Multiple multiresolution representation of functions and calculus for fast computation

G. I. Fann,[1] R. J. Harrison,[1,2] J. Jia,[1] J. Hill,[1] and D. Galindo[1]

[1]Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37830
[2]Department of Chemistry, University of Tennessee, Knoxville, TN 37996

Email: fanngi@ornl.gov

Abstract. We describe the mathematical representations, data structure and the implementation of the numerical calculus of functions in the software environment multiresolution analysis environment for scientific simulations, MADNESS. In MADNESS, each smooth function is represented using an adaptive pseudo-spectral expansion using the multiwavelet basis to a arbitrary but finite precision. This is an extension of the capabilities of most of the existing net, mesh and spectral based methods where the discretization is based on a single adaptive mesh, or expansions.

1. Introduction

One of the appealing features of the MADNESS computational environment which enhances its usability is the high level composition which permits codes to be expressed similiar to the mathematical equations describing the physics. In order to make this possible, an adaptive representation of functions using multiresolution analysis for numerical calculus is developed to preserve the accuracy of the computation. In this paper, we focus on the computer science and implementation aspects of MADNESS. In particular, the numerical calculus for permitting arithmetic of functions using multiple resolutions is described as well as its parallel implementations to achieve user defined, arbitrary but finite, precision. We work with smooth functions and assume that they can be represented and sampled in an interval in the multiwavelet basis. Previously, the approximation of discontinuous functions was described [4].

MADNESS is designed to address the following programming and computational issues:

- raising the level of composition of scientific applications, making it faster and easier to both construct robust and correct algorithms and calculate solutions to existing and new problems,
- computing using functions and operators instead of just numbers, vectors and matrices,
- providing fast and accurate solutions to a variety of differential and integral equations in one to six dimensions (perhaps higher), and
- facilitating the use of massively parallel computational resources by a wider audience.

Using MADNESS, a code is written in terms of functions and operators, using the *C++* language and for this reason, it can be thought of as a basis-free method. There are underlying representations and approximations using bases and grids that adapt and refine automatically to satisfy the user requested precision. However, the user does not have to think about either the adaptivity or refinement until efficiency or memory become an issue.

The numerical operations can be regarded as a finite precision symbolic calculus used to express the equations. For example, the code corresponding to

$$\int_{\Omega} \frac{1}{2}(\frac{d\psi}{dx})^2 + V(x)\psi(x)^2 d\Omega$$

in three dimensions is just

```
real_function_3d dpsidx=Dx(psi);
double result = 0.5 * inner(dpsidx, dpsidx) + inner(V(x), psi*psi);
```

where psi is a 3-D function, Dx(psi) computes the derivative of psi in the x-direction, and the method inner computes the L_2 inner product $(f,g)=\int f(x)g(x)\,dx$.

Within MADNESS both differential operators and integral operators can be applied to functions. Important and common convolution operators with physically relevant free space Green functions (e.g., Coulomb, bound-state Helmholtz, free-particle quantum propagator) are built in. The details of their implementation will be described elsewhere. Finally, MADNESS excels at obtaining high-accuracy solutions, computing efficiently in many dimensions, and using massively parallel computers. However, MADNESS is not appropriate for every application. In particular, if you have complicated boundary conditions or geometries, highly oscillatory functions, or only need low precision, then other tools might be more useful.

In the following, we describe approximation of functions using multiwavelets and explain the data structure and parallel computing approach which enable its usability. In Section 2, we review multiwavelets. In Section 3, we describe the approximation of functions and the computation of some elementary composition of functions in MADNESS. In Section 4, parallel implementation and programming issues are outlined.

2. Multiwavelets

We describe our algorithms in one dimension. Most of the description carries over to many dimensions via tensor products and by rescaling the domain. We note that in higher dimensions it is advantageous to use the non-standard representations [3] to decouple interactions between scales. We use wavelet and multiwavelet interchangeably when it is convenient and clear.

