Design and analysis of two highly scalable sparse grid combination algorithms

Peter E. Strazdins a, *, Md. Mohsin Ali a, Brendan Harding b

a Research School of Computer Science, The Australian National University, Australia
b Mathematical Sciences Institute, The Australian National University, Australia

A R T I C L E   I N F O

Article history:
Received 7 November 2015
Received in revised form 9 May 2016
Accepted 20 June 2016
Available online 20 August 2016

Keywords:
High performance computing
Parallel computing
PDE solvers
Sparse grid combination technique
Algorithm-based fault tolerance

A B S T R A C T

Many large scale scientific simulations involve the time evolution of systems modelled as Partial Differential Equations (PDEs). The sparse grid combination technique (SGCT) is a cost-effective method for solve time-evolving PDEs, especially for higher-dimensional problems. It consists of evolving PDE over a set of grids differing resolution in each dimension, and then combining the results to approximate the solution of the PDE on a grid of high resolution in all dimensions. It can also be extended to support algorithmic-based fault-tolerance, which is also important for computations at this scale.

In this paper, we present two new parallel algorithms for the SGCT that supports the full distributed memory parallelization over the dimensions of the component grids, as well as across the grids as well. The direct algorithm is so called because it directly implements a SGCT combination formula. We give details of the design and implementation of a 'partial' sparse grid data structure, which is needed for its efficient implementation. The second algorithm converts each component grid into their hierarchical surpluses, and then uses the direct algorithm on each of the hierarchical surpluses. The conversion to/from the hierarchical surpluses is also an important algorithm in its own right. It requires a technique called sub-gridding in order to correctly deal with the combination of very small surpluses. An analysis of both indicates the direct algorithm minimizes the number of messages, whereas the hierarchical surplus minimizes memory consumption and offers a reduction in bandwidth by a factor of \(1 - \frac{d}{4}\), where \(d\) is the dimensionality of the SGCT. However, this is offset by its incomplete parallelism (70–80%) and a factor of \(2^d\) load imbalance in practical scenarios. Our analysis also indicates both are suitable in a bandwidth-limited regime and that the direct algorithm is scalable with respect to \(d\). Experimental results including the strong and weak scalability of the algorithms indicates that, for scenarios of practical interest, both are sufficiently scalable to support large-scale SGCT but the direct algorithm has generally better performance, at least by a factor of 2 in most cases. Hierarchical surplus formation is much less communication intensive, but shows less scalability with increasing core counts. Altering the layout of processes in the process grids and the mapping of processes affects the performance of the 2D SGCT by less than 10%, and affects even less the application part of an SGCT advection application.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Very large scale scientific simulations are capable of making great advances in computational science. Usually, these involve the numerical solution of Partial Differential Equations (PDEs). There are two main obstacles for scientific applications in order to reach this goal: achieving sufficient scalability, and ensuring their reliable completion. Both of these problems arise due the immense size, and hence number of components, of supercomputing systems at these scales. The latter problem arises due to the fact that these simulations are typically long-running, and that the failure rate of the system is proportional to the number of components.

PDEs are normally solved on a regular grid. With uniform discretization across all its dimensions, the number of grid points increases exponentially with the increase of dimensionality. This behavior makes the high-dimensional PDE solver computationally expensive. In order to address this issue, the PDE can be solved on a grid with substantially fewer grid points than the regular full grid. These grids are called sparse grids [1]. An example of a sparse grid is shown in Fig. 1, where the sparse grid is represented as the union of smaller regular grids.

A numerical method called the Sparse Grid Combination Technique (SGCT) [2,3] can be employed to approximate the solutions to PDEs on the sparse grid. Instead of solving the PDE on a full regular

* Corresponding author.
E-mail address: Peter.Strazdins@cs.anu.edu.au (P.E. Strazdins).
grid, the original PDE solver is employed to obtain solutions on anisotropic grids with substantially fewer grid points, called the sub-grids or component grids, as shown in Fig. 1. The SGCT itself involves linearly combining these solutions to approximate the solution on the full grid. The technique can be applied in principle to any PDE; indeed it can be applied to any system whose solution is a field over a regular grid, with the SGCT being applied to at least 2 of the dimensions of this grid. Whether the SGCT achieves a good tradeoff in accuracy per amount of work performed depends on whether the solution has sufficient smoothness. The SGCT is known to work particularly well if the error of the approximate solutions can be expressed in specific form often referred to as an error splitting, see for example [3]. The SGCT has been applied with varying degrees of success to a wide range of problems including elliptic PDEs [2], simulation of turbulent flows [4] and stochastic PDEs in finance [5]. Other systems with solutions of sufficient smoothness for good accuracy with the SGCT include simulations of micro-turbulence in gyrokinetic plasmas [6,7] and the Taxila Lattice-Boltzmann Method (merging of two partially miscible fluids – the ‘bubble test’) and Solid Fuel Ignition simulations [8].

An important technique associated with sparse grids is the formation of hierarchical surpluses [9,10] on regular grids. These facilitate error analysis over the various dimensions at various levels of resolution, and can be used to determine when refinement is required on adaptive grids [10].

As well as making high-dimensional PDEs tractable, employing the SGCT can result in computational efficiencies even for lower-dimensional problems [2,6,7]. Furthermore, it inherently supports parallelism in the sense that component grids can be solved completely independently. The SGCT can also be extended to support algorithm-based fault-tolerance [11,12,7,8]. Upon detection of a fault, the SGCT can be applied in such a way that the component grids associated with the failed nodes are avoided. The lost data can be recovered by down-sampling from the combined (sparse) grid, and the computation can continue sustainably. The ability to repeatedly apply the SGCT over a computation is also useful in improving the accuracy of the sparse grid approximation in certain cases [6–8].

Thus, there is an urgent need to have highly parallel SGCT and hierarchical surplus formation algorithms for where the component grids themselves are large and must be solved in parallel. The contributions of this paper are to (i) present two new algorithms for the SGCT which support fault-tolerance and are suitable for computations over a large number of cores and (ii) perform an analysis and detailed experimental evaluation of their performance. To the best of our knowledge, this is the first work that does so for a SGCT algorithm which supports distributed memory parallelism both across and within the component grids. The direct algorithm is furthermore more generally in terms of the process grid sizes used for the component grids (the hierarchical surplus algorithm requires these to be powers of 2). We also believe that we also give the first general distributed memory algorithm for the conversion of a grid to/from its hierarchical surpluses. We also are the first to discuss the concept of a partial sparse grid, and give the first implementation of a distributed sparse grid data structure. In order to do so, we introduce the concept of the sparse block distribution, which we believe is novel. Furthermore, we show how the hierarchical surpluses can be coalesced and scheduled, and show that the former is crucial for its efficiency. We give a deep analysis of the scaling behaviour of these algorithms, followed by a comprehensive experimental evaluation. Finally, we expect that our approach to algorithm design and implementation using vector arithmetic and associated distribution mappings will be applicable to other communication-intensive algorithms over high dimensional spaces.

The paper is organized as follows. Section 2 describes the SGCT technique. Details of our implementation of a highly scalable SGCT algorithm are given in Section 3, followed by an analysis in Section 4. Experimental results are given in Section 5, related work is discussed in Section 6, and conclusions are given in Section 7.

2. The sparse grid combination technique

Consider the SGCT for the 2D case. Each component sub-grid $G_l$, where $l=(i_x, i_y)$, is assumed to have $(2^l+1) \times (2^l+1)$ grid points with a grid spacing of $h_1=2^{-i_x}$ and $h_2=2^{-i_y}$ in the $x$- and $y$-directions, respectively, where $i_x, i_y \geq 0$. If we consider a square domain, then the grid points of $G_l$ are

$$\{ (x, y) \mid x = 0, 1, \ldots, 2^l; y = 0, 1, \ldots, 2^l \}.$$

In the more general case, the index space for the grids will be some finite set $I \in \mathbb{N}^d$. If $u_l$ denotes the approximate solution of a PDE on $G_l$, the combination solution $u^*_l$ generally takes the form

$$u^*_l = \sum_{i \in I} c_i u_i,$$

where $c_i \in \mathbb{R}$ are the combination coefficients. Clearly, the accuracy of the combination technique approximation depends on the choice of the index space $I$ of the sub-grids and their respective coefficients. For the 2D case, good choices of the coefficients are ±1. For instance, in the classical case, we have for level $l$ the set $I = \{(i_x, i_y) | i_x, i_y \geq 0, l - 1 \leq i_x + i_y \leq l\}$ and the combination coefficients are $c_i = 1$ if $i_x + i_y = l$ and $c_i = -1$ if $i_x + i_y = l - 1$ where $i = (i_x, i_y)$. This provides the following combination formula

$$u^*_l = \sum_{i \in I} u_i - \sum_{i \in I, i_x + i_y = l-1} u_i$$

which is depicted in Fig. 2.

