Building Input Adaptive Parallel Applications: A Case Study of Sparse Grid Interpolation

Alin Muraras\u0103
Technische Universität München
Email: murarasu@in.tum.de

Josef Weidendorfer
Technische Universität München
Email: weidendo@in.tum.de

Abstract—The well-known power wall resulting in multi-cores requires special techniques for speeding up applications. In this sense, parallelization plays a crucial role. Besides standard serial optimizations, techniques such as input specialization can also bring a substantial contribution to the speedup. By identifying common patterns in the input data, we propose new algorithms for sparse grid interpolation that accelerate the state-of-the-art non-specialized version. Sparse grid interpolation is an inherently hierarchical method of interpolation employed for example in computational steering applications for decompressing high-dimensional simulation data. In this context, improving the speedup is essential for real-time visualization. Using input specialization, we report a speedup of up to 9x over the non-specialized version. The paper covers the steps we took to reach this speedup by means of input adaptivity. Our algorithms will be integrated in \textit{fastsg}, a library for fast sparse grid interpolation.

I. INTRODUCTION

In order to obtain performance on multi-cores, parallelization is mandatory. For extra performance, a deep understanding of how the hardware works is needed, including the multi-level memory structure, the vector units, and the instruction pipeline. Furthermore, since the characteristics of the input data often affect the performance behavior of an application, input adaptive optimization strategies are necessary. Adaptation to hardware and input data ensures both flexibility and performance. In simple terms, adaptation to hardware refers to minimizing the execution time (or memory, energy) by choosing the right values for a set of tunable parameters that are dependent on the properties of the underlying hardware. Through adaptation to input data we want to provide performance across multiple inputs. In input adaptation, we also include the exploitation of characteristics of the input data in order to further reduce the execution time.

In this paper we propose an input adaptive solution for sparse grid interpolation which refers to the numerical interpolation of high-dimensional functions represented using the sparse grid technique [1]. This technique is used for solving high-dimensional problems arising in data mining and computational fluid dynamics [1], finance [2], computational steering [3], thus being an alternative to the stochastic methods typically employed for these problems. As opposed to full grids that require $O(N^d)$ points to represent a $d$-dimensional function, sparse grids require only $O(N \cdot (\log(N))^{d-1})$ points, where $N$ is the number of points in each coordinate dimension. Despite requiring orders of magnitude less points, the accuracy is only slightly deteriorated for sufficiently smooth functions [1]. In this paper we focus on computational steering using sparse grids [3]. We can see this scenario in Fig. 1. High-dimensional simulation data is compressed using a sparse grid operation referred to as hierarchization [4]. Afterwards, the simulation data is decompressed for real-time visualization using sparse grid interpolation or evaluation [4].

Sparse grids are different from sparse matrices both in functionality and computational behavior. Sparse grids allow us to approximate $d$-dimensional functions and are typically stored in memory using trees and hash-tables. [4] proposes a bijection based data structure for sparse grids that consumes a minimum amount of memory. The main idea there is to store only the values in the sparse grid in a special order such that the position of a value can be passed to a bijective mapping in order to retrieve the coordinates of the $d$-dimensional point corresponding to that value. In contrast, well-known formats for storing sparse matrices include: Compressed Sparse Row (CSR), Coordinate List (COO), ELLPACK, etc. Since $d$ is in general bigger than 3, these formats are not fit for generic $d$-dimensional sparse grids. Furthermore, additional data is stored for determining the positions of the non-zero values in the sparse matrix. Hence, the formats are also not minimal with respect to memory consumption. Compared to sparse matrix - vector multiplication (SpMV), sparse grid interpolation suffers less from indirect memory accesses and is computationally
bound as opposed to SpMV which is memory bound.