We adapt the notation of [1,2] and review some results. Fix a positive integer k and the unit interval. Let $\phi_i(x) = \sqrt{2i+1}P_i(2x-1)$ denote i-th scaling function, which is the i-th Legendre polynomial $P_i(x)$ shifted to $(0,1)$ and normalized. For $k=0$, $\phi_0(x)$ is just the Haar scaling function. This set of functions span the space of "scaling functions" V_0.

Define functions $\phi_{il}^n(x) = 2^{n/2}\phi_i(2^n x - l)$, where $n = 0,1,\ldots$ and $l = 0,\ldots,2^n-1$, the dyadical scaled and translated of the scaling functions. These functions are orthonormal on $[2^{-n}l, 2^{-n}(l+1)]$ and vanish elsewhere. Denote the span of these functions by V_n^k.

The scaling functions were constructed with the following properties:

- $L_2([0,1]) = \overline{lim_{n\to\infty} V_n^k}$
- Given an initial level 0, $V_0^k \subset V_1^k \subset V_2^k \ldots$
- Invariant under translation
- If $f \in V_n$, then $f(2x) \in V_{n+1}$
- For a given scale n, $\int \phi_{il}^n(x)\phi_{jm}^n(x)dx = \delta_{ij}\delta_{lm}$[1].

[1]The δ_{ij} is the Kronecker delta function.

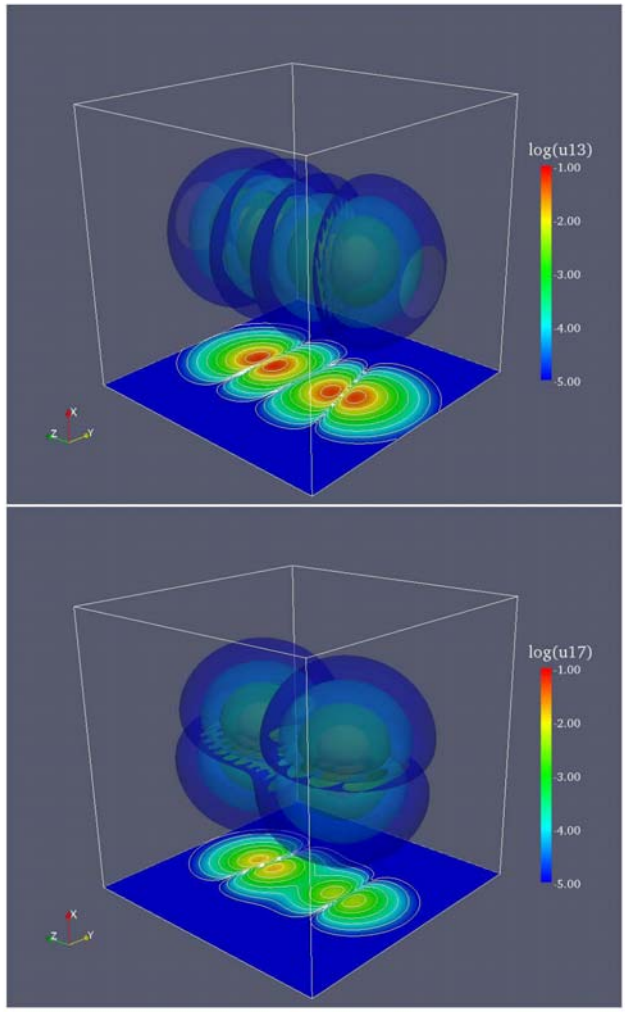

Figure 1. MADNESS has be used to compute electronic structure of molecules and nuclear structures of elements. The logarithm of the absolute value of the real part of the quasi-particle wavefunctions for nuclear density functional theory simulations [6].