In contrast to the full grid approach which needs $O(h^n_d)$ grid points, the SGCT works with only $O\left( h_1^d \log_2(h_1^{-1}) d^{-1} \right)$ grid points, where $h_1$ denotes the employed grid spacing and $d$ is the dimension. The accuracy of the solution obtained from the SGCT deteriorates only slightly from $O(h_2^n)$ to $O\left( h_2^d \log_2(h_1^{-1}) d^{-1} \right)$ for a sufficiently smooth solution.
2.1. Hierarchical surpluses

Approximations on component grids are typically computed and stored using a nodal basis representation, that is each element \( v_k \) in the vector \( v \) gives the function’s value at the grid point \( x_k = k \cdot 2^{-i} \), that is \( v_k = f(x_k) \). An alternative representation is via the hierarchical basis [9] whereby each element of the vector is the difference between the function’s value’s at the corresponding grid point and the function’s values at the hierarchical neighbours. Writing \( v \) using the level notation \( v_{l,k} \) where \( l \in \{0, \ldots, l\} \) and

\[
k \in B_l = \begin{cases} 
\{i \in \mathbb{N} : 0 < i < 2^l, i \text{ odd} \} & \text{if } l > 0 \\
\{0, 1\} & \text{if } l = 0,
\end{cases}
\]

one has the hierarchical representation

\[
v_{l,k} = \begin{cases} 
\left(f(x_{l,k}) - \frac{1}{2} \left(f(x_{l-1,(k-1)/2}) + f(x_{l-1,(k+1)/2})\right)\right) & \text{for } l > 0 \tag{3} \\
f(x_{l,k}) & \text{for } l = 0
\end{cases}
\]

where \( x_{l,k} := x_k \cdot 2^{-i} \). These \( v_{l,k} \) correspond to the heights of the hierarchical basis functions depicted in Fig. 3. The collections \( \{v_{l,k}\}_{k \in B_l} \) for each \( l \) are referred to as hierarchical surpluses. The hierarchical basis function in 2 or more dimensions is obtained by taking the tensor product of the one dimensional hierarchical basis functions. For full details we refer the reader to [9,10]. Rather than perform the combination technique directly over the component grid approximations (in nodal basis) one may instead convert each component grid to hierarchical basis, combine each of the hierarchical surpluses (distributing the result) and then reconstruct the nodal basis for each component grid. This is advantageous over direct combination for two reasons: (1) it reduces the communication volume [13], (2) it allows one to avoid interpolation onto a full grid or sparse grid structure thus reducing compute cycles and memory requirements.

The formation of the hierarchical surpluses involves applying Eq. (3) across each of the dimensions of a component grid \( G_j \). Each element in a grid will correspond to a hierarchical surplus of index \( j \), where \( j \leq l \). This is illustrated for \( i = (3, 3) \) in Fig. 4. The hierarchical process occurs in-place, with the surpluses computed from the initial grid values.

Consider the hierarchization across the \( x \) dimension in row 0. The elements with hierarchy 30 are formed by subtracting the average of the elements at distance 1 to the left and right (‘30’ is a shorthand for ‘3, 0’). Then elements with 20 are formed by subtracting the average of the elements at distance 2 from right or left. Finally, the single element of hierarchy 10 is computed by subtracting the average of the elements in hierarchy 00 at a distance 4 to the left and right.

\[
\begin{array}{cccccccc}
00 & 30 & 20 & 10 & 0 & 30 & 20 & 00 \\
03 & 33 & 23 & 13 & 33 & 23 & 33 & 03 \\
02 & 32 & 22 & 12 & 32 & 22 & 32 & 02 \\
03 & 33 & 23 & 13 & 33 & 23 & 33 & 03 \\
01 & 31 & 21 & 11 & 31 & 21 & 31 & 01 \\
03 & 33 & 23 & 13 & 33 & 23 & 33 & 03 \\
02 & 32 & 22 & 12 & 32 & 22 & 32 & 02 \\
03 & 33 & 23 & 13 & 33 & 23 & 33 & 03 \\
00 & 30 & 20 & 10 & 30 & 20 & 30 & 00
\end{array}
\]

Fig. 4. Hierarchical surplus indices corresponding to the elements of a \( l=3 \) grid.

2.2. Truncated combinations

Our SGCT algorithms support so-called ‘truncated’ combinations [5], which avoids using the highly anisotropic grids (e.g., \( G_{(2)} \), where \( 1 \leq s < l \) close to the axes of the grid index space. This avoids the problem of minimum dimension size imposed by some applications [7]. Furthermore, the highly anisotropic grids have been known to contribute least towards the accuracy of the sparse grid solution or cause convergence problems [5], enabling us to concentrate process resources on the more accurate sub-grids. Finally, it removes the limitation that the approximated full grid need not be isotropic, i.e., have an index of \((l, l)\).

In this context we use a different notion of level to that described above in describing how much smaller the sub-grids are relative to some full grid \( G_p \). In particular, a level \( l \leq \min(l_x, l_y) \) combination in this context consists of sub-grids from the index set:

\[
l = \left\{(i_x, i_y) : \begin{array}{c}
\left(\frac{i_x}{l} + 1, \frac{i_y}{l} + 1\right) \leq (k_x, k_y) \\
\frac{i_x}{l} + 1 \leq k_x + k_y + 1 - l
\end{array} \right\}. \tag{4}
\]

It can be noted from the above that the truncated combination formula is a generalization of the standard SGCT formulation, which has \( l = (l, l) \).

In general, the choice of \( l \) in the truncated formula determines the reduction in the amount of work to solve a component grid, compared to the full grid. For 2D, it is by a factor of \( 2^{l-1} \); for 3D it is \( 2^{2l-1} \). We thus need \( l = 4 \) (2D) and \( l = 3 \) (3G) for the total amount of work for the SGCT to be substantially less than that of performing a single simulation on the full resolution grid. Increasing \( l \) beyond this point reduces work but with a possible loss of accuracy. We also need \( l \geq 4 \) for fault-tolerant versions of the SGCT [7,8].

3. Parallel SGCT algorithms

In this section, we present first the direct and then the hierarchical surplus based algorithms. Finally, we discuss their current limitations and how they may be extended.

These algorithms have a succinct description when expressed via \( d \)-dimensional vector arithmetic, and we use the following notations (explained for the case \( d=2 \)). Given \( M, N \in \mathbb{N}^d \), and \( a \in \mathbb{N} \), then \( M \leq N \) means \((M_x \leq N_x) \wedge (M_y \leq N_y)\) and \( a \leq N \) means \((a \leq N_x) \wedge (a \leq N_y)\). Also \( M+N = (M_x+N_x, M_y+N_y) \) and \( a+N = (a+N_x, a+N_y) \). This applies to all other (integer) arithmetic operators.
We use conditional notations $C_{op}(M, N) \in \mathbb{N}^d$ and $I(C, M, N) \in \mathbb{N}^d$, where $C \in \mathbb{N}^d$ and:

$$(C_{op}(M, N))_d = \begin{cases} 1, & \text{if } M_d \ op \ N_d \\ 0, & \text{otherwise} \end{cases}$$

$$(I(C, M, N))_d = \begin{cases} M_d, & \text{if } C_d \neq 0 \\ N_d, & \text{otherwise} \end{cases}$$

******* for $d = x, y$ and ‘op’ is a relational operator (e.g., $=$, $<$). We also use the vector to scalar notations $\Sigma(a)$ and $\Pi(a)$ for the sum and product of elements of $a$, i.e., $\Sigma((a, a_p)) = a + a_p$ and $\Pi((a, a_p)) = a \cdot a_p$ in the 2D case.

Crucial to their description (and to the reliable construction of their implementations [14]) are the mappings of global grid lengths and offsets to local lengths and offsets (which break down in turn to process offsets and index offsets within a process). If $p \in \mathbb{N}^d$ is the id of a process on a process grid of size $P$, with $0 \leq p < P$, and given $\tilde{N} \in \mathbb{N}^d$ with $0 \leq \tilde{N} < N$, and letting $n = N/p$, we define the following vector functions for a block-distribution of $N$ over $P$:

$$I(N, p, P) = n + C_{op}(p, P - 1) \ast (N \ mod \ P)$$

$$g_0(N, p, P) = p \ast n$$

$$p(\tilde{N}, N, P) = \min(\tilde{N} / n, P - 1)$$

$$\sigma(\tilde{N}, N, P) = I(C_{op}(\tilde{N} / n, P - 1), \tilde{N} - n \ast (P - 1), \tilde{N} \ mod \ n)$$

where $I(N, p, P)$ is the local size of $N$ on process $p$, $g_0(N, p, P)$ is the global index of the 0th local point on $p$, $p(\tilde{N}, N, P)$ is the id of the process holding global point $\tilde{N}$, and $\sigma(\tilde{N}, N, P)$ is the local offset of global point $\tilde{N}$ on that process.

