Using a Filter-Based SQP Algorithm in a Parallel Environment

Gerhard Venter (gventer@sun.ac.za)*

University of Stellenbosch, Stellenbosch, South Africa

Garret N. Vanderplaats (vanderplaats@vrand.com)†

Vanderplaats Research and Development, Inc., Monterey, CA 93940

A parallel, filter-based, sequential quadratic programming (SQP) algorithm is implemented and tested for typical general-purpose engineering applications. Constrained engineering test problems, including a finite element simulation, with up to 512 design variables are considered. The accuracy and serial performance of the filter-based algorithm are compared against that of a standard SQP algorithm. The parallel performance of the algorithm is evaluated, using up to 128 cores on a Linux Cluster. The results indicate that the filter-based algorithm competes favorably with a standard SQP algorithm in a serial environment. However, the filter-based algorithm exhibits much better parallel efficiency due to the lack of a one dimensional search. A parallel efficiency of 74% is obtained when using 128 cores.

I. Introduction

General-purpose optimization is a powerful engineering tool that allows designers to couple an optimization algorithm with an analysis tool. Several commercially available tools exist that aid the designer in this process. These tools typically provide a number of robust optimization algorithms and a process integration tool that simplifies the coupling process. Examples include VisualDOC from Vanderplaats Research and Development, iSIGHT from Dassault Systèmes, modeFRONTIER from Esteco and Optimus from Noesis Solutions. Despite the availability of these tools, and the power that the underlying technology brings to the design process, general-purpose optimization is not being accepted in industry at the rate one would expect. Some of the reasons traditionally provided for this lack of acceptance are that optimization is difficult to use, the coupling of the optimizer to the analysis code is painful and that an optimization study can take a long time to complete. The mentioned tools adequately address the first two issues. However, the third issue is more problematic and is the focus of the current paper.

By its very nature, a single optimization run requires multiple analyses to complete. A typical optimization study can easily require one or two orders of magnitude more time than a single analysis. An obvious way to deal with this problem is to exploit the advantages offered by parallel computing. Most of the commercially available general-purpose optimization tools provide some parallel capabilities, but these are often difficult to setup and use, and more importantly do not fully exploit the advantages available from parallel processing. Although parallel processors are becoming more easily available to designers, think for example of multicore processors that are becoming common place in personal computers, there are few effective parallel optimization algorithms available.

When implementing a parallel optimization algorithm, one of two approaches can be followed. The first is to devise a parallel implementation of an existing algorithm, the second is to develop a brand new

---

*Professor, Department of Mechanical and Mechatronic Engineering, Stellenbosch, South Africa, AIAA Senior Member
†Chief Executive Officer, Vanderplaats Research and Development, Inc., 126 Bonifacio Place, Suite F, Monterey, CA 93940, AIAA Fellow
algorithm, specifically designed for a parallel environment. The advantage of using the first approach is that one can leverage existing technology and build on algorithms that are known to be robust, efficient and applicable to a wide range of problems. The problem with extending existing algorithms that were explicitly developed for a serial environment to a parallel environment, is that the most efficient serial algorithm does not necessarily provide the most efficient parallel algorithm (e.g., Venter and Watson\(^1\)). The advantage of the second approach is that one can potentially gain more efficiency from the parallel environment. However, developing a robust optimization algorithm that is applicable to a wide range of problems is not a trivial task.

Many researchers that look at parallelizing existing algorithms concentrate on zero-order algorithms like Genetic Algorithms (GAs) or Particle Swarm Optimization (PSO), with varying success. For example, one can obtain near theoretical speedup by implementing an asynchronous PSO algorithm (e.g., Venter and Sobieszczanski-Sobieski\(^2\) and Koh et al.\(^3\)). Even so, these zero-order algorithms are computationally inefficient, need parameter tuning that is problem dependent and are typically limited to problems with smaller numbers of design variables. Even within a parallel environment, these algorithms rarely compete with the efficiency of gradient-based algorithms. As a result, in industry, most engineering optimization studies are still performed using gradient-based optimization.

