A Load Balancing Tool for Structured Multi-Block Grid CFD Applications

K. P. Apponsah∗ and D. W. Zingg†

University of Toronto Institute for Aerospace Studies (UTIAS), Toronto, ON, M3H 5T6, Canada
Email: kwesi@oddjob.utias.utoronto.ca

ABSTRACT
Large-scale computational fluid dynamics (CFD) simulations have benefited from recent advances in high performance computers. Solving a CFD problem on 100 to 1000 processors is becoming commonplace. Multi-block grid methodology together with parallel computing resources have improved turnaround times for high-fidelity CFD simulations about complex geometries. The inherent coarse-grain parallelism exhibited by multi-block grids allows for the solution process to proceed in separate sub-domains simultaneously. For a solution process that takes advantage of coarse-grain parallelism, domain decomposition techniques such as the Schur complement or Schwarz methods can be used to reduce the solution process for the whole domain to solving smaller problems in parallel for each partition on a processor. We present a static load balancing algorithm implemented for a parallel Newton-Krylov solver which solves the Euler and Reynolds Averaged Navier-Stokes equations on structured multi-block grids. We show the performance of our basic load balancing algorithm and also perform scaling studies to help refine the basic assumption upon which the algorithm was developed in order to reduce the cost of computation and improve the workload distribution across processors.

Key words: high performance computers, computational fluid dynamics, Newton-Krylov, load balancing, multi-block grid applications, domain decomposition

1 INTRODUCTION
CFD has grown from a purely scientific discipline to become an important tool for solving fluid dynamics problems in industry. The use of CFD in solving fluid dynamics problems has become attractive due to its cost-effectiveness and ability to "quickly" reproduce physical flow phenomena compared to performing experiments [3].

However, for large scale and complex simulations, massively parallel computers are required to achieve low turnaround times. High performance computers or massively parallel computers are computer systems approaching the teraflop (10¹² Floating-point Operations Per Second — FLOPS) region. Generally, the expectation is that such high-end computing platforms will reduce computational time and also provide access to the large amounts of memory usually required for large-scale CFD applications.

For multi-block grid applications, blocks can either be of the same size or different sizes. Therefore, to achieve a good static load balance several blocks can be mapped onto a single processor by some algorithm dictated by the strategy employed for the CFD solution process.

The objective of this paper is to develop an automatic tool to split blocks and also partition workload as evenly as possible across processors, in order to derive maximum benefit from a high-end computing platform.

2 NUMERICAL METHODOLOGY
In this section, we highlight the key features of our flow solver. We explain the motivation for our choice of grids and finite-difference discretization strategy. Furthermore, we give an overview of the numerical solution method and parallel implementation considerations for our solver.
2.1 Multi-block Grids and Discretization

As the complexity of geometry increases, single-block structured grids become insufficient for obtaining accurate numerical solutions to the flow equations. Multi-block grids, either homogeneous or heterogeneous at the block level, but structured within a block provide geometrical flexibility and preserve the computational efficiency of finite-difference methods. In addition, multi-block grids allow for direct parallelization of both grid generation and the flow codes on massively parallel computers [9].

Multi-block finite-difference methods also present some challenges that must be addressed. For example, for a CFD algorithm that uses the same interior scheme to discretize nodes at block interfaces through the use of ghost or halo nodes, it is required that the mesh be sufficiently smooth in order to retain the desired accuracy. To eliminate the challenges associated with block interfaces and exceptional points, we use the summation-by-parts (SBPs) [1, 2, 4] finite-difference operators and simultaneous-approximation terms (SATs) [1, 2, 4] for discretization.

An SBP operator is essentially a centered difference scheme with a specific boundary treatment, which mimics the behaviour of the corresponding continuous operator with regard to certain properties. We discretize the Euler and Reynolds Averaged Navier-Stokes (RANS) equations independently on each block in a multi-block grid using the SBP operators. The domain interfaces are treated as boundaries, and together with other boundary types the associated conditions are enforced using SATs. For further information on SBP operators and SATs, we direct the reader to the literature on these topics [6, 11].

