Sparse data structure design for wavelet-based methods

Ecole CNRS - Fréjus, juin 2010

Méthodes multirésolution et méthodes de raffinement adaptatif de maillage

Guillaume Latu

CEA, IRFM, F-13108 Saint−Paul−lez−Durance, France.

guillaume.latu@cea.fr

and

University of Strasbourg & INRIA/Calvi project

7 rue Descartes, 67084 Strasbourg Cedex, France

Abstract

This course gives an introduction to the design of efficient datatypes for adaptive wavelet-based applications. It presents some code fragments and benchmark technics useful to learn about the design of sparse data structures and adaptive algorithms. Material and practical examples are given, and they provide good introduction for anyone involved in the development of adaptive applications. An answer will be given to the question: how to implement and efficiently use the discrete wavelet transform in computer applications? A focus will be made on time-evolution problems, and use of wavelet-based scheme for adaptively solving partial differential equations (PDE). One crucial issue is that the benefits of the adaptive method in term of algorithmic cost reduction can not be wasted by overheads associated to sparse data management.

1 Introduction

1.1 Multiscale and adaptive approaches

Mathematical techniques such as multiresolution analysis, multigrid methods, adaptive mesh refinement, among others, have produced significant advances in understanding multiscale problems [17]. However, these techniques have typically been developed and employed in very specific application areas. Consequently, multiresolution research remains largely disjointed among several disciplines, and researchers within each discipline are unlikely to be familiar with a broad range of techniques in the other fields. The development of adaptive methods needs a greater exchange of information among disparate lines of research. Multiresolution methods yield sparse, complex, hierarchical data representations. The data structures that implements sparsity may hinder performances if one does not pay enough attention. Therefore, there is an actual need to develop new mathematical techniques, new computational methods, and new softwares in a collaborative framework. It will allow one to better address challenging scientific problems that require these adaptive techniques.

In this context, *wavelets* have proven especially useful. When trying to classify wavelet applications by domain, it is almost impossible to sum up the thousands of papers written since the 1990s. The present document addresses the design of data structures and algorithms in wavelet-based applications. We will look at how to implement and efficiently use the *discrete wavelet transform* in computer applications and especially we will focus on time-evolution problems. One aim of this document is to reinforce the bridge between mathematicians and computer scientists working on wavelet-based applications.

The reader of this document is expected to have some basic knowledge of computer science, C language and wavelet analysis theory. Examples and source code included in this document are available from the web site `http://icps.u-strasbg.fr/people/latu/public_html/wavelet/data_wav.tgz`.

1.2 Some wavelet applications

Because there are so many applications that use the wavelet decomposition as a tool, we will focus on applications of wavelets in just a few fields. The connection between these fields is the use of the discrete wavelet transform computation on potentially huge data.

1.2.1 Image compression

Image compression consists in a reversible transform that maps the image into a set of coefficients which are finally encoded. An image, decomposed in wavelet coefficients, can be compressed by ignoring all coefficients below some threshold values. Threshold values are determined based on a quality number calculated using the **S**ignal-to-**N**oise **R**atio (often abbreviated SNR). The trade-off between compression level and image quality depends on the choice of the wavelet function, the filter order, the filter length, and the decomposition level. The optimal choice depends on the original image quality and on the expected computational complexity.

Numerous compression schemes have been developed in the image processing community. Wavelet encoding schemes are increasingly used and successfully compete with Fourier-based alternatives. They provide high compression rates together with a high SNR.

1.2.2 Video encoding, signal processing

In images and videos, the lowest frequencies, extracted by high scale wavelet functions, represent flat backgrounds. The high frequencies (low scale wavelet functions) represent textured regions [39, 54]. The low-pass subband gives an approximation of the original image, the other bands contain detail information.

Bit allocation is crucial in this context, as image parts with low energy levels should have fewer bits. The wavelet filters should be chosen to maximize encoding gain and to avoid adding noise. After the 2D wavelet analysis, *quantization* is performed on the coefficients. *Quantization* is a procedure that maps a continuous set of values (such as real numbers) into a set of discrete values (such as integers from 0 to 255). The coefficients are grouped by scanning and then entropy coded for compression. *Entropy coding* is a form of lossless data compression. The process of *entropy coding* can be split into two parts: modeling and coding. Modeling assigns probabilities to the coefficients, and coding produces a bit sequence from these probabilities. A coefficient with probability p gets a bit sequence of length $-log(p)$. So the higher the probability of the coefficient, the shorter the bit sequence. This coding technique reduces the length of the original bit sequence of coefficients. Entropy coding maps data bit patterns into code words, replacing frequently-occurring bit patterns with short code words.