Parallel processing is becoming more readily available to designers, especially in the form of a small number of processing units. Within this environment, the authors believe that fully exploiting the benefits provided by an efficient and robust gradient-based algorithm is an important enabling technology for the use of general-purpose optimization in industry. The current paper is guided by the basic observations from Venter and Watson,\(^1\) which implemented parallel versions of three widely used gradient-based algorithms namely, sequential quadratic programming (SQP), the modified method of feasible directions (MMFD) and sequential linear programming (SLP). The commercially available Design Optimization Tools\(^4\) (DOT) implementation of these algorithms was used for the study. For the example problem considered, the results indicated that in a serial environment the MMFD algorithm was the most efficient and the SLP algorithm was the least efficient. However, in a parallel environment the SLP algorithm required the smallest elapsed time to complete. Even though the SLP algorithm required more function evaluations to complete, it could more efficiently exploit the parallel environment, since it required no one-dimensional search. Although the SLP algorithm performed best for the example problem considered, it is in general not considered a good algorithm in the class of the SQP and MMFD algorithms and in many cases may not converge to the optimum solution.

The current paper will concentrate on a parallel implementation of a robust, efficient and widely used gradient-based algorithm, namely the sequential quadratic programming (SQP) algorithm (e.g., Wilson,\(^5\) Han\(^6\) and Powell\(^7\)). The application region is general-purpose optimization in engineering with problems that have anywhere from one or two, to several hundred design variables. Most general-purpose optimization problems in engineering fall in this category, since the number of design variables is limited by the fact that gradient information is typically not available and must be calculated using finite difference gradient calculations. The current paper will build on the lessons learned from Venter and Watson\(^1\) to investigate how an efficient SQP algorithm can be implemented in parallel.

### II. Gradient-based Optimization Background

When parallelizing an algorithm, one typically starts by identifying the main sources of computational cost. For most gradient-based optimization algorithms, one can naturally divide the computational cost into three components. First is the computational cost associated with obtaining the required gradient information, typically in the form of finite difference gradient calculations. Second is the cost of performing the one-dimensional search, typically in the form of a polynomial approximation or a Golden Section search. Third is the cost associated with the optimization algorithm itself, for example solving the direction finding sub-problem. If the analysis code requires any significant time to complete, even as little as just a few seconds, the time required to compute the finite difference gradient calculations typically dominates the
total time. This is followed by the time required to complete the one-dimensional search calculations, while the time associated with the algorithm itself is typically negligible. In most cases, engineering simulations take enough time that one can ignore the time required to perform the computations associated with the algorithm. When this is not the case, for example when the analysis takes only a fraction of a second to complete, the overall time required to complete the optimization study is typically small and parallelization is most probably not important to start with. The present work will concentrate on cases where the analysis time takes anywhere from a few seconds to a few minutes or longer per analysis.

Since the finite difference gradient calculations tend to dominate the overall time for most engineering problems and are easily parallelized, performing these computations in parallel has been the focus of many previous efforts to parallelize existing gradient-based algorithms (e.g., Venter and Watson\(^1\)). The one-dimensional search computations, although having a significant contribution to the overall computational burden, is inherently a serial process that is difficult to parallelize. As a result many parallel implementations of existing algorithms still perform these computations in a serial fashion. The present paper will look at ways to overcome this problem for the SQP algorithm, by selecting a variant of the traditional SQP algorithm that does not depend on a one-dimensional search.

The general non-linear optimization problem that will be considered in the present paper, can be summarized as follows:

\[
\begin{align*}
\text{Minimize:} & \quad f(x) \\
\text{Such That:} & \quad g_j(x) \leq 0 \quad j = 1, m \\
& \quad x^l_i \leq x_i \leq x^u_i \quad i = 1, n
\end{align*}
\]

where \(f\) is the objective function, \(g_j\) are inequality constraints and \(x\) is a vector of \(n\) design variables. \(x^u_i\) and \(x^l_i\) represent the upper and lower bounds on the design variables, referred to as side constraints.