For computational steering, accelerating sparse grid interpolation is crucial for ensuring real-time interaction with the compressed simulation data. Parallelization is a mandatory requirement but in addition to it, we employ input specialization and adaptation. In our computational steering scenario, we have observed occurring patterns in the input data, i.e. interpolation points, for which we specialize sparse grid interpolation in order to decrease the number of floating point and integer operations. In simple terms, we propose more efficient representations for the sets of interpolation points provided as input. Furthermore, by interleaving the interpolations at the different points we can reuse the results of floating point and integer computations. This results in less execution time. As an example, the observation that visualization is limited to 3d results in a speedup of approximately 3x when interpolating a 9d function. Similarly, if the set of interpolation points is generated by means of a Cartesian product, the execution time can also be reduced significantly, e.g. by up to 9x.

Besides addressing the occurring patterns in the input data, we also show the use of search based auto-tuning to handle input dependent optimizations. We show how permuting data in memory is strongly influenced by the number of interpolation points. If this number is not big enough, the permutation can actually decrease the performance. Determining the right value is achieved via empirical search. The proposed input dependent algorithms and optimizations are hidden from the user behind a uniform and simple to use interface for interpolation. Under the hood, decision trees and empirical search are used to ensure effective input adaptivity.

To sum up, the contributions of our paper are the following:

- To the best of our knowledge, we propose the first input adaptive solution for sparse grid interpolation. We describe our algorithms in detail such that they can be easily included in frameworks for interpolation, e.g. spinterp [5], [6].
- We compare the performance on 4 multi-core systems from Intel and AMD. Our input oriented optimizations result in a speedup factor of up to 9x compared to the state-of-the-art sparse grid interpolation from [4]. The OpenMP version scales linearly on all benchmarked multi-cores.

II. RELATED WORK

Techniques for input adaptation are already employed in libraries such as OSKI [7] and FFTW [8]. OSKI is a library for sparse linear algebra that is able to automatically select the best data structure and algorithm given special characteristics of the input matrix: symmetry, block and diagonal structures, etc. FFTW is a library for computing FFT transformations that recursively divides a given problem using a dynamic programming algorithm, in the end reducing it to smaller subproblems solved by the so called “codelets”, i.e. input specialized algorithms for computing FFT. Input adaptive approaches exist also for sorting [9], [10] and reduction [11]. Furthermore, general purpose frameworks have been developed that make use of machine learning techniques to map input data to multiple code versions such as STAPL [12]. A similar framework is also described in [13]. Since we propose an application specific solution for input adaptivity, our work is only similar in philosophy with these approaches. For a more comprehensive list of input adaptive solutions, we refer to [12].

Computational steering using sparse grids is described in [3] for a computational fluid dynamic application. We refer to that paper for an accuracy study. We emphasize that in this paper our focus is extra performance. At the base of our work, we use the iterative interpolation algorithm from [4]. However, we focus on its main building block, i.e. the innermost 2 loops. That is the most time consuming part common for different types of sparse grids including regular [4] and dimensionally adaptive [14], [15]. Standard optimizations, e.g. vectorization, for sparse grid interpolation are described in [16], [17]. However, the authors do not cover input adaptation which we show to be profitable resulting in a 9x speedup.

III. INPUT ADAPTIVE SPARSE GRID INTERPOLATION

Supercomputers have nowadays computation speeds in the range of PFLOPS. When running simulations on these systems, we can easily become overwhelmed by the sheer amount of output data that has to be stored, processed, and visualized. Often, we want real-time visualization and interaction with the simulation data. To reach this goal, a compressed and hierarchical representation of the data is essential. These characteristics allow us to improve the storage space and the data transfer across the network. Furthermore, we can design visualization algorithms with better response times that provide the data incrementally to the user and use multiple levels of details for different parts of the data based on their distance from the viewer or their importance. A numerical technique with such features is the sparse grid technique. The operations of the sparse grid technique used to achieve compression and decompression functionality are

![Fig. 2: Decomposition of a 2d sparse grid in simple dense block structures.](image-url)
Algorithm 1 Sparse grid interpolation core: \( \text{cinterp}(\text{in}: m, d, x[m][d], l[d], b[2[1][1]...[2][d]], \text{in/out}: r[m]) \).