The multiwavelet space W_n^k, is defined to be the orthogonal complement of V_n^k in V_{n+1}^k. In Alpert's thesis, there are two choices for the orthonormal multiwavelets, $\psi_0(x)$ in W_0^k; vanishing moments or regularity. We primarily use the multiwavelets with vanishing moments, i.e., $\int \psi_j(x)x^i dx = 0$, for $i,j = 0,\ldots,k-1$. In particular, if a function has a Taylor expansion up to order $k-1$, then the wavelet coefficients up to order $k-1$ are zero. The space W_n^k is spanned by $2^n k$ functions obtained from $\psi_0(x),\ldots,\psi_{k-1}(x)$ by dilation and translation, $\psi_{jl}^n(x) = 2^{n/2}\psi_j(2^n x - 1)$.

A key property is that Alpert's multiwavelets generate a form of multiresolution analysis [5] and are orthogonal. Let $L_2([0,1])$ represent the space of L_2-integrable functions on the unit interval with the L_2 metric, $< f,g >= \int_0^1 fg dx$.

Without loss of generality, we drop the k notation, unless necessary. Since $V_0 \subset V_1$, the basis functions for level 0 can be expressed in terms of the basis of V_1. Thus there exists constants $h_{ij}^0, h_{ij}^1, g_{ij}^0, g_{ij}^1$ such that

$$\phi_i(x) = \sqrt{2}\sum_{j=0}^{k-1} (h_{ij}^0 \phi_j(2x) + h_{ij}^1 \phi_j(2x-1)),$$

and

$$\psi_i(x) = \sqrt{2}\sum_{j=0}^{k-1} (g_{ij}^0 \phi_j(2x) + g_{ij}^1 \phi_j(2x-1)).$$

The key property of multiresolution analysis (MRA) is the separation of the behavior of functions and operators at different length scales. Central to this concept is the telescoping series which exactly represents the basis at a finest scale n in terms of basis from level 0 with successive corrections at successively finer scales,

$$V_n = V_0 + (V_1 - V_0) + \ldots + (V_n - V_{n-1}).$$

If the function is sufficiently smooth and regular in some region to be represented at the desired precision at some level, then the differences at finer scales will be negligibly small.

Each definition of multiwavelet basis and the scaling function for each level n gives a dyadic subdivision of the unit interval which is naturally represented as a dyadic tree.

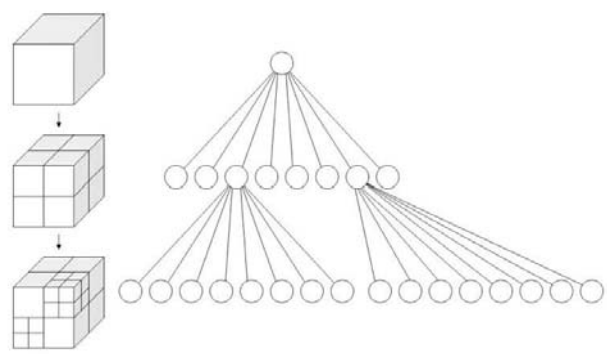

Figure 2. The multiresolution representation of a function using multiwavelets is naturally represented in a tree structure. A 3-D function is shown which corresponds to an octree (Octree picture is from WhiteTimberwolf's contribution at Wikipedia).

3. Numerical Calculus in MADNESS

In order to perform function calculus in a user friendly fashion, arithmetic operations of functions and functions of functions must also be defined and computed both efficiently and in a fast manner in the multiwavelet representation without resorting to multiple transformations and costly quadratures for resampling. The orthonormality property of the multiwavelet basis make these computations fast and efficient.

We describe the computation of a few of these representations below.

3.1. Approximation of Smooth Functions

Let $f \in L_2([0,1])$ be a smooth function. Its projection, $\hat{f} \in V_n$ can be written as $\hat{f}(x) = \sum_{l=0}^{2^n-1} \sum_{j=0}^{k-1} s_{jl}^n \phi_{jl}^n(x)$. The coefficients s_{jl}^n are computed as $s_{jl}^n = \int_{2^n l}^{2^n(l+1)} f(x)\phi_{jl}^n(x)dx$. The function expanded in the multiwavelet basis is given as

$$f(x) = \sum_{j=0}^{k-1} s_{j,0}^{0} \phi_j(x) + \sum_{j=0}^{k-1} \sum_{m=0}^{n-1} \sum_{l=0}^{2^m-1} d_{jl}^{m} \psi_{jl}^{m}(x),$$

with the multiwavelet coefficients computed as $d_{jl}^{m} = \int_{2^{-m}l}^{2^{-m}(l+1)} f(x)\psi_{jl}^{m}(x)dx$. Since the basis are piecewise polynomial an adaptive Gauss-Legendre quadrature is used.