### III. The Traditional SQP Algorithm

Sequential Quadratic Programming (SQP) is a popular general-purpose optimization algorithm that is widely used to solve problems of the type summarized in Eq. (1). The SQP algorithm creates an approximate quadratic programming (QP) sub-problem that is used to find a search direction. The QP subproblem is obtained by creating a quadratic approximation of the objective function and linear approximations of the constraints as shown in Eq. (2) (e.g., Powell\(^7\))

\[
\begin{align*}
\text{Minimize:} & \quad Q(x) = F(x) + \nabla F(x)^T s + \frac{1}{2} s^T B s \\
\text{Such That:} & \quad g_j(x) + \nabla g_j(x)^T s \leq 0 \quad j = 1, m
\end{align*}
\]

where \(Q\) is the quadratic objective function, \(B\) is a positive definite matrix which is initially the identity matrix and \(s\) is the unknown search direction. The \(B\) matrix is updated after each iteration to approximate the Hessian of the Lagrangian, using the the standard BFGS update scheme (e.g., Powell\(^7\)). The design variables are the unknown components of the \(n\)th dimensional search direction, \(s\). The QP subproblem can be solved with a specialized QP algorithm, but it is a well posed optimization problem that can also be solved with any other non-linear optimization algorithm like MMFD.

It is well known that if the SQP algorithm is started far from a solution, it may not converge to a local optimum if the search direction obtained from Eq. (2) is used directly. To ensure convergence to a local optimum, a one-dimensional search is typically performed to obtain a step size \(\alpha\). A new design point is then obtained from the search direction and the step size, as follows:

\[
x^i = x^{i-1} + \alpha s^i
\]

A new design point is only accepted if a specified merit function is reduced during the one-dimensional search. Most often, the merit function is specified as an exterior penalty function that combines the objective function and a measure of the constraint violations. A reduction in the merit function thus implies...
a reduction in the objective function and/or constraint violation. A popular merit function, based on the
Lagrangian function, (e.g., Powell\textsuperscript{7} as outlined in Vanderplaats\textsuperscript{8}) is shown in Eq. (4), where an initial value
of $\alpha = 1$ is typically a good starting value.

\[
\text{Minimize: } \phi(x) = F(x) + \sum_{j=1}^{m} u_j \{\max[0, g_j(x)]\}
\]

where: $x = x^{i-1} + \alpha s$

\[ u_j = |\lambda_j| \quad \text{First iteration} \]
\[ u_j = \max[|\lambda_j|, \frac{1}{2}(u'_j + |\lambda_j|)] \quad \text{Subsequent iterations} \]

In Eq. (4) $u'_j = u_j$ from the previous iteration and $\lambda_j$ refers to the Lagrange multipliers obtained from the QP
problem summarized in Eq. (2).

With the optimum step size $\alpha^*$ obtained from solving Eq. 4, one can update the current design point using
Eq. (3), update the $B$ matrix using the BFGS scheme (e.g., Powell\textsuperscript{7}) and repeat the process until convergence.

A. Parallel Implications

In general the SQP algorithm is thus a two step process, where one first determines the search direction
(which depends on $1^{st}$ order gradient information) and second determines the step size, the one dimensional
search. This two step process is problematic from a parallelization point of view, and is discussed in more
detail next.

In general, the time required to perform an optimization study, using a two step gradient-based algorithm
like the SQP algorithm, can be summarized as

\[ T_{\text{total}} = n\text{Iter} \left( T_{\text{grad}} + T_{1D} \right) + T_{\text{int}} \]  (5)

where $T_{\text{total}}$ refers to the total time required to complete the optimization study, $n\text{Iter}$ the number of design
iterations required to achieve convergence, $T_{\text{grad}}$ the time required to perform a single set of gradient calcu-
lations, $T_{1D}$ the time required to perform a single one-dimensional search and $T_{\text{int}}$ the time required by the
optimization algorithm itself.

