A scalable parallel approach for high-fidelity steady-state aeroelastic analysis and adjoint derivative computations. *AIAA Journal*, 2013. In press.

[4] C. Loken, D. Gruner, L. Groer, R. Peltier, N. Bunn, M. Craig, T. Henriques, J. Dempsey, C.-H. Yu, J. Chen, L. J. Dursi, J. Chong, S. Northrup, J. Pinto, N. Knecht, and R. V. Zon. SciNet: Lessons learned from building a power-efficient top-20 system and data centre. *Journal of Physics: Conference Series*, 256(1):012026, 2010.

[5] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. High-fidelity aerostructural design optimization of a supersonic business jet. *Journal of Aircraft*, 41(3):523–530, 2004.

[6] J. R. R. A. Martins, J. J. Alonso, and J. J. Reuther. A coupled–adjoint sensitivity analysis method for high–fidelity aero–structural design. *Optimization and Engineering*, 6:33–62, 2005.

[7] S. Padula and S. Stone. Parallel implementation of large-scale structural optimization. *Structural optimization*, 16:176–185, 1998.

[8] M. Papadrakakis, N. D. Lagaros, and Y. Fragakis. Parallel computational strategies for structural optimization. *International Journal for Numerical Methods in Engineering*, 58(9):1347–1380, 2003.

[9] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Pub. Co., 2nd edition, 2003.

[10] S. Venkataraman and R. Haftka. Structural optimization complexity: What has Moores law done for us? *Structural and Multidisciplinary Optimization*, 28:375–387, 2004.

[11] S. Wang, E. de Sturler, and G. H. Paulino. Large-scale topology optimization using preconditioned Krylov subspace methods with recycling. *International Journal for Numerical Methods in Engineering*, 69(12):2441–2468, 2007.