1: for \( k = 1 \) to \( m \) do
2: \( p \leftarrow 1, \text{idx} \leftarrow 0 \)
3: for \( t = 1 \) to \( d \) do
4: \( \text{idx} \leftarrow \text{idx} \cdot 2[t] + \text{floor}(2[t] \cdot x[k][t]) \)
5: end for
6: \( r[k] \leftarrow r[k] + b[\text{idx}] \cdot p \)
7: end for

We emphasize that Alg. 1 is only the representative core of sparse grid interpolation. For a multi-dimensional function, hierarchization returns a set of coefficients, i.e. hierarchical coefficients, which correspond to a set of hierarchical basis functions. Although hierarchization is also worth accelerating, the time critical operation for real-time visualization is interpolation.

Fig. 2 shows an example of a 2d sparse grid and the memory storage scheme following the approach described in [4], [17]. The 2d case is chosen here for its simplicity but the decomposition method is in general applicable to any \( d \)-dimensional sparse grid. We can see that the sparse grid is decomposed as a hierarchy of regular grids or blocks. The blocks contain hierarchical coefficients and have different shapes. In general, handling the dense blocks is more hardware friendly. It is thus important for performance to define the sparse grid operations, in particular interpolation, at block level. This is a key aspect of our work.

Interpolating a sparse grid at a given point is equivalent to computing the sum of multiple \( d \)-dimensional basis functions scaled by multiplying them with some coefficients and applied to the chosen point. Looking at Fig. 2, for each interpolation point and for each block we perform 2 operations: (1) we build a \( d \)-dimensional basis function per block which we evaluate at the given point and (2) we extract exactly one coefficient from the block which we multiply with the already built basis function. We then sum up the results across all blocks and by doing so we obtain the final result of sparse grid interpolation.

In general, the innermost part of sparse grid interpolation can be reduced to Alg. 1. In other words, Alg. 1 is executed per block. Here, \( d \) is the dimensionality, i.e. number of dimensions. \( m \) is the number of interpolation points and \( x \) is an \( m \times d \) matrix containing their coordinates. \( \text{idx} \) is used to select a coefficient from a \( d \)-dimensional block which can be seen as a multi-dimensional array \( b[2[1][1]...[2][d]] \). \( l \) is just an array of integers acting as the unique identifier for \( b \) and providing the shape of \( b \). Although there are multiple variants of expressions for the 1d basis function \( \phi \) from line 4, we use in the rest of the paper the following definition:

\[
\phi_u(x) := \max(1 - |2^u + 1 \cdot x - 2^u \cdot x| \cdot 2 - 1, 0).
\]

Besides the standard optimizations, we propose input specialization as an extra optimization technique based on two observations regarding characteristics commonly found in \( x \):

1) for visualization, the interpolation points are typically 3d (or 2d), i.e. for \( d \)-dimensional data, \( d - 3 \) columns of \( x \) share one value across all rows,
2) often, the interpolation points form a regular grid, i.e.
Algorithm 2 Version for sha. pattern: \textit{cinterp\_sha\textbf{(in:}} m, d, x[m][d], l[d], b[2[d]...2[d]], sd, sel, isel, in/out: r[m]).

\begin{algorithm}
\begin{algorithmic}
\STATE 1: \texttt{psum}[d+1] ← 0, \texttt{psum}[d] ← l[d]
\FOR {i = d → 1 downto 1}
\STATE 3: \texttt{psum}[t] ← \texttt{psum}[t+1] + l[t]
\ENDFOR
\STATE 5: \texttt{sp} ← _, \texttt{sidx} ← _0
\FOR {t = 1 to \texttt{sd}}
\STATE 6: \texttt{sp} ← \texttt{sp} · \texttt{\phi}[[\text{ise}\texttt{l}[t]](x[0][\text{isel}[t]])]
\STATE 8: \texttt{sidx} ← \texttt{sidx} + floor(2[[\text{isel}[t] · x[0][\text{isel}[t]]]) · \texttt{2psum}[\text{isel}[t]+1]
\ENDFOR
\STATE 10: for k = 1 to \texttt{m} do
\STATE 11: p ← \texttt{sp}, \texttt{idx} ← \texttt{sidx}
\STATE 12: for t = 1 to \texttt{d} do
\STATE 13: p ← p · \texttt{\phi}[[\text{isel}[t]](x[k][\text{isel}[t]])]
\STATE 14: \texttt{idx} ← \texttt{idx} + floor(2[[\text{isel}[t] · x[k][\text{isel}[t]]]) · \texttt{2psum}[\text{isel}[t]+1]
\ENDFOR
\STATE 16: r[k] ← r[k] + b[idx] · p
\ENDFOR
\end{algorithmic}
\end{algorithm}