To investigate the parallel speedup that can be achieved from this general setup, consider the following
assumptions: (1) the $T_{\text{int}}$ component is small and can be neglected; (2) each analysis takes the same time
denoted by $T_{\text{anal}}$; (3) the total time is normalized to obtain $\overline{T}_{\text{total}} = T_{\text{total}} / T_{\text{anal}}$; (4) enough processors
are available and perfect speedup is obtained to give $T_{\text{grad}} / T_{\text{anal}} = 1$; and (5) the one-dimensional search
is performed in series. Experience with the commercially available SQP algorithm provided as part of the
Design Optimization Tools (DOT) library,\textsuperscript{4} shows that the one-dimensional search (using a polynomial a-
proximation) on average takes about three analyses, yielding $T_{1D} / T_{\text{anal}} = 3$. The total time to complete the
optimization study in our assumed, best case parallel environment then becomes:

\[ \overline{T}_{\text{total}} = n\text{Iter} \left( 1 + 3 \right) \] \hspace{1cm} (6)

Equation (6) represents the best case scenario when parallelizing only the finite difference gradient calcu-
lations. Assuming that the one-dimensional search is also parallelized, the best case scenario would be to
perform the one-dimensional search in the time of a single analysis to obtain:

\[ \overline{T}_{\text{total}} = n\text{Iter} \left( 1 + 1 \right) \]  \hspace{1cm} (7)

This is half the time of only parallelizing the finite difference calculations, however if the gradient calculations
and the one-dimensional search can be combined into a single operation, the overall time can be further
reduced to:

\[ \overline{T}_{\text{total}} = n\text{Iter} \]  \hspace{1cm} (8)
Equation (8) can be achieved in one of two ways: (1) by calculating the gradient information at each point used in the parallel one-dimensional search or (2) using an algorithm that has no one-dimensional search. Of the two approaches, approach (2) is preferred, since approach (1) would typically require a large number of processors. The next section will look at a variant of the traditional SQP algorithm that provides similar performance, without the use of a one-dimensional search.

IV. Filter based SQP

Fletcher and Leyffer\textsuperscript{9} introduced a variant to the traditional SQP algorithm that does not use a penalty function to guarantee convergence to a local optimum. Their goal was to introduce an algorithm that is easy to implement, robust and that converges rapidly to a local optimum. From a parallel perspective, the attractive feature of their algorithm is that no penalty function, and thus no one-dimensional search, is required. It is thus possible to use a well-established, robust, algorithm efficiently in a parallel environment.

The algorithm by Fletcher and Leyffer alleviates some implementation issues associated with the traditional SQP algorithm. First, it is no longer necessary to deal with the problem dependent penalty function and associated penalty parameters. Second, when a penalty function is used, the super-linear convergence associated with the SQP algorithm can be destroyed. This is known as the Maratos effect.\textsuperscript{10} For super-linear convergence, $\alpha^*$ needs to be close to unity in the final iterations before convergence. However, this is often prevented by the penalty function that returns an $\alpha^*$ value much less than one. The new algorithm also does not suffer from the Maratos effect.

Fletcher and Leyffer\textsuperscript{9} proposed the use of a bi-objective formulation to eliminate the need of a penalty function. Their approach is based on the observation that there are two competing aims in non-linear programming. The first is to minimize the objective function $f$ and the second is to minimize the constraint violation. The traditional SQP algorithm makes use of a penalty function approach to combine these two objective functions into a single objective minimization problem. Fletcher and Leyffer replaces the penalty function with a bi-objective formulation as follows

Minimize: $f(x)$
Minimize: $h(g(x))$ \hspace{1cm} (9)

where the constraint satisfaction $h(g(x))$ is expressed as:

$$h(g(x)) = \sum_{j=1}^{m} \max(0, g_j(x)) \hspace{1cm} (10)$$

Instead of solving Eq. 9 using a formal multi-objective approach, a filter is defined based on the concept of dominance. A solution $(f^k, h^k)$ dominates another solution $(f^l, h^l)$ if and only if both $f^k \leq f^l$ and $h^k \leq h^l$. The filter consists of a set of non-dominated solutions and is used as the criterion (instead of a penalty function) for accepting or rejecting a new solution from the SQP method. A new solution is acceptable by the filter if it is not dominated by any other solution already in the filter. When a solution is accepted by the filter, it is also used to update the filter, by including it in the list of non-dominated solutions. After a new solution is included in the filter, the filter is maintained to ensure that the new solution does not dominate any solutions already in the filter. Any existing solutions that are dominated by the new solution are removed from the filter. In addition to the filter a trust-region, or move limit strategy, is implemented to guarantee convergence to a local optimum. The trust-region simply limits the magnitude of the search direction and is implemented by adding a constraint to Eq. (2) as shown in Eq. (11).

Minimize: $Q(s) = F(x) + \nabla F(x)^T s + \frac{1}{2} s^T B s$
Such That: $g_j(x) + \nabla g_j(x)^T s \leq 0 \quad j = 1, m$
$\|s\|_{\infty} \leq \rho \hspace{1cm} (11)$

American Institute of Aeronautics and Astronautics
When a new solution is rejected by the filter, the trust region radius \( \rho \) is reduced and the SQP step is repeated. As with any trust region approach, reducing the trust region radius can lead to a situation where no feasible solution exists within the trust region. This will lead to an infeasible QP solution and calls for a restoration phase. Fletcher and Leyffer suggest a simple restoration phase that consists of minimizing \( h(g(x)) \) until a feasible solution is found. In the present work the design variables are normalized and an initial trust-region radius of 0.5 is used. If the new point is accepted by the filter and the maximum search direction component is equal to the trust-region radius, the radius is increased by a factor of two. If the new point is rejected by the filter, the radius is reduced by a factor of four. The filter-based SQP algorithm can then be outlined as shown in Algorithm 1 below.

**Algorithm 1** Filter based SQP algorithm

Given \( x^0, \rho \) and \( k = 0 \)

1: repeat
2: Solve the QP (Eq. 11) to obtain \( s^k \)
3: if QP is infeasible then
4: Find a new point \( x^{k+1} \) in the restoration phase
5: else
6: Set \( x^{k+1} \leftarrow x^k + s^k \)
7: if \( (f^{k+1}, h^{k+1}) \) is accepted by the filter then
8: Accept \( x^{k+1} \)
9: Add \( (f^{k+1}, h^{k+1}) \) to the filter
10: Possibly increase the trust region radius \( \rho \)
11: else
12: Reject step and set \( x^{k+1} \leftarrow x^k \)
13: Reduce the trust region radius \( \rho \)
14: end if
15: end if
16: set \( k \leftarrow k + 1 \)
17: until convergence

In the present work only the basic algorithm, as outlined above, is implemented. Fletcher and Leyffer proposed several heuristics to improve both the robustness and the efficiency of the basic algorithm. The results from the current paper can thus be considered as a worst case scenario, with a production quality implementation expected to yield a more robust (not an issue for the examples considered here) and a more efficient algorithm. The heuristics will not influence the parallel implementation of the algorithm and a more efficient variant will simply result in fewer iterations and thus shorter elapsed time.

V. Parallel Implementation and Testbed

The advantage of using the filter-based SQP in a parallel environment is that the penalty approach is not used and as a result the one-dimensional search is eliminated. In a parallel environment, the filter-based algorithm thus effectively reduces the traditional SQP algorithm from a two step to a single step process where all the analyses within a design iteration can be performed all at once. If the traditional and the filter-based algorithms both require the same number of iterations for convergence, the filter-based algorithm will be twice as fast in an ideal parallel environment, as outlined in Section III. Alternatively, as long as the filter-based algorithm require less than twice the number of iterations as compared to the traditional SQP, the filter-based approach would still be more efficient in a parallel environment.

The parallel implementation of the filter-based algorithm is fairly straight-forward. The only function evaluations associated with each design iteration is the finite difference gradient calculations and these can easily be performed in parallel. The parallel implementation used here, makes use of the OpenMPI\(^1\)

---

1 American Institute of Aeronautics and Astronautics
implementation of the Message Passing Interface (MPI) standard. OpenMPI was used to create a master-worker parallel algorithm. In this implementation, the master processor decides what work should be done and then assigns a task to each worker processor. As soon as a worker processor is done with its task it reports back to the master processor and checks if there are more tasks available. This master-worker configuration thus provides dynamic load balancing within the parallel environment.