The above assumes that $N > P$. We now introduce what we will call the sparse block distribution which covers also the case $N < P$:

$$P(N, p, P) = I(C_{op}(N, p, N, n, P), P(N, p/2, P/2))$$

$$= I(C_{op}(N, p, N, n, P), C_{op}(p \ mod \ r, 0))$$

$$g_0(N, p, P) = I(C_{op}(N, p, g_0(N, p, P), g_0(N, p/2, P/2)))$$

$$= I(C_{op}(N, p, g_0(N, p, P), r)$$

where $r = \frac{P}{n}$. The above formulations assume $P$ and $N - 1$ are powers of 2. The sparse block distribution arises when we take power of two sub-vectors from a block distribution, as for example when forming hierarchical surpluses (see Fig. 4). When the sub-vectors become smaller than the process grid, we effectively form a new process grid where the processes holding no elements of the sub-vector are removed.

3.1. Direct SGCT

The presentation of our SGCT algorithm begins first with the scaled addition of a single sub-grid component into the combination solution. Each sub-grid is assumed to be distributed over an independent set of processes, arranged in a d-dimensional logical grid. The combination solution is similarly distributed over a logical process grid. The parallelization of this over each component grid forms a gather stage. This is illustrated in Fig. 5. For computations performing the SGCT repeatedly, it is necessary to reverse this process by sending the respective points of the combined grid back to the component sub-grids. This is called the scatter stage, and can be visualized by reversing the arrows in Fig. 5.

3.1.1. Single sub-grid components

Consider the scaled addition of a single grid into a full grid representation of the combination $u_{i}^{1} \leftarrow u_{i}^{1} + c_i u_i$ in Eq. (1). For brevity, we write this as $u' \leftarrow u' + c_i u_i$. Let $N, N' \in \mathbb{N}^d$ represent the global sizes of $u$ and $u'$, respectively. For the SGCT, the elements of $N - 1$, $N' - 1$ will be powers of 2 and $(N-1) = r^2 (N-1)$, where $r \in \mathbb{N}^d$.

Assume $u$ and $u'$ are distributed over process grids of size $P$ and $P'$ respectively ($P, P' \in \mathbb{N}^d$, $0 < p < N$ and $0 < P' < N'$).

On process $p$, the gather of $u$ to the appropriate process(es) on $P'$ is described in Algorithm 1. The loop structure is shown for $d = 2$; this generalizes in the obvious way for $d > 2$. The number of messages sent is bounded by $\Pi((P + P - 1)/P + 1)$. On line 9, note that a volume of grid points of size $dn + 1$ is sent: an extra uppermost row and column of points is required for interpolation. It is assumed that $u$ has storage for its $n$ points plus a ‘halo’ of neighbouring points to the positive direction, and that these have been filled by a halo exchange operation from the corresponding processes in $P$ prior to the start of the algorithm.

Algorithm 1. Sending of $u$ by process $p$ to be gathered by the corresponding processes on $P'$.

```plaintext
1. $N' = r g_0(N, p, P)$;
2. $p' = p(N', P', P')$;  $d' = \sigma(N', P', P')$;
3. $i = 0$;  $n = I(N, p, P)$;
4. while $i < n$ do
5.     while $i < n_d$ do
6.         $d' = C_{op}(i, 0)$;  // local offset @ $p'$
7.         $n' = I(N', p', P') - d' - o'$;  // local size @ $p'$
8.         $dn = \min(n'/r, n - i)$;  // local size here
9.         send local points $i : i + dn$ of $u$ to $p'$;
10.     $i_2 = i + dn_1$;  $p'_{i_2}$;
11. $i_2 = i + dn_2$;  $p'_{i_2}$;
```

1. Commonly, when solving a PDE on a regular grid, each process needs to acquire a ‘halo’ of neighbouring points from the surrounding processes in order to update each of its own points. If such an exchange has not occurred in the PDE solver, the SGCT algorithm itself will need to perform the exchange.
The receipt on process \( p' \) of the grid \( u \) (with combination coefficient \( c \)) and its interpolation into \( u' \) is essentially the mirror image of this and is described in Algorithm 2. The number of messages received is bounded by \( \Pi((P + P' - 1)/P' + 1) \).

**Algorithm 2.** Receipt of sub-grid \( u \) on process \( p' \) from the gather stage from the corresponding processes on \( P \).

1. \( \hat{N} = g_0(N', p', P')/r; \)
2. \( p = p(\hat{N}, N, P); \)
3. \( \delta = o(\hat{N}, N, P); \)
4. \( \alpha(i) = v'' = (\hat{N}', p', P'); \)
5. while \( i < n'_p \) do
6.     \( \phi = \delta + c_N(i, 0); \) // local offset \( 0 \)
7.     \( n = l(N, p, P) - \phi; \) // local size \( p \)
8.     \( dN' = \min(n, n'' - i); \) // local size here
9.     \( n = \min(u, dN' / r); \) // corresponding size \( p \)
10. receive \( n + 1 \) points from \( p \) to \( u \);
11. interpolate \( c \) in \( u \) to points \( i : i + dN' - 1 \) of \( u'' \);
12. \( i'' = dN' \); \( p_{\phi} + r; \)
13. \( i'' = dN'; \) \( p_{\phi} + r; \)

These algorithms are general for arbitrary \( P \); as expressed here, it assumes that on \( P, g_0(N', p', P') \mod r = 0 \), that is, the global indices of the 0th points on \( p' \) are a multiple of \( r \). This is satisfied for the SGCT if the elements of \( P' \) are powers of 2.

### 3.1.2. Overall SGCT algorithm and repeated SGCT support

Algorithm 1 is run in parallel on each \( P_i \), and Algorithm 2 is run iteratively over \( i \) (i.e., for each sub-grid) on \( P \). To support the latter, arrays over \( i \) containing the process grid \( (P_i) \), grid index and combination coefficient of each grid are maintained over all processes in \( P \).

Our implementation can take advantage of MPI::Irecv() semantics by allocating different buffers \( u \) for each grid \( i \), and overlapping the interpolation with the receipt of messages. Where repeated combinations are performed, it is necessary to scatter back the points of the combined grids (with appropriate down-sampling) on \( P \) iteratively to each sub-grid on \( P_i \). It is assumed that the processes in \( P_i \) have been restored by this stage if there was a failure in \( u \) detected for the gather stage. The scatter send stage on \( P \) is identical to Algorithm 2 except that lines 10–11 are replaced with:

sample points \( i : i + dN' - 1 \) of \( u' \) into \( u \); send the \( n \) points of \( u \) to \( p \).

The receipt of the scatter occurs in parallel over each process grid \( P_i \). The algorithm is identical to Algorithm 1 except line 9 becomes:

receive \( dN \) points from \( p' \) into \( u ; \) store \( v \) into points \( i : i + dN - 1 \) of \( u \).

### 3.1.3. Partial sparse grid data structure for efficient interpolation

On the receive stage of the gather operation, if we use a full grid representation of the combined grid \( u' \), the interpolation stage (line 11 of Algorithm 2) will involve interpolating on \( O(2l - 1) \) points in the full grid which are not represented by any point in any of the component grids. This is wasteful as these points are never used (in the sample step, above), in terms of both time and space, and furthermore this overhead is exponential in \( l \).

This can be avoided by interpolating onto a sparse representation of \( u' \). However, when using the truncated combination formula (Eq. (4)), the resulting data structure will be a generalization of both a regular full grid \( (l = 1) \) and a (classical) sparse grid \( (l = l_x = l_y) \). We call this representation a partial sparse grid, which represents exactly the union of points in all component grids. This section will describe how to create a distributed data structure for a partial sparse grid and the resulting interpolation and sample algorithms.

Fig. 6 gives a visualization of this data structure for level \( l = 3 \) onto grid \( C^2 \) where \( i = (4, 4, 4) \). This introduces a fill-in factor \( f_i = i - 1 = (1, 1, 1) \) where any gaps of size \( 2i^l \) in the corresponding sparse grid are filled in (as far as this is possible). Note that the smaller planes must be compressed first so that there are no gaps in their boundary points.