In video compression, the time dimension has to be taken into account. This expands the wavelet analysis to 3D. Compression can be performed similarly to 2D image compression. Another video compression method performs motion estimation. This approach is based on the compression of the separate images in combination with a motion vector. In the synthesis bank, the separate images are reconstructed and the motion vector is used to form the subsequent frames and to reduce the differences between these frames [54].

1.2.3 Application to PDEs

Partial differential equations (PDEs) can be used to describe physical phenomena. They are often difficult to solve analytically, hence numerical methods have to be used. In order to keep computation time low enough, partial differential equations that encounter either singularities or steep changes require non-uniform time-spatial grids or moving elements. Wavelet analysis is an efficient method for solving these PDEs [13, 27, 39, 59]. The wavelet transform can track the position of a moving steep front and increase the local resolution of the grid by adding higher resolution wavelets. In the smoother regions, a lower resolution can be used. An essential point is the ability to locally refine the grid in the regions where the solution exhibits sharp features. This implies a strategy for dynamically identifying those critical regions and allocating extra grid points accordingly.

In the present work, this is done by taking advantage of multiresolution data representation. This adaptive strategy allows for a nearly constant discretization error throughout the computational domain. In addition, because smooth regions are represented by only a relatively small amount of data, memory and processor requirements are reduced. Grid points are removed or added according to the magnitude of the corresponding wavelet coefficient. If the wavelet coefficient is smaller than a predefined threshold, the grid point can be removed. Transient changes in the solution are accounted for by adding extra grid points at the same levels and at lower levels. To exemplify this point, an adaptation procedure in 1D can be described as follows [13, 39, 59]. Suppose we have a given function f sampled on a uniform grid $x_{k\in[0,2N]}$ and we want to adapt the grid to the function. A criterion for this adaptation is:

1. Compute an interpolation $P(x_{2k+1})$ for each *odd* grid point x_{2k+1} of the grid using the f values known on *even* points $[f(x_{2k})]_{k=*}$.

2. Compute the interpolative error coefficients

$$d_k = \left|P(x_{2k+1}) - f(x_{2k+1})\right|,$$

   and apply the grid reduction/extension according to

   (a) if $d_k^j < \epsilon$, remove the grid point,

   (b) if $d_k^j \geq \epsilon$, keep the grid point.

If the original function is smoothly varying, then the details d_k^j are small and associated grid points can be removed. Doing so, the number of grid points in the sparse representation is optimized; and, the accuracy of the representation stays really under control.

1.3 Objectives

In this document, the design of sparse data structures and related algorithms for wavelet-based applications are discussed.

A sparse data structure is usually described as a data in which many of the coefficients are equal to zero, so that we can gain both in time and space by working only on the non-zero coefficients. For sparse matrix calculations, this approach has been very successful for tackling large sparse problems [43]. The difficulty is that sparse data structures induce memory overhead since locations of non-zero entries have to be stored alongside the numerical values. Furthermore, as the data structure is sometimes dynamically updated (the shape of the data structure evolves), the design of a compact representation in memory is difficult because one has to think that the data structure may extend in several ways. A crucial parameter of a sparse representation is the Number of Non-Zero entries (hereinafter abbreviated as *NNZ*). The *NNZ* quantity parametrizes the costs of algorithms working on the compressed representation.

Sparse representations has become the topic of an active field of research in both mathematics and engineering, providing a powerful tool for many applications. Developing, implementing and analyzing fast sparse algorithms for dedicated applications is challenging because it is related to multiple fields in the computer science area, such as computer architecture, compilation, data structures, algorithms, information theory, and applied mathematics. Here is a list of several objectives we would like to concentrate on, while designing a sparse wavelet representation:

- Reduce the number of operations when performing sparse computations (including the possible overhead due to sparse data management).
- Implement a compact representation in memory of sparse data structures.
- Obtain a multiresolution approximation: we want to possibly access to a given wavelet resolution without taking care of finer scales.
- Improve computer performance for managing sparse representations: use cache memory better, avoid random access in memory, maximize the number of operations done per processor cycle.

1.4 Pitfalls

1.4.1 Efficient data structures

Processor and memory technology trends show a continual increase in the cost of accessing main memory. Machine designers have tried to mitigate this through user-transparent mechanisms, such as multiple levels of cache, dynamic instruction scheduling, speculative execution or prefetching. Yet, a wide gap remains between the available and achieved performance of software, notably because of access time to main memory. Furthermore, today's compilers are unable to fully harness the complexity of processor architecture. Thereby, there is a need for efficient programming techniques and performance tuning strategies.