2.2 Solution Method and Parallel Implementation Considerations

We use an inexact Newton method [12] to solve the nonlinear equations resulting from the discretization of the steady Euler and RANS equations. The resulting sparse linear system of equations at each outer iteration is solved inexactly using a preconditioned Krylov solver, the flexible generalized minimal residual method (FGMRES) [7]. An approximate Schur [8] preconditioner is used during the Krylov solution process, with slight modifications to the original technique, basically to improve CPU time [4].

For domain decomposition, we assign one block to a processor or combine blocks to form a partition on each processor when the number of blocks exceeds the number of processors. On each partition the following main operations in our solution process are parallelized: Block Incomplete LU factorization (BILU(k)), inner products, and matrix-vector products required during the Krylov stages. For example, to compute a global inner product, we first compute local inner products in parallel and then sum the components local to each processor by using the MPI command MPI_Allreduce(). With regard to communication, we use the MPI non-blocking communication commands for data exchanges. This communication approach has the advantage of partially hiding communication time, since the program does not wait while data is being sent or received, thus some useful work which does not require the data to be exchanged can be done.

Previously, our ability to obtain a good static load balance was restricted to homogeneous multi-block grids and to processor numbers which are factors of the number of blocks present in a grid. In order to accommodate more general grids and offer more flexibility in terms of the number of processors, an automatic load balancing tool was needed.

3 Load Balancing Technique

The load balancing algorithm implemented is based on the coarse-grain parallelism approach. Entire blocks are assigned to a processor during computation. The load balancing tool proposed is static, that is, block splitting and workload partitioning are done before flow solution begins. In addition the tool can handle cases where the number of blocks is less than the number of processors.

3.1 Workload Distribution Strategy

The assumption is that the CPU time per node remains constant. Blocks are sorted in descending order of size and at every instance of block assignment the processor with the minimum work (nodes) receives a block [5]. The processors which do not meet the load balance threshold set have their largest block split. Two constraints are used to control block splitting. The first is the block size constraint, $f_b$, defined as:

$$f_b = \frac{w_i}{\min\{w_{i}^{(j)}\}}, \quad i = 1, 2, 3, \ldots, n_b$$

(1)

where

- $w_i =$ number nodes in a block
- $n_b =$ number of blocks
\( w_j^{(o)} \) = number of nodes in a block in the initial mesh.

The block size constraint is enforced when the number of blocks is less than the number of processors, and blocks vary in size. It restricts a large block from being greater than the smallest block in the initial mesh by more than a factor specified by the user. The second constraint is the workload threshold factor, \( f_i \), which is defined as:

\[
 f_i = \frac{\max(w_j)}{\bar{w}}, \quad j = 1, 2, 3, \ldots n_p
\]

where

\[
 \bar{w} = \frac{\sum_{i=1}^{n_b} w_i}{n_p}
\]

\( n_p = \) number of processors

The workload threshold factor ensures that the workload per processor does not exceed the average workload by more than the factor specified by the user. We also define a correction factor, \( c \), as:

\[
 c = \frac{\bar{w}^{(f)}}{\bar{w}^{(o)}}
\]

where \( \bar{w}^{(o)} \) and \( \bar{w}^{(f)} \) are the workload averages during initial and the final load balance iterations respectively. The parameter, \( c \), is a measure of the increase in volume-to-surface ratio due to block splitting. For problems that are difficult (i.e. small \( f_i \) and a smaller number of blocks compared to processors) to load balance we observe that \( c \) is high. In the next section we provide further insight into the effects of this parameter on the parallel scaling of problems that use the load balancing tool developed.

We present some results for a homogeneous grid with 64 blocks for the ONERA M6 wing and a heterogeneous grid with 569 blocks for the Common Research Model Wing-Body (CRM) used for the Drag Prediction Workshop IV. Table 1 shows the flow conditions and some properties of the multi-block grids used.

<table>
<thead>
<tr>
<th>Properties</th>
<th>ONERA M6 - Euler</th>
<th>CRM - RANS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n_p )</td>
<td>64</td>
<td>569</td>
</tr>
<tr>
<td>block(_{min})</td>
<td>(37 \times 37 \times 37)</td>
<td>(11 \times 11 \times 15)</td>
</tr>
<tr>
<td>block(_{max})</td>
<td>(37 \times 37 \times 37)</td>
<td>(44 \times 41 \times 43)</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>3.06(^{\circ})</td>
<td>2.356(^{\circ})</td>
</tr>
<tr>
<td>( Re )</td>
<td>-</td>
<td>(5.0 \times 10^{6})</td>
</tr>
<tr>
<td>( M )</td>
<td>0.699</td>
<td>0.85</td>
</tr>
</tbody>
</table>

Table 1: Mesh properties and flow conditions for test cases.