We use concepts similar to the CSR format for sparse matrices, except we take advantage of the regularity of the partial sparse grid to avoid column indexes. Recall that this corresponds to a full grid of size \( N' = 2^d + 1 \). In the 2D case, global row \( i, 0 \leq i < N_y' \), will have global length \( N_{Iy}(i) = 2^{l_y-i}l(i) + 1 \) where \( l(i) = Z(2^{i+1} + 2) \) and \( Z(n) \) gives the number of rightmost zeros before the first 1 bit in \( n > 0 \). The stride between the consecutive elements in this row that must be stored is \( 2^l \).

The local length in the x-dimension on process \( p' \) of process grid \( P' \) of the partial sparse grid is then given by

\[
n_{0}(i) = P_{Iy}(N_{Iy}(i), p_y' , P_y')
\]

where we are now using the scalar \( (d = 1) \) version of Eq. (9). We need to use this equation as now \( N_{Iy}(i) < P_y' \) is possible and hence we now need to assume \( P_y' \) is a power of 2.

Process \( P' \) computes these lengths for \( n_y = l(N_y', p_y' , P_y') \) rows beginning from local point 0 (global point \( i = g_0(N_y', p_y' , P_y') \)) and creates the required storage vector \( u' \) and a row index vector \( x_{ix}[0 : n_y - 1] \) accordingly. It caches the corresponding row strides in \( a[0 : n_y - 1] \).

**Algorithm 3.** Interpolation of part of a 2D component grid \( u \) of size \( n + 1 \), corresponding to \( n' \) points and local offset \( 0 \) of the full grid, onto the partial sparse grid \( u' \) \( r \) is the ratio of full to component grid sizes, and \( n' = r^* n \). \( c \) is the component grid’s combination coefficient.

1. for \( i = 0 ; i < n'_p ; i++ \) do
2. \( i' = i + o_y; \) \( s = i'[r]; \)
3. \( j_0 = (s - io_x) \mod s; \) \( j_0' = [\frac{y}{x}]; \)
4. for \( j = 0 ; j < \lceil \frac{n_x}{s} \rceil ; j++ \) do
5. \( i' = \lfloor \frac{i}{s} \rfloor; \) \( j = \lfloor \frac{j}{s} \rfloor; \)
6. add interpolant of \( c \) to \( u'[i' : i' + 1, j + 1 \) onto \( u'[x'[i'] + j'0'; j + 1]\).

Algorithm 3 gives the resulting code for the local interpolation step. It requires both alignment and scaling to take into account the grid row stride \( (s[i]) \). These calculations in turn assume that the first local element in each row corresponds to a global (full grid) index which is a multiple of \( s[i] \); this will be the case if \( P_y' \) is a power of 2, as assumed previously.

**Algorithm 4.** Sample \( n \) points from local full grid offset \( 0 \) from partial sparse grid \( u' \) into part of a component grid \( u \). \( r \) is the ratio of full to component grid sizes.

1. for \( i = 0 ; i < n'_p ; i++ \) do
2. \( i' = i + o_y + o_x; \) \( s = i'[r]; \)
3. \( j_0' = [\frac{y}{x}]; \)
4. for \( j = 0 ; j < n_x; j++ \) do
5. \( u[i, j] = u'[x'[i'] + j'0'; j + x_r / s]; \) // note: \( s[r_x \text{ and } r_x \geq 1] \)
The sample operation (Algorithm 4) iterates over the elements of the component grid instead of the partial sparse grid.

To generalize this to 3D, the \( y \) and \( z \) dimensions take the place of \( y \) and \( x \), respectively, in the above. For the \( j \)th row in \( y \) of \( i \)th plane in \( z \), the global length of this row is given by \( N_y(i,j) = 2^{l_i-j} + 1 \), where \( k(i,j) = 2^{l_i-j} + 2^{l_i-j} \).

### 3.1.4. Support for fault tolerance

As described in [7], the classical SGCT algorithm can be made fault-tolerant with alternate combination formulas requiring the inclusion of extra diagonals (2D) or planes (3D). We perform the simulation with these extra (redundant) grids. Before the SGCT is applied, failed processes are detected and recovered, and the grids of the failed processes are allocated a combination coefficient of \( c = 0 \). Assuming that these processes were associated with grid \( i \), the gather of \( u_i \) on \( P_i \) and \( P'_i \) is not performed.

A fault-tolerant communication infrastructure such as ULFM-MPI can be employed, and all surviving processes are informed which processes have failed. The corresponding grids of the failed processes are identified and will receive a combination coefficient of 0. The redundant grids in the presence of no failures have a combination coefficient of 0. However, they may receive a non-zero coefficient \( \geq 1 \) if one of the main grids on the same row or column has a failed process.

As mentioned in Section 3.1.2, the abstraction of having a simple array of grid data means that the algorithm is effectively oblivious to the extra requirements for a fault-tolerant SGCT.

### 3.2. Hierarchical surpluses-based algorithm

An alternate approach to the SGCT is to form the hierarchical surpluses on each component grid. This can be done in-place, as observed in Section 2.1 and this can occur in parallel over each component grid’s process grid \( P \). Then, for each surplus \( h \in N^d \), where \( h \leq l \), the (direct) SGCT is performed on each. Only the processes holding part of surplus \( h \) need participate, that is those holding the component grid \( G_i \) where \( i \geq h \). Finally the original component grids can be recovered from the combined surpluses, thus performing an overall SGCT.

Only surpluses common to more than one grid need be combined. For 2D, this is for hierarchy \( h \) satisfying:

\[
H2(x, y) : h_x + h_y \leq l_x + l_y - 1
\]

This is illustrated in Fig. 7. For 3D, the condition becomes:

\[
\Sigma(h) \leq \Sigma(l) - 2l + 1 \land H2(x, y) \land H2(x, z) \land H2(y, z)
\]

As the number of hierarchies is \( O((\Pi(l)) \), applying the SGCT on them individually is likely to suffer from high startup overheads.

In this section, we describe a parallel algorithm for forming the surpluses (and recovering the original grid from its surpluses). This is followed by a description of how to coalesce these hierarchies, in order to try to reduce startup overheads, and how to schedule them where possible in parallel. We then describe how to overcome an issue with this approach, that is how to handle the case where a hierarchy becomes too small for a process grid \( P > 1 \), noting hierarchy \( h \) is of size \( 2^{h-1} \) for \( h > 0 \). Our solution is called process sub-gridding. Finally, we present how the direct SGCT algorithm was extended to operate on coalesced surpluses.

### 3.2.1. Formation of hierarchical surpluses

Let \( p \in N^d \) denote the process grid over which component grid \( G_i \) is distributed and \( N \in N^d \) be the global size of this grid \( (N = 2^i + 1) \). Algorithm 5 gives the corresponding parallel algorithm for the 3D case (which can easily be generalized for higher dimensions). The notation \( N \ll X \in N^d \) denotes a vector with the same elements as \( N \) in all dimensions except \( d \), where it has a value of \( X \in N \).

![Fig. 6. Partial Sparse Grid for level \( l = 3 \) SGCT onto grid \( (4, 4, 4) \), shown in slices across the \( z \) plane. ‘Filled in’ points (and planes) are shown in green, with points from the sparse grid \( (4, 4, 4) \) in black. The compression factor \( c \) indicates the actual distance between neighbouring \( x \)-points on the edges. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)](image)

![Fig. 7. Coalesced hierarchical index spaces for a 2D truncated SGCT with \( l = 5 \) and \( r = (9, 9) \). Indices of 0 are not shown. The corresponding indices of the component grids are filled with yellow. Indices to surpluses common to only one grid are in dashed boxes. Note that the size of the corresponding surplus halves as an index is decreased by 1. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)](image)
First, we calculate the global left and rightmost global indices of points on process $p$ (line 1). We iterate over each dimension (line 2) and then over each stride length of the hierarchies of that dimension (line 3). Note that the index for the hierarchy is given by $h = l_y - s$. Then, we calculate the global indices of the leftmost and rightmost source points for this surplus (lines 4–5). With this, we calculate the surpluses for local points which do not need points from neighbouring processes (lines 6–7). “Plane $j$ of $u$” should be read as “plane of global index $j$ in dimension $d$ of $u$”. Note that the local index on $p$ is given simply by $j - N_d$. If $p$ owns the leftmost source point $j$ and another process on the left holds the next destination point $k$, we send plane $j$ to that process, and similarly for $j'$ (lines 8–10). If $p$ owns $k$ then we receive the plane $k^1 - 2^k$ from the owning process on the left and plane $k^1$ can now be hierarchized, and similarly for $k'$ (lines 13–18).