In this context, the task of redesigning and reorganizing data structures to improve cache locality should be considered, especially for pointer-manipulating programs. Pointer-based structures allow data to be placed in arbitrary locations in memory. This freedom enables a programmer to improve performance by applying data optimization techniques[1] in order to use caches better. Implementing *cache-conscious* data structures is a tricky task, but some simple techniques will be shown in this document. All data structures we will examine here are related to wavelet representation and are used in real-world applications. We will see that tuned data structures can save a lot of computer resources.

1.4.2 Designing correct algorithms

Designing the right algorithm for a given application is a difficult job [51]. In the field of wavelet-based application, the task is especially difficult because we mix sharp mathematics, with sparse representations in memory and a collection of tricky algorithms.

Generally, programmers seek algorithms that are correct and efficient, while being easy to implement (partly in order to reduce the debug cycle). These three goals may not be simultaneously achievable. In industrial settings, any program that seems to give good enough answers without being too slow is often acceptable, regardless of whether a better algorithm exists. The issue of finding the best possible answer, or achieving maximum efficiency, or even prove the correctness of the program, usually arises only after serious trouble.

One can expect that correct algorithms come with a proof of correctness,

[1]such as clustering or compression of data in memory.

which is an explanation of *why* we know that the algorithm must take every instance of the problem to the desired result. Looking for invariants that should remain true during problem resolution is an important task for the application designer. These invariants are quite different for each specific application or algorithm. It may require discussing with collaborators belonging to a specific application domain. Identifying some invariants helps both to debug and to give a proof of correctness. In wavelet applications, some commonly used invariants are *mass* conservation (*average* conservation), or *energy* conservation.

There is a real difficulty in programming wavelet-based applications. A robust implementation should provide some invariants or some quantities that can be checked.

1.4.3 Optimizing program performance

Modern compilers are sophisticated tools that are able to produce good code [6]. As programmers, we do not need to know the inner workings of the compiler in order to write efficient code. However, in order to make good coding decisions in our programs, we do need a basic understanding of machine-level code and how the compiler is most likely to translate code statements into machine code. For example, is an `if-else` statement costly or not? How much overhead is incurred by a function call, by a system call ? Are pointer references more efficient than array indexes? Why does our loop run so much faster if we sum into a local variable instead of an argument that is passed by reference?

Exercise 1. Answer to the previous questions.

Optimization of computer programs is a wide area that can not be covered here. We will however focus on some particular points that can be useful for wavelet applications.

2 Algorithmic and performance issues

The topics of algorithms and data structures cannot be separated since the two are inextricably intertwined [40]. So before we begin to talk about data structures, we must quickly look at some basic notions of algorithmics and, in particular, on how to characterize algorithm performance. The main issue in studying the efficiency of algorithms is the amount of resources they consume. It is usually measured in either the memory space or time consumed. There are classically two ways of measuring these quantities. The first one consists in a mathematical analysis of the algorithm under consideration, called an asymptotic analysis, which can capture gross aspects of efficiency for growing input sizes (but cannot predict exact execution times). The second is an empirical analysis of an actual implementation, to determine exact running times for a sample of specific inputs. Both measures will be explained in this section, and the same type of approach can be also employed to evaluate memory consumption. These measures characterize the performance of programs.

2.1 Computer architecture

2.1.1 Memory hierarchy

In a simple schematic view of a computer system, the central processing unit (CPU) executes instructions, and a memory system holds instructions and data for the CPU. In this view, the memory system is a linear array of bytes, and the CPU can access every memory location in a constant amount of time. The truth is quite more complex and this view does not reflect the way that modern systems really work.

In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times. CPU registers hold the most frequently used data. Small and fast cache memories nearby the CPU (named[2] L1, L2, L3) act as staging areas for a subset of the data and instructions initially-stored in the relatively slow main memory. The memory hierarchy works well because optimized programs tend to access the storage at any particular level more frequently than they access the storage at the next lower level. The storage at the next level can thus be slower, larger and cheaper per bit. The expected overall effect is the illusion of accessing a large pool of memory that costs as much as the cheap storage near the bottom of the hierarchy, but that serves data to programs at the rate of the fastest storage near the top of the hierarchy.

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data needed by your program are stored in a CPU register, then they can be accessed in zero or one *cycle*[3] during the execution of the instruction; if stored in a cache, from 1 to 30 cycles; if stored in main memory, from 50 to 200 cycles. Therefore, one can notice that a well written program that

[2]letter 'L' stands for cache level.