We use the term "compression" to mean multiwavelet compression, analogous to wavelet compression of an image. In many examples an image or a signal (denoted by f) can be written as $f = u + v$ where u is a structured component and v is a noisy component. The structure part u can often be represented using very few coefficients (in a compressed form).

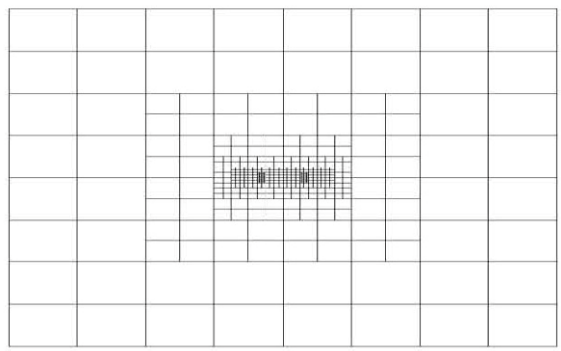

Figure 3. A two-dimensional slice of the three-dimensional adaptive refinement of cubes shows where significant scaling and multiwavelet coefficients are required for the representation of the singular potential corresponding to the nuclei of the Hydrogen molecule.

3.2. Truncation Criteria

Discarding small differences coefficients in the multiwavelet expansion while maintaining precision is crucial to speed and sparsity, and drives the adaptive refinement. Recall that for smooth functions the level of the multiwavelet expansion depends on the user given error criteria and is dependent, dyadically, on the averaged value of the basis functions and dies off with respect to level of adaptive refinement and the box sizes.

Different truncation criteria are used in MADNESS in the multiwavelet representation. Let d_l^n denote the tensor of coefficients d_{il}^n for level n, leaf node l. The norm $||d_l^n||_2$ denotes the norm $\sum_i \sqrt{|d_{il}^n|^2}$ of d_l^n.

Mode 0: The difference coefficients of the leaf nodes are neglected according to

$$\sum_i \sqrt{|d_{il}^n|^2} \leq \varepsilon.$$

This truncation is appropriate for most calculations where the functions have deep levels of refinements (such as those near singularities, steep gradients or discontinuities).

Mode 1: Mode 1 is appropriate when an accurate representation of both the function and its derivative is required. Difference coefficients of leaf nodes are neglected according to

$$||d_l^n|| \leq min(1, c * 2^{-n})\varepsilon$$

where c is a constant depending on the simulation domain size before rescaling to the unit interval.

Mode 2: Mode 2 is appropriate only for smooth functions with nearly uniform level of refinement for the entire domain. The multiwavelet difference coefficients are neglected according to

$$||d_l^n|| \leq 2^{-nd/2}\varepsilon$$

where d is the dimension of the approximation. In this mode, the error between the representation between the levels $n-1$ and n is within ε, $||f^n - f^{n-1}||_2 \leq \varepsilon$.

3.3. Adaptive Refinement

After projection has been performed at boxes number $2l$ and $2l+1$ at level n, the scaling function coefficients may be tranformed or filtered to produce multiwavelet coefficients at box l at level $n-1$. If the desired truncation is not satisfied, the process is repeated at the child boxes $4l, 4l+1, 4l+2, 4l+3$ at level $n+1$. Otherwise, the computed coefficients are inserted at level n.

3.4. Evaluation

Once a function has been represented at level n with the desired approximation. The function can be evaluated at different points by recurring down the dyadic tree to find the appropriate box which contains the point, evaluating the basis functions and summing over its products with the coefficients.