Algorithm 5. Hierarchization of grid $u$ of global size $N = 2^d + 1$ by process $p$ in process grid $P$.

1. $N^l = g_0(N, p, P)$, $N^r = N^l + l(N, p, P) - 1$;
2. for $d = x, y, z$ do
3.   for $s = 0 : l_d - 1$ do
4.     $j^l = N_d + \min[x \geq 0 \mid (N_d + x) \mod 2^s + 1 = 0]$
5.     $j^r = N_d + \min[x \geq 0 \mid (N_d + x) \mod 2^s + 1 = 0]$
6.     for $j = j^l + 2^d : j^r - 2^d : 2$ do
7.       hierarchize plane $j$ of $u$
8.       $k^l = j^l - 2^d : k^r = j^r + 2^d$
9.     If $0 \leq k^l \leq N^l_d \leq k^r \leq N^r_d$ then
10.      send plane $j^l$ of $u$ to $p(N^l_d, k^l, k^r)$
11.     if $N^l_d \leq j^l \leq k^l < k^r \leq N^r_d$ then
12.      send plane $j^r$ of $u$ to $p(N^r_d, k^r, k^l)$
13.     if $N^l_d \leq j^l \leq k^l \leq N^r_d$ then
14.      receive plane from $p(N^l_d, k^l, 2^d)$, $(N, P)$
15.     if $N^l_d \leq k^l \leq k^r \leq N^r_d$ then
16.      receive plane from $p(N^l_d, k^r, 2^d)$, $(N, P)$
17.     if $N^l_d \leq k^l \leq N^r_d$ then
18.      hierarchize plane $k^1$ of $u$
19.     end for
20.   end for
To restore the original grid from the surpluses, we simply reverse the order of the stride loop (line 4), and add instead of subtract the averages of the source points (line 7, 15 and 18).

3.2.2. Scheduling and coalescing of surpluses

A set of hierarchies may be coalesced without introducing redundant communications if present on a common set of component grids. This is illustrated for a 2D example in Fig. 7, which shows the index spaces for each surplus can be coalesced under this principle. The component grids are indicated in yellow; indices on the upper boundaries (dotted) are common to only one grid and need not be combined.

The surpluses may be combined in the following order. Surpluses $(k^l - l + 1, k^r - 1)$, ..., $(k^l - l, k^r - l + 1)$ are combined (separately) first, then those of the next diagonal in the triangular area, until finally surplus $(k^l - l + 1, k^r - l + 1)$. It should be noted that each surplus in this triangular area operates on different sets of component grids, and so may not be coalesced with another (without causing redundant communications). Then we combine the surpluses coalesced in one dimension, $(0 : k^l - l, k^r - 1)$, ..., $(0 : k^r - l, 0 : k^r - l - 1)$ and $(k^r - 1, 0 : k^r - l)$, ..., $(k^l - 1, 0 : k^l - l)$, and finally those coalesced in both $(0 : k^l - l, 0 : k^r - l)$. This coalescing is possible as they reside on common sets of the component grids.

While the avoidance of communicating the surpluses common to only one grid $((k^l - l + 1, k^r - 1)$, ..., $(k^l - l, k^r - l + 1))$ is an advantage of this approach, it has a drawback in that there is limited parallelism over the whole system. This can be minimized if surpluses $i$ along each diagonal in the triangular area at a distance $\delta(i) = l^i + l'i - l + 2 - (l_i + l'_i)$ apart are scheduled in parallel, as they share no common component grids. For example, in Fig. 7, along the diagonal beginning with $(8, 5)$, surpluses marked ‘1’ may run in parallel, as may those marked ‘2’. Similarly, for the surpluses shared in one dimension, if these are scheduled from biggest to smallest, i.e., $(0 : k^l - l, k^r - 1)$, $(k^l - 1, 0 : k^l - l - 1)$, $(0 : k^r - l, k^r - 2)$, $(i' - 2, 0 : i' - l)$, ..., then the first $\lceil l/2 \rceil$ of these pairs operate on disjoint component grids and hence will run in parallel. This is illustrated in Fig. 7 with the coalesced surpluses marked ‘3’.

It should be noted that (coalesced) surpluses need to have the same size if they are to be effectively scheduled together.

For 3D, the generalization involves the combination of $(l - 1)(l - 2)(l - 3)/3, 3(l - 1)(l - 2)/2, 3l$ and 1 groups coalesced in 0, 1, 2, and 3 (respectively) dimensions. The scheduling distance $\delta(i)$ increases by 1. This, combined with the fact that the average diagonal is shorter, means that the average scope of parallelization of coalesced surpluses markedly diminishes from the 2D case.

3.2.3. Extension of direct algorithm to support surpluses

We now consider how the SGCT algorithm can be applied to the hierarchical support $h \in N^d$, for which the Boolean vector $\tilde{c} \in \{0, 1\}^d$ indicates that it is not coalesced with all lower indices in each dimension.

Let the current process hold part of grid $G_i$ of size $N = 2^d + 1$, with an associated process grid $P_i$ and this process has $id$ on this grid. If $h \neq 0$ does not hold, support $h$ is not in $G_i$ and the process participates no further. Otherwise, the process proceeds with the corresponding grid index for the SGCT given by $h = h - \tilde{c}$.

The (local portion of the) surplus must first be extracted. The relevant parameters of are the stride $s$, global index offset $0$, local index offset $0$ and the local length $n$:

$s = 2^{l - h - \tilde{c}}$

$O = \tilde{c} \cdot s / 2$

$a = \min[\nu \geq 0 \mid [a_0(N, p, P) - O + \nu] \mod s \geq 0]$

$n = (l(N, p, P) - a + s - 1) / s + C_c(p, P - 1) + \tilde{c}$

The last term for $n$ represents a padding required for an uncoalesced surplus, since these have a global size which is exactly a power of 2. We can also compute $n$ by

$n = l(N, p, P), \quad \text{where } N = 2^{l - h + 1}$

The elements of the extracted surplus may be now supplied to the direct SGCT algorithm, which has $h' (N')$ set to both the component and combined grid index (size) for the current combination.

The SGCT algorithm is also set into hierarchical mode. In this mode, computations over all the original component grids must skip any grid not containing the hierarchy $h'$. Here, the grid ratio $r = 1$ and we send $dn$ instead of $dn + 1$ points (line 9 of Algorithm 1), as interpolation and hence boundary points are not required. Finally, the combination process grid $P'$ in this case may only involve processes participating in the current combination, which will be a subset of the processes operating on all component grids.

3.2.4. Process sub-gridding

Even with the coalescing of hierarchical surpluses, a component grid $G_i$ that is distributed of a process grid $P$ can have a hierarchy $h$ that is, in some dimensions, smaller than the process grid. In this case, the regular block distribution breaks down and the direct SGCT algorithm cannot be applied. Fig. 8 illustrates this case.
where $P = (8, 8)$. It can be observed however that here we still have a sparse block distribution (see Eqs. (9) and (10)), and that, if we take a sub-grid of $P$ beginning at process offset $(0, 2)$ and stride $(2, 4)$, the hierarchy will retain a block distribution over that sub-grid. An exception occurs for the rightmost elements of the hierarchy, where the process stride is 1.

Our solution to this problem is to form such a sub-grid and allow only the processes that are part of it to participate in the SGCT. The following calculations determine the process offset, stride and ‘zero delta’ size and of the process sub-grid:

$$
\begin{align*}
& s = l(C_b(P, 2^b), 1, l(C_b(h, 0), P, P/h)) \\
& O = \tilde{c} + s/2 \\
& Z = (1 - \tilde{c}) \wedge (s > 1) \\
& P' = \min(P, 2^b + C_b(h, 0) * Z)
\end{align*}
$$

Algorithm 6 can be used to determine if a process of $P$ is in the sub-grid and, if so, its id within that sub-grid.

**Algorithm 6.** Determination of sub-grid process ids.

1. $q = r$; // $r$ is the 1D rank of this process in $P$
2. for $d = x, y, z$ do
3. \[p_d' = (q - O_d) \mod P_d; \delta = \left\{ \begin{array}{ll}
Z_d & \text{if } p_d' = P_d - 1 \\
0 & \text{otherwise}
\end{array} \right. \]
4. if $(p_d' + \delta) \mod s_d = 0$ /*is part of sub-grid*/ then
5. \[p_d'' = (p_d' + \delta)/s_d; q = q/P_d; \]

3.3. Limitations and extensions

As noted at the end of Section 3.1.1, the elements on sparse grid process grid ($P'$) currently need to be powers of 2 for the direct algorithm. This can be relaxed if the halo points of $u$ have also been filled in the negative direction, and points $i-1:j+dn$ are sent at line 7 of Algorithm 1 and $n+2$ points are received on line 10 of Algorithm 2 and included in the interpolation (line 11).