It is important to note that a synchronous parallel implementation is used, where all analyses within a given design iteration is completed before analyses from the next iteration is started. In some cases, a synchronous implementation can lead to situations where all the worker processors are waiting for a single analysis to complete, thus resulting in poor parallel speedup. This can easily happen when: (1) the time required to complete a single analysis depends on the design point being analyzed; (2) a heterogeneous system of processing units is used; and (3) the number of analyses is not an integer multiple of the number of processors. Unfortunately, gradient based optimization and in particular the SQP algorithm does not provide for an asynchronous implementation, where analyses from the next design iteration can be analyzed in the current design iteration. The fact that a synchronous implementation is used, means that a drop-off in parallel efficiency is expected as the number of processors is increased, since more processors will be idle at the end of each design iteration.

The parallel code was tested on the high performance computing facility at Stellenbosch University. This facility consists of a Linux cluster that has a total of 168 2.83 Ghz Xeon cores (21 compute nodes, each with 2 quad core processors), 2 GByte memory per core and 300 GByte local disk storage for each compute node.

VI. Numerical Results

Engineering example problems were selected to illustrate the parallel speedup that can be achieved from the proposed implementation. The first step was to compare the performance of the filter-based SQP algorithm with that of a traditional algorithm in a serial environment. A small example problem is used for this purpose. The goal here was to show that the filter based algorithm provides roughly the same performance and accuracy as a traditional SQP algorithm. The traditional SQP implementation that was selected for the comparison, is the commercially available algorithm provided as part of the Design Optimization Tools (DOT) library. The second step was to evaluate the parallel performance of the filter-based algorithm in a parallel environment, using a larger example problem. The goal here was to verify the performance of the filter based algorithm in a parallel environment.

A. Example problem 1: Cantilevered Beam

Before considering the parallel performance of the new algorithm, it was necessary to first investigate the performance of the filter-based algorithm compared to that of the traditional algorithm. This comparison was conducted in a serial environment by solving the example problem for different number of design variables. The cantilevered beam problem from Vanderplaats was considered as an example problem, as shown in Fig. 1. In the current paper, a beam with a fixed length of 0.5 m, an applied load of 50 kN and a material with Young’s Modulus of 200 GPa was used. The beam is divided into segments of equal length, each with width $b_i$ and height $h_i$.

The optimization problem is defined as changing the height and width of each segment to minimize the volume, while satisfying displacement, stress and geometric constraints as shown in Eq. (12).
Minimize: \( \text{Volume} = \sum_{i=1}^{\text{nSeg}} b_i h_i l_i \)

Such That: \( \frac{\sigma_i}{\sigma} - 1 \leq 0 \quad i = 1, \text{nSeg} \)
\( h_i - 20 b_i \leq 0 \quad i = 1, \text{nSeg} \)
\( \frac{y_n}{y} - 1 \leq 0 \)
\( b_i \geq 1.0 \quad h_i \geq 5.0 \)

(12)

The stress and geometric constraints are applied to each segment, while only the tip displacement is constrained. The maximum stress \( \sigma \) is equal to 140 MPa, while the maximum displacement \( y \) is equal to 25 mm. In all test cases the optimization is started from \( b_i = 5 \) and \( h_i = 40 \). The problem is implemented in such a way that the number of segments (nSeg) can easily be changed to control the problem size.

The number of design variables was varied from 2 to 512 to cover the application range mentioned earlier in the paper. The objective function value and number of iterations required for convergence is summarized in Table 1.