3.2 Block-splitting Tool

The block splitting tool is based on recursive edge bisection (REB) [10]. Blocks are split half way or at the nearest half way point along the index with the largest number of nodes. This tool can be used independent of the main load balancing loop as well.

4 RESULTS

In this section, we discuss some of the results obtained so far and suggest some possible improvements to the basic algorithm described with regard to our flow solution algorithm. Lastly, we show how load balancing can improve turnaround times for CFD problems especially in the case of heterogeneous multi-block grids.

4.1 Homogeneous Multi-block Grids

We refer to homogeneous multi-blocks grids as grids with blocks of the same size throughout the domain. We consider two cases based on the initial number of processors started with for the scaling studies, since we extrapolate the ideal time based on this starting point. We define an ideal time for a number of processors for a constant problem size as:

\[
 t_p,\text{ideal} = \frac{t_1}{n_p}
\]

where \( t_1 \) is the computational time if the problem is run on a single processor (serial case) and \( n_p \) is the number of processors. Since we do not run any of the cases considered on a single processor we modify equation (4) slightly to:

\[
 t_p,\text{ideal} = \frac{t^{(o)}_p \times n^{(o)}_p}{n_p}
\]

where \( t^{(o)}_p \) is execution time for the smallest number of processors in the scaling studies group. It should be noted that \( t_p,\text{ideal} \) does not account for the changing size (measured with \( c \)) of each problem.

In Figure 1, we show a combined scaling diagram for the two distinct cases considered. In addition we include the ideal cases based on the two extrapolation points, 16 and 180 processors respectively. From Figure 1 we observe that 128, 320, 640, 720 and 960 processors we obtained lower execution times compared to the processors proceeding them. One would expect that we observe a lower turnaround time for 180 processors than 128 processors and 460 processors than 320, but this is not the case. This anomaly
can be attributed to the fact that 128, 320, 640, 720, 960 and 1024 are multiples of the initial number of blocks (64). If the ratio of the number of processors to the number of initial blocks is equal to a power of 2 then we obtain a perfect load balance considering our block splitting strategy. Therefore for 128 and 1024 processors we obtain a perfect load balance. In the case (320, 640, 720 and 960 processors) where the number of processors to number of blocks ratio is not equal to a power of 2 we still obtain a near perfect load balance compared to the subsequent processors, hence recording a low turnaround time. Therefore for homogeneous multi-block grids low turnaround times and better performance can be achieved when the number of processors is a multiple of the number of blocks.

We show the parallel efficiency of the processors that are multiples of the number of blocks in Tables 2 and 3. Table 2 is computed with respect to the time measured for 16 processors while for Table 3 we use 180 processors.

![Figure 1: Execution times obtained for 16, 30, 48, 72, 128, 180, 240, 250, 320, 460, 500, 580, 640, 720, 760, 840, 960 and 1024 processors.](image)

<table>
<thead>
<tr>
<th>Processors</th>
<th>$f_t=1.10$</th>
<th>$f_t=1.20$</th>
<th>$f_t=1.40$</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>69%</td>
<td>69%</td>
<td>69%</td>
</tr>
<tr>
<td>320</td>
<td>40%</td>
<td>40%</td>
<td>53%</td>
</tr>
<tr>
<td>640</td>
<td>29%</td>
<td>29%</td>
<td>39%</td>
</tr>
<tr>
<td>720</td>
<td>33%</td>
<td>33%</td>
<td>33%</td>
</tr>
<tr>
<td>960</td>
<td>21%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>1024</td>
<td>28%</td>
<td>29%</td>
<td>29%</td>
</tr>
</tbody>
</table>

Table 2: Parallel efficiency for 128, 320, 640, 720, 960 and 1024 processors using 16 processors as the extrapolation point for the ideal time.