3.5. Products

We briefly describe the procedure for multiplication of multiwavelet functions represented in an interval [2]. Consider two functions f and g approximated in the multiwavelet basis, each represented on the interval by k-term interpolating scaling polynomial expansions. The interpolating scaling functions $\phi(x)$ constructed from the Legendre interpolating polynomial at the quadrature points is related to the scaling functions $\phi(x)$ by

$$\phi_j(x) = \sqrt{w_j} \sum_{i=0}^{k-1} \phi_i(x_j)\phi_i(x)$$

where $x_0, \ldots x_{k-1}$ denote the roots of the Legendre polynomail $P_k(2x-1)$ and its associated quadrature weights $w_0, \ldots w_{k-1}$. There is a fast transform to and from the multiwavelet scaling functions and the interpolating scaling functions.

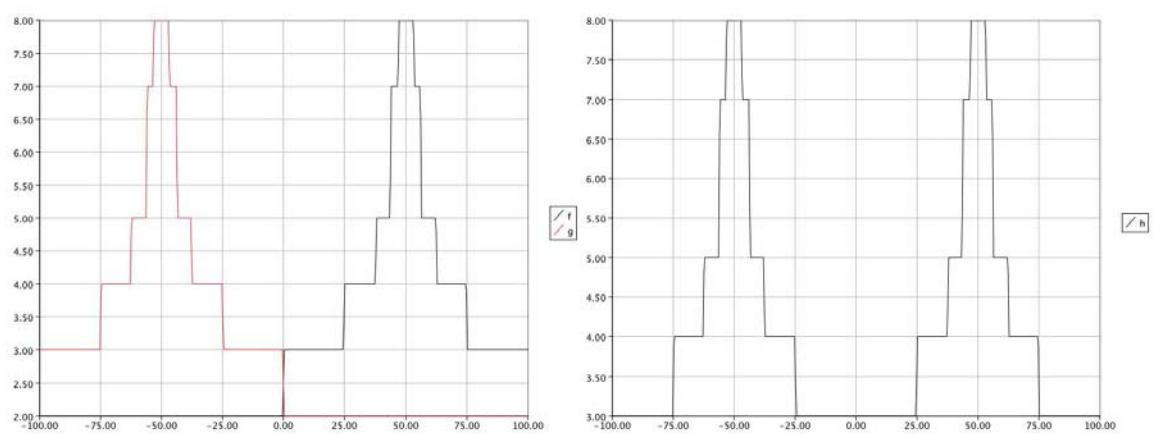

Figure 4. The adaptive structure of two functions $f(x) = 1 + e^{(-x+50)^2}$ and $g(x) = 1 + e^{(-x-50)^2}$ and their product $h = fg$ is shown. The vertical axis indicates the level n of the multiwavelet expansion that is used in the adaptive discontinuous expansion.

We obtain the product $f \cdot g$ as follows. Let $f(x) = \sum_{j=0}^{k-1} \sum_{l=0}^{2^n-1} a_{jl} \phi_{jl}^n(x)$ and $g(x) = \sum_{j=0}^{k-1} \sum_{l=0}^{2^n-1} b_{jl} \phi_{jl}^n(x)$, with possibly two different n. Then:

- Refine the functions f and g so that the representations of the two functions are at the same level.
- Now refine the representation of f and g by dividing the subinterval on the finest level into two, to allow sufficient resolution to represent the product.
- The coefficients of the product of representations $h(x) = f(x)g(x) = \sum_{j=0}^{k-1} c_{jl} \phi_{jl}^n(x)$ on a subinterval at level n is $c_j = (2^{n/2}) \frac{a_j b_j}{\sqrt{w_j}}$, where w_j are the Gaussian quadrature weights.
- The threshholding is applied. The coefficients for the two finest levels n and $n+1$ are compared. If the coefficiencts at level $n+1$ are below the truncation threshold, those coefficients are deleted.