the set of points is the Cartesian product of \textit{d} sets of values, each corresponding to a dimension.

We refer to (1) as \textit{sha pattern} and to (2) as \textit{car pattern}. We will show that we can derive from \textit{car} other patterns of interest, e.g. \textit{3d pattern}. For every pattern we built a specialized algorithm. To decide which algorithm to use, we use the decision tree depicted in Fig. 3a. Nevertheless, Fig. 3b shows that the patterns are not always disjoint as there is input data for which more than two patterns can be applied. For problem sizes not significantly big and still of interest, an algorithm with higher complexity but more hardware friendly can outperform an algorithm with lower complexity but hardware unfriendly. We discuss the occurrence of this situation in Sec. IV.

A. The sha. Pattern

1) Detection: Our patterns refer to characteristics of the matrix \textit{x}. The first step in applying Alg. 2, the specialized algorithm for \textit{sha}, is to detect the pattern. We now briefly describe the procedure. It starts by traversing \textit{x} and checking if there are columns sharing one value across all rows. For columns with this property, their index is saved in the array \textit{sel}. Let \textit{sd} be the number of integers added to \textit{sel}. The indices of the remaining columns are added to the array \textit{isel}. Hence, the complexity of \textit{sha} detection is \textit{O}(m · d). This might seem too much relative to the interpolation core but it is executed once for 100,000 calls to Alg. 2.

2) Description: The benefit of Alg. 2 over Alg. 1 results from a reduction of the number of floating point and integer operations. We can see in Alg. 2 how specializing for \textit{sha} changes Alg. 1. Since there are columns that share one value, we apply loop invariant code motion for a part of the product \textit{p}, i.e. \textit{sp}. Thus, \textit{sp}, corresponding to the \textit{sd} shared dimensions, i.e. columns sharing one value across all rows, moves from the innermost loop in Alg. 1 to line 5 in Alg. 2. A significant number of operations is saved. But to get to \textit{O}(m · (d−sd)) for Alg. 2, part of the \textit{idx} computation must also be moved outside of the innermost loop from Alg. 1. For this, we store in the array \textit{psum} the prefix sums of the array \textit{l}. Now, computing \textit{idx} is a standard reduction. Hence, it is safe to move the computation of \textit{sidx} corresponding to the shared dimensions outside the innermost loop.

3) Optimizations: The standard optimizations described for Alg. 1 are also applicable here, i.e. loop unroll-and-jam, vectorization. The parallelization is realized in the same way.

We now propose an optimization for Alg. 2 that improves the Instruction Level Parallelism (ILP) and the locality. In lines 13 and 14, we see an indirect access inside each row of \textit{x} via \textit{isel} which introduces stalls in the instruction pipeline. This is addressed by permuting the columns of \textit{x} such that columns with at least 2 distinct values are stored consecutively in memory, i.e. their new column numbers are from \textit{sd} + 1 to \textit{d}. The permutation is also applied to the 1d arrays \textit{l} and \textit{psum}, and the \textit{d}-dimensional array \textit{b}. Note that for \textit{sha} many values from \textit{b} are skipped since multiple dimensions of \textit{b} are fixed, i.e. are invariant with respect to the rows of \textit{x}. The permutation of \textit{b} has the effect of packing closely together the values that are actually used from it, i.e. the variable dimensions at the end. For a 2-dimensional \textit{b} this is equivalent to transposition of \textit{b} for removing column-wise accesses. The permutation is an input dependent optimization as it is beneficial only when \textit{m} is big enough. The permutation of \textit{b} is done for every call to Alg. 2, thus it is costly. In contrast, \textit{x}'s permutation is less expensive as it is done once for all invocations of Alg. 2. The permutation of \textit{l} and \textit{psum} is per call but is cheap as these are short 1d arrays. We determine the minimal \textit{m} that can trigger the permutation of \textit{b} and \textit{x} through search based auto-tuning.