4.1.1 Case I: $n_b^{(o)} > n_p^{(o)}$

For this case our initial 64-block mesh is started off with 16 processors. We consider three different load balancing thresholds, 1.10, 1.20 and 1.40. Figure 2 shows how our code scales as we increase the number of processors. Since we start with a smaller number of blocks compared to processors, blocks have to be split several times in order to meet the workload threshold set. In cases where a low $f_t$ is set or the number of processors far exceeds the number of initial blocks, a high $c$ results due to the increased number of block splittings. For example in the case of 250 processors $c$ goes as high as 1.15 as seen in Figure 3 when $f_t=1.10$. This means the computational nodes in the initial grid increase by 15% after the final load balancing iteration. Considering our parallel solution process, a high $c$ results in longer preconditioning time and a higher number of Krylov iterations as observed in Figures 4 and 5.

During the Schur preconditioning stages we obtain approximate solution values for all the internal interface nodes. The approximate solution to the Schur complement system is accelerated using GMRES. Consider a single block of size $37 \times 37 \times 37$ which is split into two in all three index directions resulting in 8 blocks, we record a $c$ value of approximately 1.08. This ~8% increase in the size of the problem results in a ~100% increase in the number of interface nodes translating into a bigger Schur complement problem, therefore increasing preconditioning time significantly. Figure 4 shows the CPU time spent at the preconditioning stages for the cases in Figure 2.

In [4], Hicken and Zingg showed that the number of Krylov iterations required to solve the linear system at each inexact-Newton iteration of our flow solver using the approximate Schur preconditioning technique is independent of the number of processors, but dependent on the size of the problem. The implication of the preceding conclusion for our load balancing tool
is that the total Krylov iterations required to solve a problem will remain the same for any two or more problems that have the same \( c \) irrespective of the \( f_t \) set and the number of processors used. For instance, in Figure 3, for processor numbers 72 and 250 and \( f_t \) set to 1.10 and 1.40 respectively, the problem size is the same with a \( c \) of 1.08 and we record approximately the same number of Krylov iterations. The value of \( c \) begins to grow steadily as we increase the number of processors due to the small number of blocks started with, which in turn leads to increased Krylov iterations and increased computational time.

To help improve the scalability of our code for difficult load balancing problems we introduce a new exit criterion for our main load balancing loop. With the knowledge that a small \( f_t \) ensures a good static load balance but does not necessarily improve the scalability of our code in terms of overall CPU time, Krylov iterations and preconditioning time, we relax the strict enforcement of \( f_t \) by allowing it to operate only when a blocks-to-processors ratio (\( b_r \)) constraint has not been met. This new constraint is set by the user as well. Once the blocks-to-processors ratio for an iteration is equal or greater than \( b_r \), the main load balancing iteration loop is exited even if \( f_t \) has not been met. The lines in Figures 2, 4 and 5 indicated with "new" show the effect of introducing this new constraint. We show results only for \( f_t = 1.10 \), \( b_r = 2.0 \) and processors 30, 48, 72, 250, 500. The processors selected had \( b_r \) greater than 2.0 after the final iterations for the initial runs without the introduction of \( b_r \). Table 4 shows the percentage reductions obtained for overall CPU time, preconditioning time and Krylov iterations for the large number of processors which are of much interest to us in these scaling studies.

<table>
<thead>
<tr>
<th>Processors</th>
<th>CPU time</th>
<th>Precon time</th>
<th>Krylov its</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>12%</td>
<td>20%</td>
<td>8%</td>
</tr>
<tr>
<td>250</td>
<td>5%</td>
<td>13%</td>
<td>7%</td>
</tr>
<tr>
<td>500</td>
<td>6%</td>
<td>15%</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 4: % reductions obtained after the introduction of \( b_r = 2.0 \).
4.1.2 Case II: $n_b^{(o)} < n_p^{(o)}$

We consider this case mainly to draw the reader’s attention to how the overall CPU time scales with respect to the initial number of blocks in comparison to the number of processors used to extrapolate the ideal CPU time. Figure 6 shows the scaling performance. In the previous case, a perfect static load balance was obtained for the smallest number (16) of processors without any block splitting. But in this case, some block splitting was required for the smallest number (180) of processors. Although the case with $f_t=1.40$ tends to scale well compared to the other cases, we still observe poor scaling with increasing number of processors. Further investigation into the use of different parallel preconditioning techniques is being considered. In a previous study performed using the Schwarz preconditioner we realised that for cases where the ratio of number of blocks to the number of processors is high, Schwarz preconditioning improves CPU time compared to the Schur approach. We intend to conduct further study in this direction to take advantage of either preconditioning technique to gain further improvement in CPU time. Figure 7 shows the turnaround times observed in our preconditioning comparison study using a 16-block ONERA M6 mesh. The initial grid was then split into 32, 64, 128, 256, 512 and 1024 blocks. All the cases were run on 16 processors using the Euler equations. The ideal case is extrapolated by multiplying the time for 16 blocks by the increase in problem size, $c$.