The case of squaring a function is special and can be performed faster since it is an in-place operation. In some scenarios, it can be used to multiply two functions. This follows from Beylkin's observation that $f \cdot g = \frac{1}{4}((f+g)^2 - (f-g)^2)$.

In MADNESS, a slight different approach is used. Let us describe the point-wise multiplication since this is fundamental. The products of two functions is performed in the scaling function basis. Intuitively, each function is transformed to values at the quadrature points, multiply the values and then transformed back. In practice and in robust implementation there are three complicating issues.

First, the product cannot be exactly represented in the polynomial basis. The product of two polynomials of order $k-1$ produces a polynomial of order $2k-2$. Beylkin makes a nice analogy to the product of two functions sampled on a grid – the product can be exactly formed on a grid with double the resolution. This is not exact for polynomails but the expected error is reduced by a factor 2^{-k} where k is the number of multiwavelets.

In MADNESS, an option to automatically refine and form the product at a lower level is given. This is performed by estimating the norm of each of the leaf boxes which is a products of the Frobenius norm of the two functions.

Second, the functions may have different levels of adaptive refinement. The two options are to compute the function with coefficients at a coarser-scale directly on the grid required for the finer-scale function, or to refine the function down to same level, which is what we previously choose to do. However, this leaves the tree with scaling function coefficients at multiple levels that must be cleaned up after the operation. Since it essential (for efficient parallel computation) to perform multiple operations at a time on a function, having it in an inconsistent state makes things complicated. If all we wanted to do were perform other multiplication operations, there would be no problem; however this seems to be an unnecessary restriction on the user. It is also faster to perform the direct evaluation and this is what has been implemented.

Third, the above does not use sparsity or smoothness in the functions and does not compute the result to a finite precision. For instance, if two functions do not overlap their product is zero but the above algorithm will compute at the finest level of the two functions doing a lot of work to evaluate small numbers that will be discarded upon truncation. Eliminating this overhead is crucial for reduced scaling in electronic structure calculations. At some scale we can write each function in the domain in terms of its usual scaling function approximation at that level s and the correction/differences d from all finer scales. The error in the product of two such functions is then the sum of the products of the approximation of the scaling function coefficients and multiwavelet difference coefficients for f and g.

Note again that while the scaling function coefficients are as used everywhere else in this document, the difference multiwavelet part of the expansion is the sum of corrections at all finer scales. Thus, by computing the scaling function coefficients at all levels of the tree and summing the norm of the differences up the tree we can compute with controlled precision at coarser levels of the tree. The sum of

the norm of differences can also be computed by summing the norm of the scaling function coefficients from the finest level and subtracting the local value.

3.6. Quotients and Rational Functions

Similiar to products we describe the procedure for representating rational functions $f(x)/g(x)$. In this case we have to find the coefficients of $h(x)$ such that $f(x) = h(x)g(x)$. From the procedures for products, at level n, $a_j = (2^{n/2})\frac{c_j b_j}{\sqrt{w_j}}$. Thus, $c_j = 2^{-n/2}\sqrt{w_j}a_j/b_j$ with $b_j \neq 0$.

3.7. Square Root

The square root of a function can also be described. Using the products of two functions we have the coefficients for $h(x) = f^2(x)$ being represented on level n as $c_j = (2^{n/2})\frac{a_j a_j}{\sqrt{w_j}}$. The coefficients of the function $f(x) = \sqrt{h(x)}$ can be represented as $\sqrt{2^{-n/2}c_j\sqrt{w_j}}$.

Computation of cubic and higher roots of functions can be performed in an analogous way.

4. Data Structure and Computational Methodologies

Multiwavelet representations are best understood in terms of 2^d-tree for d dimensions correspond to the dyadic scaling of the basis functions in the tensor product representation. Each level of the tree corresponds to a level of the refinement using the 2-scale relation in each dimension. The difficulties of the tree architecture is the serial manner in which efficient tree traversal algorithms are implemented, as the order of summation that must be performed must vary to minimize round-off or numerical summation errors. In MADNESS, the tree is mapped to a hash table. In this section we assume the reader has some knowledge of $C++$ data structure and templates as well as parallel computing.