This limitation arises due to the fact that, if $P'$ is a power of 2, if it can be guaranteed, for each process $P'$ of $P$, $0 \leq P' < P$, that $g_0(N, P', P) \mod r = 0$ and hence the negative direction halo points will not be needed for the interpolation. This cannot be guaranteed if $P'$ is not a power of 2. The extra cost of sending the negative direction halo points is that $n + 2$ instead of $n + 1$ points must be sent to process $P'$ during the SGCT.

As noted in Section 3.1.3, our implementation of the sparse grid data structure in turn requires $P_0$ to be a power of 2 (and $P'$ to also be a power of 2 for the 3D SGCT). However, one of the dimensions of $P'$ may be a non-power of 2. This is still sufficient to permit the vast majority of processes in the component grids $P_1$ to be made available for $P'$, facilitating near-maximal parallelization of the SGCT.

The hierarchical surplus algorithm however requires all component process grids sizes ($P$) to be powers of 2. This is because the surpluses have a power of two stride (see Fig. 4); thus the elements of the surpluses will only retain a block distribution (Eqs. (5)–(8)) in this case. This could be overcome at the expense of redistribution.

Our current implementation can accommodate 2D or 3D combinations. The main engineering issues in extending it to a higher dimension $d$ are (i) for a given level $l$, the number of sub-grids increases rapidly with $d$ (so load balance in the underlying SGCT application will become an increasing concern), (ii) enumerating the grid indices for these sub-grids across the hyperplanes in grid index space becomes increasingly complex and (iii) the sparse grid data structure becomes increasingly complex. For the hierarchical surplus algorithm, the enumeration of the coalesced surpluses subsquares (Section 3.2.2) also becomes increasingly complex.

As mentioned in [7], the implementation of our algorithms supports a 2D or 3D SGCT on 4D or higher-dimensional grids. A lower dimension (say of size $b$) than those used for the SGCT is supported by extending the implementation to handle blocks of $b$ elements. Higher dimensions can be dealt with by applying the SGCT iteratively. The dimensions chosen for the SGCT must be contiguous, and a local transposition can be applied in order to ensure this.

Our direct SGCT algorithm supports parallelization over arbitrary process grids ($P_i$) in the SGCT dimensions of a data grid. As also explained in [7], parallelization in a non-SGCT dimension can be simply performed by applying the whole SGCT in parallel over this dimension.

4. Analysis

We now give an analysis of our SGCT algorithms under typical operating conditions imposed it with conjunction with an application [12, 7].

In order to perform a SGCT over the desired set of sub-grids (u), where $u_i$ is distributed over the process grid $P_i$, we form the combined process grid $P$ over a subset of these processes. That is, $\Pi(P') = 2^k$ for the largest $k$ such that $2^k \leq \Sigma_i P_i$. The aspect ratio of $P$ will be set to match the grid index of the combined (full) grid. In this case $P \geq P_i$. For simplicity (and because it is a case of strong practical interest [12, 7]), we assume each $P_i$ is a power of 2 also.

In terms of load balancing, we allocate the same number (p e $\mathbb{N}$) of processes on each of the $P_i$ on each diagonal in the grid index space (see Fig. 2); in the 3D case, the diagonal becomes a plane. The next lower diagonal receives $p/2$ processes. This strategy balances the amount of data points and hence work across each processes, which approximates to a first order the load for that process.

We denote $g = g(d, l)$ to be the number of sub-grids given dimensionality $d$ and level $l$. We have $g(d, l) = O(d^{l-1})$ and more specifically:

$$
\begin{align*}
g(2, l) &= 2l - 1 \\
g(3, l) &= 3l^2 - 2l + 1
\end{align*}
$$

Let $m$ denote the number of data points in a sub-grid in a given process. By the above assumptions $m$ should be constant across all processes. Let $\alpha$ and $\beta$ denote the cost of starting up a message and the transmission cost per unit data.
4.1. Direct SGCT algorithm

For this algorithm, in the worst case, each process in P will receive 2m words of data. Each process in each P sends and receives \( \Pi(P/P_i) \leq g \) messages, and each process in P receives and then sends g messages. Since the whole operation should be completely parallelized, the total cost of the direct gather-scatter is given by:

\[
t^E \leq 2g\alpha + 3mb
\]

(13)

This indicates that the algorithm should be efficient in bandwidth-constrained (large m) circumstances, which will be the case for most large-scale simulations. Otherwise, when g is large, it may be less efficient compared to an algorithm based on multistage reductions with a constant fan-in [13], with a cost of \( 2 \log_2(g(\alpha + mb)) \).

4.2. Interpolation overheads

Consider a level l SGCT on a d-dimensional isotropic grid of grid index \( i^l \in \mathbb{N}, i^l \geq l \). The full grid will have \( O(2^d) \) points. The partial sparse grid will have a fill in factor of \( f(d, l) = 2^d - l \) and hence will have

\[
f(d, l)2^d O(l^d - 1) = 2^d - (d - 1)l^d - 1
\]

data points, noting a classical sparse grid \( (i^l = l) \) has \( 2^d \) points. Each component grid will have \( 2^d - (d - 1)l^d - 1 \) points.

Assume the upper level grids are distributed across p processors, and partial sparse grid is distributed over the total number of processors, \( g(d, l)p = O(l^d) \), using Eqs. (11) and (12).

Then the ratio of partial sparse grid to component grid points per process is given by:

\[
\frac{2^d - (d - 1)l^d - 1}{2^d - (d - 1)l^d - 1} = O(1)
\]

This indicates that the interpolation stage should scale with p, l, and d. However, the number of interpolation calculations per received point is \( O(l^d) \) and hence, for sufficiently large l and d, interpolation time will dominate communication time.

By contrast, if a full grid is used for interpolation, the ratio of full grid to component grid points per process is

\[
\frac{2^d}{2^d - (d - 1)l^d - 1} = O(l^d - 1)
\]

which is essentially exponential in l.

4.3. Hierarchical surplus-based algorithm

This approach avoids communication of surpluses shared by only one grid, which reduces the total communication volume by a fraction \( \frac{1}{2^d} \). This is however offset by both the limited parallelization and load imbalance imposed. Furthermore, even with coalescing, the overhead of communication start-ups will be considerably increased over the direct method. This will however be partially offset by the average effective value of g being \( \approx 30\% \) on average less. For example, the direct algorithm should perform fairly optimally, given Eq. (13), on the largest surpluses, which have a minimal g = 3.

Roughly the load imbalance factor will be \( \approx 2^d - 1 \), due to the fact that process grids on the inner diagonal (see the highlighted areas of Fig. 7) have half the number of processes, but the sizes of the surpluses is the same across all grids. The average degree of parallelization will be \( \approx 2^d \) for the scenario of Fig. 7. Consider the largest surpluses, labelled ‘1’ and ‘2’. Those labelled ‘2’ cannot run in parallel with those marked ‘1’. Those marked ‘3’ can run in parallel, but then only \( \approx 2^d \) of the processes can be utilized. As we move closer to the origin, the overall utilization increases, till it is 1 when the indices are less than \( i^l = l + 1 \) (=5, 5) in the figure). However, these areas represent only a small fraction of the overall data. A more precise analysis needs to be done empirically. Included in the code release for this work is a Python script calculating the overall degree of parallelization for the combination of all surpluses. This generated the data for the following table, for the 2D case \( i^l = (9, 9) \) with coalesced surpluses:

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

This indicates that scheduling should make a worthwhile difference, achieving a 70–80% degree of parallelization. An experiment where the grid size grows with \( l \), i.e., \( i^l = (9 + l, 9 + l) \), gave essentially identical results.

Taking all these factors together, it is unclear if this approach, even with scheduling, has any overall inherent advantage in speed even for \( d = 2 \).

5. Experimental results

In this section, we give performance results for our algorithms, under the operating conditions mentioned in Section 4.

All experiments were conducted on the Raijin cluster located at the Australian National University. It has 3592 compute nodes, consisting of dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors (i.e., 16 cores) with on average 44 GB memory per node, and with an Infiniband FDR interconnect (56 Gbps). Version 1.43 of Open MPI was used and our codes were compiled with g++ 4.6.4 with the -O3 option.

5.1. Direct SGCT performance

In order to demonstrate the effect of our parallel SGCT algorithm on an application, we use an advection solver capable of solving 2D and 3D problems. As the amount of work per timestep for the advection problem is relatively small compared to most real applications, any inefficiencies in the SGCT will be easily seen.