4.2 Heterogeneous Multi-block Grids

We refer to heterogeneous multi-block grids as grids with different block sizes. The results for this case were obtained with the CRM grid using the RANS equations with the Spalart-Allmaras one-equation turbulence model. The case considered in this section supports our view that a load balancing tool can significantly improve turnaround times for large-scale problems. We run the initial 569 blocks on 569 processors without any load balancing. The average of three different runs is shown with the * symbol in Figure 8. We run this case with load balancing, setting $f_t$ to 1.10, 1.20 and 1.40 on 190, 280, 370, 450, 560 and 740 processors. We observe that with load balancing we are able to obtain lower turnaround times compared to the CPU time measured for the case without load balancing even on fewer than 569 processors.
The parallel efficiency for 740 processors with $f_t = 1.20$ is $\sim 68\%$.

5 CONCLUSIONS

We described our basic load balancing algorithm and presented some results on workload distribution and computational times recorded for the different processor numbers considered. In Figures 1, 2 and 6 we showed the scaling performance of our Newton-Krylov CFD code and how our extrapolation point for computing ideal CPU times impacts scaling diagrams.

To improve on CPU time for computation and preconditioning, and the number of Krylov iterations, we introduced a new constraint to control block splitting during the use of our load balancing tool. We show the improvements gained from introducing this new constraint in Figures 4 and 5 and Table 4. In Figure 7 we show the impact of block splitting on CPU time considering different preconditioning techniques. Finally, we show in Figure 8 how the automatic tool for load balancing can lead to reduced turnaround times for large-scale multi-block CFD applications with heterogeneous meshes.

APPENDIX A: LOAD BALANCING ALGORITHM

$$W_{b} = (w_1, w_2, ..., w_{nb})$$
$$w_i , i = 1, nb$$
$$w_{p,j} , j = 1, np$$
$$w_r,j , j = 1, np$$
$$p_x = 0$$

repeat

$$w = \frac{\sum_{j=1}^{nb} w_{b,j}}{np}$$
$$w_{p,j} = 0 , j = 1, np$$

for $i = 1$ to $nb$ do

$$w_{p, \text{min}(j)} = w_{p, \text{min}(j)} + w_i$$

end for

for $i = 1$ to $np$ do

$$w_r,j = \frac{w_{p,j}}{w}$$

if $w_r,j > f_t$ then

$$p_x = p_x + 1$$

end if

end for

if $p_x > 0$ then

Split blocks associated with $w_r , i = 1, p_x$

$$n(W_{nb,new}) = n(W_{nb}+p_x)$$

$$W = W_{new}$$

end if

until $p_x = 0$ or $\frac{nb}{p_x} >= b_r$

$nb = \text{number of blocks in grid file}$

$np = \text{number of processors specified}$

$W_{nb} = \text{is the set of initial block sizes sorted in descending order}$

$W_{nb,new} = \text{is the set of new block sizes sorted in descending order}$

$w_i = \text{block workload}$

$w_p = \text{processor workload}$

$w_r = \text{processor workload ratio}$

$p_x = \text{number of processors with workload exceeding load balance threshold}$

$f_t = \text{load balance threshold specified}$

$b_r = \text{blocks-to-processor ratio}$

$\text{min}(j) = \text{index of } w_{p,j} \text{ with the minimum workload}$

ACKNOWLEDGEMENTS

The authors gratefully acknowledge financial assistance from Mathematics of Information Technology and Complex Systems (MITACS), Canada Research Chairs Program and the University of Toronto. All computations were performed on the GPC supercomputer at the SciNet\footnote{SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.} HPC Consortium.

REFERENCES