A d-dimension function is represented as a 2^d-tree. Nodes in the tree are labeled by a tuple key (n, l) where n is the level and l is a d-vector representing the translation. Nodes are represented by instances of FunctionNode<T,d>, where T is a templated data type, that presently contains the coefficients and an indicator if this node has children. Nodes, indexed by keys, are stored in a distributed hash table that is an instance of a MADNESS container class. This container uses a two-level hash to first map a key to the processes (referred to as the owner) in which the data resides, and then into a local instance of either a GNU hash_map or a TR1 unordered_map. Since it is always possible to compute the key of a parent, child, or neighbor we never actually store (global) pointers to parents or children. Indeed, a major aim of the MADNESS runtime environment is to replace the cumbersome partitioned global address space (global pointer is process id + local pointer) with multiple global name spaces. Each new container provides a new name space. Using names rather than pointers permits the application to be written using domain-specific concepts rather than a rigid linear model of memory.

Data common to all instances of a function of a given dimension (d) and data type (T, e.g., float, double, float_complex, double_complex) are gathered together into FunctionCommonData<T,d> of which one instance is generated per wavelet order (k). An instance of the common data is shared read-only between all instances of functions of that data type, dimension and wavelet order. In addition to reducing the memory usage of the code, sharing the common data greatly speeds the instantiation of new functions which is replicated on every processor. In order to facilitate shallow copy and assignment semantics and to make empty functions inexpensive to instantiate, a multi-resolution function, which is an instance of Function<T,d> contains only a shared pointer to the actual implementation which is an instance of FunctionImpl<T,d>. Uninitialized functions (obtained from the default constructor) contain a zero pointer. Only a non-default constructor or assignment actually instantiate the implementation. The main function class forwards nearly all methods to the underlying implementation. The implementation primarily contains a reference to the common data, the distributed container storing the tree, little bits of other state (e.g., a flag indicating the compression status) and several methods.

Default values for all functions of a given dimension are stored in an instance of FunctionDefaults<d>. These may be modified to change the default values for subsequently created functions. Functions have many options and parameters and thus we have made it easy to specify options and selectively override defaults. Since $C++$ does not provide named arguments (i.e., arguments with defaults that may be specified in any order rather than relying on position to identify an argument) we adopt the named-parameter idiom. The main constructor for Function<T,d> takes an instance of FunctionFactory<T,d> as its sole argument. The methods of FunctionFactory<T,d> enable setting of options and parameters and each returns a reference to the object to permit chaining of methods.

Most memory is reference counted using Boost-like shared pointers. An instance of SharedPointer<T>, which wraps a pointer of type T*, is used to wrap memory obtained from the $C++$ new operator. The exceptions are where management of the memory is immediately given to a low-level interface. Shared-pointers may be safely copied and used with no fear of using a stale pointer. When the last copy of a shared pointer is destroyed the underlying pointer is freed. With this mode of memory management there is never any need to use the $C++$ delete operator and most classes do not even need a destructor.

4.1. Maintaining a consistent state of the 2^d-tree

The function implementation provides a method verify_tree() that checks connectivity and consistency of the compression state, presence of coefficients, child flags, etc, as described below.

A node in the tree is labeled by the tuple (n, l) key and presently stores the coefficients and a flag indicating if the node has children. In 1D, the keys of children are readily computed as $(n + 1, 2l)$ and $(n + 1, 2l + 1)$. In many dimensions it is most convenient to use the KeyChildIterator class. In the reconstructed form (i.e. the scaling function basis rather than the multiwavelet basis), a tree has coefficients (held in kd tensors) only at the lowest level. All interior nodes will have no coefficients and will have children. All leaf nodes will have coefficients and will not have children.