The 2D solver solves the linear hyperbolic PDE for a field \( u(t, x, y) \):

\[
\frac{\partial u}{\partial t} = -v \cdot \nabla u
\]

over the unit square with periodic boundary conditions \( u(t, 1, y) = u(t, 0, y) \) and \( u(t, x, 1) = u(t, x, 0) \) and the initial condition of \( u(0, x, y) = \sin(2\pi(x + 2y)) \). We discretize this with the Lax-Wendroff stencil [15]. Linear basis functions are used in conjunction with the SGCT. The 3D solver is similar.

Fig. 9 and Table 1 show strong scaling results. The plots BSend indicate an MPI BSed based implementation was used: this imposes a fixed order of message receipt. The plots ISend indicate that an MPI ISend/IRrecv implementation was used, which does not have this limitation and allows the overlap of communication with the interpolation of the gather stage. For the 3D problem, results for small core counts are missing for the former, as the MPI implementation was not able to handle the very large BSend buffers that were required.

The results indicate that the application scales even with the SGCT applied relatively frequently, with the ISend implementation lending \( \approx 20\% \) extra performance to the overall execution time.

Fig. 10 indicates the weak scaling performance of the direct 2D SGCT in isolation using the partial sparse grid data structure (the default). We chose \( 2^{14} \) points per process as this represents a
Fig. 9. Overall execution time of the advection application with the direct SGCT running over 1024 timesteps (MPI warm-up time excluded). Results shown are an average of two experiments.

Table 1

Tabular form of ISend plots of Fig. 9. Rows indicate execution time (s) for various numbers of combinations applied.

<table>
<thead>
<tr>
<th>Cores</th>
<th>44</th>
<th>88</th>
<th>176</th>
<th>352</th>
<th>704</th>
<th>1408</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) 2D problem, $l=4, 2^{13} \times 2^{13}$ points</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>12.50</td>
<td>6.44</td>
<td>3.09</td>
<td>1.09</td>
<td>0.51</td>
<td>0.31</td>
</tr>
<tr>
<td>4x</td>
<td>12.81</td>
<td>6.59</td>
<td>3.17</td>
<td>1.13</td>
<td>0.52</td>
<td>0.28</td>
</tr>
<tr>
<td>16x</td>
<td>14.63</td>
<td>7.54</td>
<td>3.73</td>
<td>1.38</td>
<td>0.66</td>
<td>-</td>
</tr>
<tr>
<td>64x</td>
<td>21.47</td>
<td>11.51</td>
<td>5.80</td>
<td>2.43</td>
<td>1.10</td>
<td>0.55</td>
</tr>
<tr>
<td>Cores</td>
<td>31</td>
<td>62</td>
<td>124</td>
<td>248</td>
<td>496</td>
<td>992</td>
</tr>
<tr>
<td>(b) 3D problem, $l=3, 2^9 \times 2^9 \times 2^8$ points</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1x</td>
<td>19.64</td>
<td>10.16</td>
<td>5.29</td>
<td>2.65</td>
<td>1.37</td>
<td>0.72</td>
</tr>
<tr>
<td>4x</td>
<td>20.28</td>
<td>10.42</td>
<td>5.43</td>
<td>2.77</td>
<td>1.43</td>
<td>0.75</td>
</tr>
<tr>
<td>16x</td>
<td>23.75</td>
<td>12.07</td>
<td>6.32</td>
<td>3.27</td>
<td>1.61</td>
<td>0.86</td>
</tr>
<tr>
<td>64x</td>
<td>35.65</td>
<td>17.98</td>
<td>9.57</td>
<td>4.74</td>
<td>2.47</td>
<td>1.31</td>
</tr>
</tbody>
</table>

reasonable amount of data, and is sufficiently large so that the band-width components of the SGCT (see Eq. (13)) becomes significant. It indicates that once communication is predominantly inter-node, the execution time remains relatively flat, indicating good scaling performance to the limit of our tests at around 3000 cores.

It can be noted that execution time increases slowly with increasing $l$ when the sparse grid data structure is used (Fig. 10(b)). At $l=3$ and $l=11$, the total core counts are an exact power of two; this is the best case for the algorithm, as all cores can hold part of the partial sparse grid. We have some performance loss at $l=5, l=7$

Fig. 10. Execution time of the average of 10 repetitions of the 2D direct SGCT in isolation (MPI warm-up time excluded). The workload of each core is $2^{14}$ grid points. ‘gs’, ‘in’ denote the level of the SGCT, combination time, and the time on the interpolation step only, respectively. ISend-based implementations are used. Results shown are an average of two experiments.
and \( l = 9 \) where only 80%, 76% and 88% of the cores hold part of the sparse grid. There is approximately a factor of 2 between the performance at \( l = 3 \) and \( l = 11 \) cases. Much of this difference is due to the interpolation time (‘in’), whose proportion of overall execution time increases with \( l \). This is largely due to load imbalance effects: the sparser the (partial) sparse grid becomes, the less evenly the points divide over a regular process grid. As can be seen from Fig. 1, the processes at the corners may have significantly more points than their neighbours towards the interior.

Fig. 10(a) shows the results if the much simpler full grid data structure is used for interpolation. This introduces a factor of \( \approx 2^{l-1} \) redundant data points. As expected, we get an exponential increase in execution time with respect to \( l \) with interpolation time quickly accounting for almost all execution time. It indicates that the sparse grid data structure is essential for good scaling with \( l \).

While interpolation appears to account for much of the total time, it should be noted that message receipt is overlapped with it, hence message receipt time is attributed to interpolation time. Results corresponding to Fig. 10(a) for the ISend implementation indicate that interpolation never consumes >50% of execution time. They also show poorer scaling with core count than the ISend results. Other experiments indicate the partial sparse grid interpolation function achieves near-ideal thread-level parallelism.

Fig. 11 gives a comparison of the 2D and 3D algorithms using the sparse grid data structure. In terms of the number of component grids, \( l = 3, 5, 7 \) in 3D is equivalent to the cases \( l = 5, 17, 31 \) in 2D. We can see that 2D and 3D performance is very similar for \( l = 3 \) (3D) and \( l = 5 \) (2D), and 3D scales similarly well with increasing core count. We also see a gradual decrease in performance as the number of component grids increases becomes large, due to the increased effects of startup latency.

All data shown so far are for ‘warmed’ timings where an SGCT was performed before the timing was taken. Fig. 11 shows the time for the direct SGCT without this warmup; we see a degradation in performance by a factor of approximately 20. This is caused by OpenMPI setting up new (inter-grid) connections between processes in order to perform the SGCT. While not part of the SGCT itself, this overhead needs to be taken into account by any application using the SGCT. The results of our previous work [16] do not take this effect into account.

5.2. Hierarchical SGCT performance

Fig. 12 indicates the performance of the hierarchical surplus based algorithms. A comparison of sub-figures (a) to (b) and (c) to (d) indicates the clear benefit of coalescing the surpluses. Comparing sub-figures (a) and (c) to Fig. 11, we see that this version is slightly slower for smaller \( l \) and core counts. We also see that it scales less well with \( l \), due to the fact there are \( O(l^2) \) sub-operations and the number of cores (especially as \( g(l) \) increases). Somewhat surprisingly, scheduling of the surpluses gave no clear overall benefit.

The forming and restoring from the hierarchical surpluses formed a small factor (\( \approx 1/10 \)) of this. As expected, the results vary little with \( l \) for 2D, but show some increase with \( l \) for 3D.

It should also be noted that there are no other SGCT codes available that can be used to perform the above experiments for comparison.

5.3. Load balance and communication profiles

Fig. 13 gives IPM profiles [17] for the 2D advection application with a \( l = 11 \) SGCT. We can clearly see the structure of the advection sub-problems. MPI_Send and MPI_Recv are used only in the advection phase, being used for halo exchanges. The fluctuations in MPI_Waitany in part (a) indicates (inversely) load imbalance across the process grid caused by differences in computational speed across the different sets of advection problems. It is also the dominant communication time for the SGCT operation (part(b)).

5.4. Effect of process grid aspect ratio and process layout on performance

The aspect ratios of each component grid and its process grid determines the shape of the decomposed domain. Changing the decomposed domain shape and changing how processes are aid out across nodes changes the communication pattern for both the computation of application on the grid and the SGCT. Each application has its own communication requirements, and so process grid aspect ratio and layout may affect them differently. We analyze the effect for the 2D advection application with different process layouts.
Fig. 12. Execution time of the average of 10 repetitions of the hierarchical SGCT in isolation (MPI warm-up time excluded). The workload of each core is $2^{15}$ grid points. 'l', 'gs', and 'hc' denote the level of the SGCT, combination time, and the time to form and restore from the hierarchical surpluses, respectively. As the scheduling of hierarchical surpluses does not offer significant improvement, those are not included in the graph. ISend-based implementations are used. Results shown are an average of two experiments.