B. The car. Pattern

1) Detection: Let \textit{X} be the set of interpolation points. If \textit{X} = C_1 × C_2 × \ldots × C_d we say that the input data corresponds to the \textit{car} pattern. For detecting this pattern we go over each column of the matrix \textit{x} and count the unique values per column. This is done using a tree based hashable in \textit{O}(m · log(m)) time per column. Subsequently, we multiply the \textit{d} counters, one for each column. We know that if the product equals \textit{m}, the number of rows of \textit{x}, then \textit{x} is
Algorithm 4 Version for 3d. pattern: cinterp_3d(in: m, d, x[m][d], l[d], b[2][1]...[2][d]), sd, sel, isel, in/out: r[m]).

1: Note: this algorithm includes lines 1 - 9 from Alg. 2
2: for sl = 1 to size[isel[1]] do
3:   p1 ← sp · φ[isel[1]][isel[2][sl]]
4:   idx1 ← idx + floor(2[isel[1]], c[isel[1]][sl] · gsum[isel[1]+1]
5: for st2 = 1 to size[isel[2]] do
6:   p2 ← p1 · φ[isel[2]](isel[2][sl])
8: for st3 = 1 to size[isel[3]] do
9:   p3 ← p2 · φ[isel[3]](isel[3][sl])
11: r[t] ← r[t] + b[idx3] · p3
12: t ← t + 1
13: end for
14: end for
15: end for

the result of a Cartesian product provided that there are no duplicate rows in x. Regarding the detection’s complexity, it is $O(d \cdot m \cdot \log(m))$. The same observation from sha holds also here: the detection is done once and is reused for 100,000 invocations of Alg. 3, i.e. the algorithm paired with car.

2) Description: At data structure level, the initial matrix x is replaced with a more compact c matrix containing d rows. Each row i of c contains the unique values of column i of x. Considering that the rows of c have the same number of values, the memory used for storing the interpolation points reduces from $d \cdot s^d$ to only $d \cdot s$ floating point numbers. With regard to complexity, Alg. 3 based on the traversal of matrix c has $O(m)$ time compared to $O(m \cdot d)$ for Alg. 1 and $O(m \cdot (d - s))$ for Alg. 2.

Alg. 3 assumes a more general scenario. More exactly, the rows of c may contain different number of values. The lengths of the rows are stored in the array size. In our C implementation, c is an array of pointers where each pointer references the unique set of values for its respective dimension. In Alg. 3, c is a $d \times \maxs$ matrix, where $\maxs$ is the maximum among all integer values stored in the array size. Alg. 3 uses 3 stacks: st, p, and idx. The meaning of p and idx is the same as before with the difference that they are now arrays, not scalars. Having them as arrays enables us to reuse the results of floating point and integer operations. st and k generate the interpolation points.

3) Optimizations: For this algorithm, achieving vectorization is different from Alg. 1 and Alg. 2. It can be achieved in 2 ways. We can insert a for loop at line 5 by modifying the condition at line 4 from $k = d+1$ to $k = d$. The newly inserted loop iterates over all the values from row $d$ of c, computes the product $(p[d])$ and the index $(idx[d])$, and finally updates $r[t]$. This can be easily vectorized.