In the compressed form (wavelet basis), a tree has coefficients (in $(2k)d$ tensors) at all levels. The scaling function block of the coefficients is zero except at the coarsest level. Logically, this tree is one level shallower than the reconstructed tree since the scaling function coefficients at the leaves are represented by the difference coefficients on the next coarsest level. However, to simplify the logic in compress and reconstruct and to maintain consistency with the non-standard compressed form (see below), we do not delete the leaf nodes from the reconstructed tree. Thus, the compressed tree has the same set of nodes as the reconstructed tree with all interior nodes having coefficients and children, and all leaf nodes having no coefficients and no children.

In the non-standard compressed form (redundant basis), we keep the scaling function coefficients at all levels and the wavelet coefficients for all interior nodes. Thus, the nonstandard compressed tree has the same set of nodes as for the other two forms but with all nodes having coefficients (a $(2k)d$ tensor for interior nodes and a kd tensor for leaf nodes).

To keep complexity to a minimum we don't want to introduce special states of the tree or of nodes, thus all operations restore the tree to a standard state by their completion.

Truncation is applied to a tree in compressed form and discards small coefficients that are logically leaf nodes. Logically, because in the stored tree we still have the empty nodes that used to hold the scaling coefficients. For a node to be eligible for truncation it must have only empty children. Thus, truncation proceeds as follows. We initially recur down the tree and for each node spawn a task that takes as arguments futures indicating if each of its children have coefficients. Leaf nodes, by definition, have no children and no coefficients and immediately return their status. Once a task has all information about the children it can execute. If any children have coefficients a node cannot truncate and can immediately return its status. Otherwise, it must test the size of its own coefficients. If it decides to truncate, it must clear its own coefficients, delete all of its children, and set its has_children flag to false. Finally, it can return its own status.

Adding (subtracting) two functions is performed in the wavelet basis. If the trees have the same support (level of refinement) we only have to add the coefficients. If the trees differ, then in addition to

adding the coefficients we must also maintain the has_children flag of the new tree to produce the union of the two input trees. To permit functions with different processor maps to be added efficiently, we loop over local data in one function and send them to nodes of the other for addition. Sending a message to a non-existent node causes it to be created.

Acknowledgements

This work was supported by the Scientific Discovery through Advanced Computing (SciDAC) program of the U.S. Department of Energy, Office of Science, the Division of Basic Energy Sciences, and the Office of Advanced Scientific Computing Research, Program in Applied Mathematics and through the SciDAC Scientific Application Pilot Program under contract number DE-AC05-00OR22725 with UT-Battelle, LLC (Oak Ridge National Laboratory). It is also supported under the UNEDF SciDAC Collaboration. Computational resources were provided by the National Center for Computational Sciences at Oak Ridge.

References

[1] B. Alpert. A class of bases in L^2 for the sparse represetntation of integral operators. *SIAM J. Math. Anal.*, 24(1):246–262, 1993.

[2] B. Alpert, G. Beylkin, D. Gines, and L. Vozovoi, Adaptive solution of partial differential equations in multiwavelet bases. *J. Comput. Phys.*, 182(1):149–190, 2002.

[3] G. Beylkin, R. R. Coifman, and V. Rokhlin. Fast wavelet transforms and numerical algorithms I. *Comm. Pure and Appl. Math.*, 44:141–183, 1991. Yale Univ. Technical Report YALEU/DCS/RR-696, August 1989.

[4] M.E. Brewster, G.I. Fann, and Z.Y. Yang, Wavelets for electronic structure calculations. *Journal of Mathematical Chemistry*, 22(2-4):117–142, 1997.

[5] Stephane G. Mallat, Multiresolution approximations and wavelet orthonormal bases of $L^2(\boldsymbol{R})$. *Trans. Amer. Math. Soc.*, 315(1):69–87, 1989.

[6] J.C. Pei, M.V. Stoitsov, G.I. Fann, W. Nazarewicz, N. Schunck, and F.R. Xu. Deformed coordinate-space Hartree-Fock-Bogoliubov approach to weakly bound nuclei and large deformations. *Physics Review C*, 78, 064306, 2008.