The default SGCT process grid layout is made to match the shape of the component grid $G_i$. Allocating 64 processes to the largest component grids, the default process layout for $2D G_i$ with $l= (3, 0), (2, 1), (1, 2), (0, 3), (2, 0), (1, 1),$ and $(0, 2)$ are $16 \times 4, 8 \times 8, 8 \times 8, 4 \times 16, 8 \times 4, 8 \times 4,$ and $4 \times 8$, respectively. That of the 2D full grid is $16 \times 16$.

How process are mapped to nodes also affects communication. The process-to-node mapping of all the previous benchmarks were done by mapping the processes to cores in nodes linearly (default), as shown in Fig. 14a for a 2D process grid. This will minimize inter-node communication horizontally at the expense of inter-node communication in the vertical dimension. A block-wise mapping, such as shown in Fig. 14b, will balance inter-node communication in both dimensions.

The effect of process aspect ratio and layout for a 2D advection application is presented in Tables 2 and 3 in isolation. It is observed from Table 3 that $8 \times 8, 8 \times 4$ process grid with block map shows a small advantage in advection computation time. Applying
the same configuration also improves the combination time slightly as shown in Table 3.

6. Related work

Early work in the parallelization of the SGCT for Laplace’s equation and the 3D Navier–Stokes system was reported in [2,4], respectively. However, the algorithm assumed each node had multiple sub-grids and concentrated on load balancing aspects, whereas our algorithm is designed for the many processes per sub-grid case.

More recently, work on a parallel SGCT algorithm has also involved allocating multiple grids per process group of size limited to the number of cores in a socket, with OpenMP parallelism applied within each process group [11]. The algorithm was targeted towards higher dimensional problems and the concern was in load balancing within sub-groups of O(1000) grids.

Work on load-balancing of component grid instances for the SGCT and a parallel SGCT algorithm using both the direct and hierarchical surplus implementations has been reported in [18,13], respectively. The latter paper also gives an analysis of their algorithms’ communication overheads. However, it is assumed that there is only one process per component sub-grid (i.e., P2 = 1), severely limiting its applicability to large-scale simulations. Their algorithm also does not support the truncated combination formula. While in both cases, the (direct) algorithm is based on a gather-scatter approach at the top level, our algorithm has only two stages, whereas theirs has O(log_2 J) stages. Furthermore, their version of the direct algorithm involves a significant amount of redundant communications, resulting in their conclusion that the hierarchical surplus approach is 10× (3.5×) faster for a high SGCT level (J) for 2D (3D) combinations. For our algorithms, we reach the opposite conclusion. The paper mentions the idea of “merging” (coalescing?) hierarchical surpluses only in the context of future work.

Efficient data structures for sparse grids usually rely on tree or hash methods [19,20]. More recently, a more compact representation involving multi-level indexing of a flat array, ranged in the order of the hierarchical surpluses [21]. It also offers faster indexing. Our sparse grid data structure based on sparse matrix ideas shares these advantages and moreover has potentially better data locality, as the flat array is based on the spatial layout of the original component grid. Furthermore, our implementation is more general than any, as it is fully distributed and covers partial sparse grids as well.

Ref. [12] used our 2D direct SGCT implementation to build a fault-tolerant 2D advection solver. As its emphasis was on fault-tolerance it gave no details of the SGCT algorithm nor did it examine its performance in isolation. Similarly for [14], which studied the computational complexity of the parallel SGCT and described software engineering techniques to reduce the complexity and ensure the reliability of the 2D direct SGCT implementation. Indeed, these techniques proved even more crucial to the more complex 3D and hierarchical surplus based implementation of the algorithms presented here. However, these papers give no details about the parallel SGCT algorithm itself.

This paper is an extended version of our preliminary work [16]. The algorithms have been perfected, with details for 3D added, and we have introduced the sparse block distribution, the partial sparse grid data structure and the process sub-gridding technique. The analysis is more comprehensive and more precise. The experimental results are all significantly improved and extended results have been included.

7. Conclusions

In this paper, we have presented two general SGCT combination algorithms both supporting parallelism within and across process grids and the more general SGCT ‘truncated combination’ formula. They can also support fault-tolerant SGCT combination formulae.

---

Table 2

<table>
<thead>
<tr>
<th>Combination</th>
<th>SGCT default</th>
<th>8 × 8, 8 × 4</th>
<th>Block map</th>
<th>8 × 8, 8 × 4 and block map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>4.34</td>
<td>4.35</td>
<td>4.32</td>
<td>4.28</td>
</tr>
<tr>
<td>Maximum</td>
<td>4.49</td>
<td>4.55</td>
<td>4.44</td>
<td>4.34</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>Combination</th>
<th>SGCT default</th>
<th>8 × 8, 8 × 4</th>
<th>Block map</th>
<th>8 × 8, 8 × 4 and block map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
<td>2.04E–02</td>
<td>2.03E–02</td>
<td>2.02E–02</td>
<td>2.02E–02</td>
</tr>
<tr>
<td>Maximum</td>
<td>2.20E–02</td>
<td>2.23E–02</td>
<td>2.18E–02</td>
<td>2.17E–02</td>
</tr>
</tbody>
</table>
The algorithms are inherently complex but can be formulated (and implemented) tractably when expressed in terms of d-dimensional vector arithmetic and the associated mappings for a block distribution. The direct algorithm is highly efficient in bandwidth-limited scenarios and is general in terms of the process grid configurations of the component grids. It is particularly suited to SGCT where the number of component grids is small to moderate, i.e., \( d \leq 5 \). It does however need a partial sparse grid data structure, a generalization of the sparse grid, to be scalable with the SGCT level and the dimensionality \( d \). Our distributed implementation of this data structure, based on sparse matrix concepts, is comparably efficient to existing non-distributed implementations, in terms of both space and access time, and scales in the same sense as the SGCT.

The hierarchical surplus based algorithm uses the direct algorithm for combining the individual and coalesced hierarchical surpluses. It thus inherits some of the performance characteristics of the direct algorithm. It offers a bandwidth reduction by a factor of \( 1 - 2^{-d} \); however, this seems to be offset by an incomplete parallelization factor \( \geq 2/3 \) for 2D (and even this requires careful scheduling of the computation), a factor of \( \approx 2^{-d} \) load imbalance (arising from the uneven number of processes across the component grids) and overall increased startup times. The latter is reduced by our technique of coalescing the surpluses for combination. It appears to have an advantage in memory consumption, with the extra memory for the SGCT being reduced to the size of the largest common surpluses, which is a factor of \( 2^{-d} \) of the largest component grids. However, the direct algorithm can match this by partitioning the SGCT into a similar number of stages.

Our algorithm for hierarchical surplus formation is much less communication-intensive. It hoes however show weaker scalability than the 4SGCT over increasing core counts.

Experimental results with 2D and 3D advection solvers indicate that, even with a relatively frequent application of the direct SGCT algorithm, the application will scale to at least 1500 cores. The MPI Isend based implementation, imposing no order on message receipt, gave worthwhile benefits of up to 15% for the whole application. A weak scalability analysis shows that the execution time remains relatively flat, indicating good scaling performance to limit of our tests at around 3000 cores. The hierarchical algorithm requires coalescing of the surpluses for satisfactory performance, and, surprisingly, scheduling the computation creates no appreciable benefit. We find that increasing the SGCT level increased execution time only by a factor of 2 from \( l^3 \) to \( l^11 \) for the direct algorithm. The hierarchical algorithm scaled slightly less well in this respect, due to increased startup costs.

Future work will can still improve the direct algorithm by overcoming the limitation of a power of two process grid for the combined grid. This can provide a potential increase of performance by a factor of up to 2.


Acknowledgements

The authors thank Markus Hegland and Jay Larson for valuable advice and support. We also thank the NCI National Facility for the use of the Raijin cluster. This research was supported under the Australian Research Council’s Linkage Projects funding scheme (project number LP110200410).

References

Mohsin Ali is a PhD student in the Computer Systems Group at the Research School of Computer Science, at The Australian National University. He came to Australia from Bangladesh in 2012. He has done his Masters from Canada and Undergraduate from Bangladesh both are majoring in Computer Science. His research interests include high-performance computing, algorithms, mobile computing, and computer networks.

Brendan Harding completed a PhD at the Mathematical Science Institute at The Australian National University in 2016 and is currently a postdoctoral research fellow. His research interests include numerical analysis, high performance computing and fractal geometry.