Another way to vectorize Alg. 3 relies on further specializing car, e.g. for 3d interpolation points into the 3d pattern whose corresponding Alg. 4 replaces st with equivalent 3 for loops whose iteration variables are: stI, st2, and st3. In this scenario, we use instead of st a loop nest where each loop iterates over a row of c. We assume in this case that $d - 3$ dimensions are fixed as in Alg. 2 whereas 3 dimensions are variable and correspond to the Cartesian product. In other words, we handle the $d - 3$ dimensions as in lines 1 - 9 from Alg. 2. Lines 2 - 15 from Alg. 4 generate the Cartesian product. The advantages here are two-fold: a simpler structure of the loop nest and less array indexing. Note the arrays indexed using constant integers, e.g. isel[...], size[isel[...]], l[isel[...]], and psmp[isel[...]] + 1. They can all be stored in variables / registers. For this purpose, in our implementation of Alg. 4 we use temporary variables which we initialize at the beginning of the loop nest by indexing the different arrays. They are then used in the loop nest body instead of the equivalent but more expensive array references. The consequence is that compilers can more easily apply loop transformation to these loops.

Similarly to Alg. 1 and Alg. 2, we can also apply here a permutation that reduces the number of floating point and integer operations and improves the locality. The central idea is to order ascendantly the rows of c according to the number of values they contain. Such a permutation of c implies also a corresponding permutation of l, b, and this time also of r.

This technique has several benefits. A first advantage is explained in Fig. 4. Generating the Cartesian product is similar to traversing a tree in a depth-first fashion. Level i in the tree maps to row i of c. The orientation of the edges shows how values are traversed in c. The number of edges directly correlates with the number of operations executed. The same problem is computed in Fig. 4a and Fig. 4b but first without permutation and then with permutation of c. In Fig. 4a the rows of c have the lengths 4, 2, and 1. After permutation, they become 1, 2, 4. The number of operations (edges) reduces from 31 to 13, i.e. 2.4x less operations.

Second, the loop that is inserted for vectorization in Alg. 3
TABLE I: Benchmark size. Number of calls to interpolation core and size of sparse grid depending on the dimensionality.

<table>
<thead>
<tr>
<th># dims</th>
<th># calls interp. core</th>
<th># sparse grid points</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>286</td>
<td>114,687</td>
</tr>
<tr>
<td>4</td>
<td>1,001</td>
<td>471,041</td>
</tr>
<tr>
<td>5</td>
<td>3,003</td>
<td>1,579,007</td>
</tr>
<tr>
<td>6</td>
<td>8,008</td>
<td>4,571,137</td>
</tr>
<tr>
<td>7</td>
<td>19,448</td>
<td>11,829,247</td>
</tr>
<tr>
<td>8</td>
<td>43,758</td>
<td>28,000,257</td>
</tr>
<tr>
<td>9</td>
<td>92,378</td>
<td>61,616,127</td>
</tr>
<tr>
<td>10</td>
<td>184,756</td>
<td>127,574,017</td>
</tr>
</tbody>
</table>

has more iterations, meaning the SIMD units on current CPUs with up to 8 single precision SIMD lanes [18] can be used more efficiently.

Third, locality is improved following the same logic explained for Alg. 2. For this algorithm too, permutation depends on \( m \) and auto-tuning is done to find the minimum \( m \) for which permutation is profitable. We emphasize again that transforming \( x \) to \( c \), permuting \( c \) and \( r \) are cheap as they are done once for all invocations of Alg. 3. Permutation of \( b \) is expensive as it is executed per call.

IV. Evaluation

This section describes our experiments that prove the benefits of the proposed algorithms for interpolation.

A. Experimental Setup

Our algorithms are implemented in C and the parallelization is based on OpenMP. For compilation, we use Intel icc v. 12 with “-O3 -xHost”. We focus on the scenario depicted in Fig. 3b as it allows for a straightforward comparison of the algorithms and is found in our computational steering scenarios. This means that the interpolation points belong to a 3d regular grid. We emphasize that our algorithms are independent from the values stored in the sparse grid. They exploit patterns found in the set of interpolation points provided as input.