[3]CPU *cycle* refers to a single pulse of the processor clock.

often access data in cache memory will be much faster than another version of the program that mainly accesses data randomly in main memory. That work can not be made entirely by the compiler and the programmer should think accurately about the organisation of data in memory (also called data layout).

If you understand how the system moves data up and down the memory hierarchy, then you can write your application programs so that their data items are stored higher in the hierarchy, where the CPU can access them quicker. This idea centers around a fundamental property of computer programs known as *locality*. Programs with good locality: 1) tend to access the same set of data items over and over again, or 2) they tend to access sets of nearby data items. Programs with good locality tend to access more data items from the upper levels of the memory hierarchy (registers, L1 and L2 caches) than programs with poor locality, and thus run faster. For example, the running times of different matrix multiplication kernels that perform the same number of arithmetic operations, but have different degrees of locality, can vary by a factor of more than 20. This example is illustrative of a typical situation in scientific computing. We will emphasize this point in the matrix-matrix multiplication example (see section 3.1.2).

2.1.2 Cache matters

A computer spends a lot of time moving information from one place to another. As the processor runs a program, instructions are copied from main memory into the processor registers. From a programmer's perspective, much of this copying is overhead that slows down the "real work" of the program. Thus, a major goal for system and hardware designers is to make these copy operations run as fast as possible. Because of physical laws, larger storage devices are slower than smaller storage devices, and faster devices are more expensive to build than their slower counterparts. For example, a typical register file stores only a few hundred bytes of information, as opposed to billions of bytes in the main memory. However, the processor can read data from registers almost 100 times faster than from main memory. Even more troublesome, as semiconductor technology progresses over the years, this processor-memory gap still remains. Application programmers who are aware of cache memories can exploit them to improve the performance of their programs by a few orders of magnitude.

2.1.3 Data access: spatial locality, temporal locality

Temporal locality is the tendency of programs to use data items over and again during the course of their execution [5]. This is the founding principle behind caches. If a program uses an instruction or data variable, it is probably a good idea to keep that instruction or data item nearby in case the program wants it again in the near future.

One corresponding data-management policy or heuristic that exploits this type of locality is *demand-fetch*. In this policy, when the program *demands* an instruction or data item, the cache hardware or software *fetches* the item from memory and retains it in the cache. Before looking into memory for a particular data item, the cache is searched for a local copy that can already be there.

Spatial locality arises because of the tendency of programmers and compilers to cluster related objects together in the memory space. As a result of this, memory references within a narrow range of time tend also to be clustered. The classic example of this behavior is array processing, in which the elements of an array are processed sequentially, one right after the other: elements i and $i+1$ are adjacent in the memory space, element $i+1$ is usually processed immediately after element i. To tackle this behavior, a *look-ahead* strategy can be used. When instruction i (or data item i) is requested by the program, that datum should be brought into the processor core, and, in addition, instruction (or data) $i + 1$ should be brought in as well.

The use of *cache blocks*[4] is a *passive* form of exploiting spatial locality. Whenever a program requests a single data item, it results in the loading from main memory to the cache of the entire block to which the data item belongs. Another effective way to exploit spatial locality is *prefetching* (an *active* form). Prefetching is a mechanism in which the program asks for loading an instruction or a data item before the processor really needs it. Therefore, one can expect reduce memory latency to access this data in anticipating the needs. A possible strategy for prefteching uses the memory address with the associated recent history or requested addresses to speculate on which cache block(s) might be used next. An obvious hardware algorithm can be something like "always fetch next cache block", called *one-block look-ahead* prefetching.

[4]also referred to as "cache line".

2.1.4 Pointers, indirections

Every variable defined in a program is located at an unique location within the computer memory. This location has its own unique address, the *memory address.* A variable that stores a memory address is called a pointer. Accessing a variable's value by a pointer is called *indirection*, since the value of variable is accessed indirectly.

Dynamic data structures are those data structures whose size and shape can change during run-time. Linked-lists and binary trees are two common examples. In traditional imperative programming languages, pointers or indirections enable the existence of such dynamic structures. The common approach is to allocate storage to the individual nodes of such structures dynamically, maintaining the logical connection between these nodes via pointers or indirections. Once such a data structure goes through a sequence of updates (such as insert/delete or modify), it may get scattered accross the entire memory space; resulting in poor spatial locality and making poor usage of cache.