<table>
<thead>
<tr>
<th>Design Variables</th>
<th>Objective DOT</th>
<th>Filter-Based DOT</th>
<th>Iterations DOT</th>
<th>Filter-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>89533</td>
<td>89431</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>72986</td>
<td>72868</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>66177</td>
<td>66211</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>64513</td>
<td>64439</td>
<td>12</td>
<td>23</td>
</tr>
<tr>
<td>32</td>
<td>63957</td>
<td>63857</td>
<td>15</td>
<td>24</td>
</tr>
<tr>
<td>64</td>
<td>63751</td>
<td>63654</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>128</td>
<td>64244</td>
<td>64263</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>256</td>
<td>63930</td>
<td>63660</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>512</td>
<td>64658</td>
<td>64766</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 1 illustrates that the two algorithms are able to accurately find the optimum with fairly similar
performance. For the 9 test cases considered, the maximum difference in objective function value was only 0.42%. The performance was more difficult to compare. The number of iterations fluctuates with problem size and between the two algorithms. On average, DOT required 11.11 iterations for the nine test cases, while the filter-based algorithm required 14.22. Additionally, the filter-based algorithm never required more than two times the number of iterations as compared to DOT. In all cases, the filter based algorithm thus would outperform the traditional SQP algorithm in a parallel environment.

B. Example problem 2: Finite Element Simulation

Section A provided a validation of both the accuracy and efficiency of the filter-based algorithm, by comparing it against a commercially available algorithm in a serial environment. The second example concentrates on the performance of the filter-based algorithm in a parallel environment. The example considers a derivation of the first example, where a cantilevered pipe with a hollow circular cross section is considered as shown in Fig. 2.

![Pipe example](image)

A steel pipe with $E = 206$ GPa, $\nu = 0.28$ and $\rho = 7620$ kg/m$^3$ is considered. It is assumed that the pipe is completely fixed at the wall (all degrees of freedom are restrained), while a vertical tip load is applied at the free end. The load is applied at the center point of the cross section, at a node that is connected with rigid elements to the circumference of the pipe. A finite element model that consists of 7050 linear shell elements and has 42306 degrees of freedom is used to analyze the structure. Each finite element analysis calculates the displacement and Von Mises stress values due to the tip load, as well as the natural frequencies of the beam. To increase the computational time of a single analysis (in an attempt to simulate a simple real world finite element analysis), the first 200 frequencies are calculated.

For the optimization problem, the pipe is split into 150 property groups. Each property consists of a closed ring of elements, exactly one element wide along the length of the beam, as shown in Fig. 2. The objective is to minimize the mass of the pipe, by changing the thickness value of each property. There is thus a total of 150 independent design variables, each with an initial value of 25 mm and each allowed to change between 1 mm and 100 mm. In terms of constraints, a maximum tip displacement of 10 mm and a maximum Von Mises stress of 150 MPa is enforced, while the first natural frequency must be larger than 35 Hz. The location of the maximum stress within each property set does not change during the optimization. As a result, only the maximum stress for each property set is used as a constraint. In total there is thus 1 displacement, 1 frequency and 150 stress constraints.

The algorithm converged in 21 iterations, where each iteration requires a set of finite difference gradient analyses. In total, 3171 analyses were required to obtain the optimum solution. To evaluate the parallel performance of the algorithm, the same problem was repeated using different numbers of processors as shown in Fig. 3 and summarized in Table 2. The speedup is obtained by dividing the total time required to complete the optimization using a single processor by the total time required to complete the optimization
using \( n \) processors. The theoretical speedup is a result of the synchronous implementation associated with the SQP algorithm. The theoretical speedup for the synchronous implementation is controlled by the number of processors and is obtained from:

\[
\text{Speedup}_{\text{Theoretical}} = \frac{n\text{Tasks}}{\text{ceil}\left(\frac{n\text{Tasks}}{n\text{Proc}}\right)} \tag{13}
\]

where \( n\text{Tasks} \) is the number of analyses, \( n\text{Proc} \) is the number of processors and \( \text{ceil} \) is the ceiling operator. For example, the current example problem requires 151 analyses for each iteration. Since all analyses must be completed before moving to the next iteration, it makes no difference whether say 80 or 140 processors are used. In both cases there are more processors than half the number of analyses and as a result it would take the equivalent time of two analyses to complete the iteration. This behavior is clearly illustrated in both Fig. 3 and Table 2. Finally, the efficiency is a measure of how good the actual speedup is relative to the theoretical speedup and is obtained from:

\[
\text{Efficiency} = 100 \times \frac{\text{Speedup}_{\text{Actual}}}{\text{Speedup}_{\text{Theoretical}}} \tag{14}
\]