The dimensionality, \( d \), is from 3 to 10, while the number of interpolation points is \( 10^4 \) for the serial case and \( 10^5 \) for parallel. The sizes of the sparse grids for the different \( d \) are shown in Table I. The size of the multi-dimensional array \( b \) is up to 1024 (floating point numbers). The performance is always single precision and is measured for the entire interpolation routine that calls the algorithms multiple times, up to 184,756 as shown in Table I. For evaluation, we use 4 multi-core systems:

- Intel Nehalem, dual-socket, hexa-core, 24 hardware threads, 2.93 GHz
- Intel Sandy Bridge, single-socket, quad-core, 8 hardware threads, 3.4 GHz
- AMD Opteron Barcelona, 4 sockets, quad-core, 2.29 GHz
- AMD Opteron Magny-cours, dual-socket, 8 cores per socket, 2 GHz.

B. Results

On Sandy Bridge, the serial performance for different dimensionalities is depicted in Fig. 5. The “++” versions include permutation as described before for \( sha \), \( car \), and \( 3d \). The \textit{default} interpolation from [4] is used as the reference for the speedup. Note that this implementation is the state-of-the-art and is vectorization friendly, allowing us to harness the computational power of both CPUs and GPUs. Its benefits at both data structure level and algorithm level over other implementations of sparse grids are covered in [4]. \textit{default} includes only standard loop transformations described in [17], i.e. loop interchange, loop unroll-and-jam, and vectorization. These transformations accelerate the unoptimized routine up to 7x on Sandy Bridge. Fig. 5 shows that our input specialized versions are 8.8x faster than \textit{default} on Sandy Bridge. Hence, the overall speedup relative to the unoptimized version is 61.6x. Another point to mention here is that \( 3d \) is faster than \textit{car}, i.e. for a 10d problem, \( 3d \) is 8.8x faster than \textit{default} whereas \textit{car} is 8.2x faster. This validates to a great extent our assumption that the specialization of \textit{car} into \( 3d \) is more compiler friendly. Only for \textit{sha} permutation is beneficial for the chosen input data. In general, we have observed that the permutation of the set of interpolation points provides more benefits and less overhead than the permutation of the blocks in which the sparse grid is decomposed. This motivates a split of permutation in two parts: one for interpolation points and second for blocks. Of course, this observation is empirical and is based entirely on our data sets, meaning that it might be invalidated on different data, e.g. blocks exceeding the size of the L1 cache. For completeness, the entire theory was presented in Sec. III such that it can also be applied outside the scope of our work.

In Fig. 6a we compare the sequential performance on 4 multi-core architectures. The number of dimensions is fixed to 8. In order to pin point the fastest machine, we use an application specific metric: number of interpolations / second. GFLOPS is not applicable here since our optimizations reduce the number of floating point operations which can make the GFLOPS (measured with PAPI) lower although the execution is actually faster. The obvious conclusion is that Sandy Bridge is the best system for interpolation. Nevertheless, for neither one of our algorithms we can see a 2x speedup for Sandy Bride compared to Nehalem. Especially for \textit{default}, \textit{sha}, and \textit{sha++}, which are vectorizer friendly, a factor of 2x, coming from the transition from SSE to AVX, was expected but is not visible in the results. Although \textit{car}, \textit{car++}, \( 3d \), and \( 3d++ \) are also vectorizable, they contain a significant amount of sequential non-vectorizable code until they reach the single vectorizable loop explained for Alg. 3.

In Sec. III we have described the advantages and disadvantages of the permutation optimization. Since permutation produces overhead, it is important to activate it only when the number of interpolation points, \( m \), is big enough. For an 8d problem, Fig. 6b shows the dependence of the best version on \( m \). We can see that whenever the interpolation points fit the \( 3d \) pattern, \( 3d \) performs better than \textit{car}. Regarding permutation, it provides little benefits to \textit{car} and \( 3d \), and makes sense only when \( m \) is bigger than 102,400. If we isolate the \textit{sha} and \textit{sha++} versions from the rest we can see that the switching
Fig. 5: Serial performance of input specialized sparse grid interpolation for different number of dimensions. We can see up to 8.8x for input specialized versus default version.