Table 2. Pipe speedup

<table>
<thead>
<tr>
<th>Num. Proc.</th>
<th>Time (s)</th>
<th>Speedup</th>
<th>Theoretical Speedup</th>
<th>Efficiency (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>159064</td>
<td>1.00</td>
<td>1.00</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>52649</td>
<td>3.02</td>
<td>3.97</td>
<td>76</td>
</tr>
<tr>
<td>8</td>
<td>23524</td>
<td>6.76</td>
<td>7.95</td>
<td>85</td>
</tr>
<tr>
<td>16</td>
<td>11274</td>
<td>14.11</td>
<td>15.10</td>
<td>93</td>
</tr>
<tr>
<td>32</td>
<td>6012</td>
<td>26.46</td>
<td>30.20</td>
<td>88</td>
</tr>
<tr>
<td>48</td>
<td>4611</td>
<td>34.50</td>
<td>37.75</td>
<td>91</td>
</tr>
<tr>
<td>64</td>
<td>3804</td>
<td>41.82</td>
<td>50.33</td>
<td>83</td>
</tr>
<tr>
<td>80</td>
<td>3038</td>
<td>52.36</td>
<td>75.50</td>
<td>69</td>
</tr>
<tr>
<td>96</td>
<td>3153</td>
<td>50.45</td>
<td>75.50</td>
<td>67</td>
</tr>
<tr>
<td>112</td>
<td>2983</td>
<td>53.33</td>
<td>75.50</td>
<td>71</td>
</tr>
<tr>
<td>128</td>
<td>2836</td>
<td>56.09</td>
<td>75.50</td>
<td>74</td>
</tr>
</tbody>
</table>

Note that in Fig. 3 and Table 2 no data is provided for the case where 2 processors is used. This is because the master processor does not do any work, it is only used to coordinate communication between the worker
processors. Running the code in series on one processor or running it in parallel on two processors thus results in similar timing information. The fact that the master processor does not do any work, also explains the relatively low efficiency for four processors since 25% of the processors are not doing any work. When considering the 16 processors case, only 6.25% of the processors are not working.

Figure 3 and Table 2 indicates that we get very good parallel performance of above 85% for up to 64 processors. There is then a drop-off in the efficiency which seems to stabilize around 70-75% for up to 128 processors. Overall, there is good parallel efficiency up to a relatively large number of processors. The parallel efficiency is shown graphically in Fig. 4 below.

![Figure 4. Parallel Efficiency](image)

### VII. Concluding remarks

This paper investigates the use of a filter-based SQP algorithm in a parallel environment. The accuracy and parallel efficiency of the filter-based algorithm is evaluated using two engineering example problems. The first problem is used to establish the accuracy and performance of the algorithm relative to a commercially available algorithm in a serial environment. The second to evaluate the parallel performance of the algorithm in a parallel environment.

For the first example problem, the problem size was varied between 2 and 512 variables. The results show that the filter-based SQP algorithm compares very well with a traditional SQP algorithm, both in terms of accuracy and efficiency.

The second example indicates that the algorithm has potential for implementation in a parallel environment with large numbers of processors. In this case, up to 128 processors were used with a parallel efficiency of 75%. However, the parallel performance of the algorithm is influenced by the synchronous nature of the algorithm where all analyses within an iteration must be completed before the next iteration is started.

Overall, the filter-based SQP algorithm is a good candidate for efficiently solving typical engineering problems with up to a few hundred design variables in a parallel environment where the number of processors is less than or equal to the number of design variables. The algorithm provides the efficiency of an established gradient-based algorithm and the parallel efficiency associated with a one step iterative procedure.

### References