Fig. 6: Serial performance across different systems and workload sizes, i.e. number of interpolations.

Fig. 7: Parallel performance of input specialized interpolation.

point on the horizontal axis is at 1600 and the benefit from switching permutation on is up to 1.76x speedup in favor of sha++ compared to sha. This is an important result especially for those cases in which sha is applicable but not car, e.g. for 8d, 5 dimensions are fixed whereas for the remaining 3 dimensions, the interpolation points are random and do no result from a Cartesian product. We use search based auto-tuning to determine m for which the best version changes. Currently, the results of the search are stored in a file so
that they can be reused across executions. Since the search takes places in an 1d space, the overhead is acceptable for our application in which auto-tuning has a low frequency and its cost is in any case amortized by reusing the search results across runs.

The parallelization using OpenMP is first evaluated on the 12-core Nehalem system with Hyperthreading. The strong scaling results are depicted in Fig. 7a for an 8d problem. We use pinning to ensure that up to 12 threads, there is 1 thread per physical core. 3d is the fastest version here. For 24 threads, its speedup relative to the sequential case is 15.4x which is bigger than the number of physical cores. In other words, 3d benefits significantly from Hyperthreading. This observation is also valid for the rest of the versions. This is an aspect which deserves further inspection and will be addressed in our future work. We briefly explain now its relevance. In general, a 1.29x improvement from multi-threading means there is still potential for serial optimizations that exploit more efficiently the ILP. Finding these optimizations is crucial for improving the serial performance and the parallel performance especially on architectures without multi-threading, e.g. the AMD Opterons included in this paper. For sparse grid interpolation, optimizations that can further improve the performance include: overlapping of computation with memory access and interleaving the interpolation on 2 or more blocks that compose the sparse grid as shown in Sec. III and Fig. 2 for a 2d case.

We can also see in Fig. 7a that for less than 12 threads, 3d++ is the fastest version whereas for more than 13 threads, 3d is up to 1.05x faster than 3d++. The conclusion is that the number of threads also influences the best algorithm. Another view of scalability is shown in Fig. 7b, this time for the 4 multi-cores considered in our evaluation. We differentiate between the case with and without Hyperthreading. We can see clearly once more how Hyperthreading’s being enabled or not affects the performance. It is also worth mentioning that the best version of interpolation is not invariant across architectures and for best performance, empirical selection of algorithms or search based auto-tuning must be employed.

V. CONCLUSION

In this paper, we have shown the importance of exploiting patterns in the input data for optimizing a computational steering application in which the main requirement is real-time visualization of simulation data compressed using the sparse grid technique. The critical routine in this application is sparse grid interpolation addressed in this paper. Our work follows the same philosophy as well known libraries like OSKI and FFTW. By analyzing the characteristics of the input data for interpolation, we have identified patterns which can be automatically detected and are handled by specialized algorithms. Our input specialized algorithms are up to 9x faster when compared to the state-of-the-art version which includes only standard optimizations, i.e. loop unroll-and-jam and vectorization. We have also shown the impact on performance of input dependent optimizations, i.e. data permutation which should be switched on only when the number of interpolation points is big enough. Permutation can result in 1.76x.

The new algorithms allow our CPU versions of interpolation to be at the same performance level as the GPU versions. [4] reports a 70x speedup on GPUs relative to a 1-core CPU. Since our input specialized algorithms, especially car and 3d, are not GPU friendly in their current form, i.e. due to complex control flow and data dependencies, the 9x speedup applies only to the CPU side, allowing CPUs to better compete with GPUs which are well suited for the default sparse grid interpolation covering the general case of random interpolation points.

The implementations of the algorithms from our paper will be released as part of fastsg [17], an open source library for sparse grid interpolation. spinterp [5], [6] is a Matlab framework for sparse grid interpolation which can also benefit from our input adaptive solution for interpolation.

ACKNOWLEDGMENT

This publication is based on work supported by Award No. UK-C00020, made by King Abdullah University of Science and Technology (KAUST).

REFERENCES