2.1.5 Data layout and algorithms that favor locality

If the application accesses instructions and data in a largely sequential manner, then a simple mechanism is all that is necessary to get temporal locality, and the performance results can be quite good. If the program behavior is more complex (say, for example, that accesses in memory are quite random), the performance can be hindered.

If the program accesses data that are distributed widely across the memory space, simple look-ahead or prefetching strategies are expected to be insufficient. For example, let us assume that your application contains a *linked list* as main data structure. Let us say this *list* is large enough not to fit into any of the caches of the machine. A walk through the *list* will possibly exhibit neither temporal nor spatial locality. Depending on the way the data has been filled, the performance of the walk may be very bad because of the big jumps in memory space between two consecutive cells in the list. The performance of the walk will be largely parametrized by the order of elements in the *linked list* together with the element locations in memory space.

To improve the execution time, at least three ways can be explored.

1. First, it is possible that the choice of a *linked list* as the main data

structure is not compulsory and can be revised. In this case, one can think of replacing the linked list with a data structure with a linear memory access pattern, such as an *array* or a *dynamic array* [66]. Compared to linked lists, *arrays* have faster indexing (constant time versus linear time) and typically faster traversal due to improved locality of reference. However, dynamic arrays require linear time to insert or delete an element at an arbitrary location, since all following elements must be moved, while linked lists can do this in constant time. Another alternative is the *chunk list*: instead of storing a single record in every node, store a small, constant size, array of records in every node. Tuning the number of elements per node can exhibit different performance characteristics: storing many elements per node has performance more like an array, storing few elements per node has performance more like a linked list.

2. A second solution, less invasive for the source code, consists in improving the locality of the linked list. One can periodically copy the *list* into a new one with the goal of reordering the records in memory. After such a copy, a walk through the copied linked list induces only contiguous accesses in memory. If we suppose that the list does not change very often, one can expect an improvement in spatial locality and performance by doing this. Modern memory systems are tuned to favor the access of contiguous areas of memory. In contrast, linked lists can actually be a little inefficient, since they tend to iterate through memory areas that are not adjacent. The copy/reorder algorithm is a cheap way to do a sort of list records according to memory addresses.

3. A third solution, simpler but a bit more technical, is *software prefetching*. Data prefetch occurs when a program requests a data from main memory *before* it is actually needed The processor that receives such a demand decides on his own to perform or not the prefetch. In the walk through the linked list, it would mean that we add a compiler specific built-in function (for example, `__builtin_prefetch` for gcc/icc compilers) in the loop, in order to preload next records of the list. The main objective of this strategy is to hide memory latency by requesting data before it is really needed. Albeit this third solution is quite simple (*i.e.* only one single statement to add to the code), one should noticed that it can hardly give better performance than solutions 1 and 2. Let us remark that if you activate some optimisations during compiling step, it can happen that prefetch is automatically turned on, and you may see no improvement at all inserting prefetch statements.

Timing results are given in Table 1 to compare the three solutions against the initial one. All tests were done on the same machine, an Intel Nehalem-EP node[5]. We consider here a program that performs a traversal (computing the sum of the data stored in records) on a linked list. The size of the list is 8M records (corresponding to 160MB in memory which is larger than the L3 cache). Source code is stored into the `linked-list` directory. The execution times of the traversal are given in Table 1. Solutions 1 or 2 lead to speedups larger than 10 compared to initial linked-list version. Solution 1 shows that the speedup of an optimized data structure can be as large as 15 over a simpler data structure.

	time (second)	speedup
Initial traversal (linked list)	0.84 s	1
Solution 1 (array)	0.056 s	15
Solution 2 (copy/reorder)	0.061 s	13.8
Solution 3 (prefetch statement)	0.82 s	1.02

Table 1: Time measurements for performing a sum over nodes stored in a linked list or in other data structures

The work we have done on the linked list can be transposed so as to improve access locality in other dynamic data structures (as trees, for example, as we will see afterwards). The lesson learned from this study is that changing a program's *data access pattern* or its *data layout* can radically modify spatial or temporal locality.

The careful placement of records in memory provides the essential mechanism for improving cache locality, and therefore performance. A *cache-conscious* data layout [57] places objects with high temporal/spatial affinity close to each other so they can reside in the same cache block. The design of optimized data-structures is often difficult for programmers because it requires:

- a complete understanding of an application's code and data structures,
- knowledge of the underlying cache architecture, something many programmers are unfamiliar with,
- significant rewriting of an application's code, which is time-consuming,
- eventually consulting a computer scientist.

[5]Xeon X5570, Bi-processor quad-core, 2.93 Mhz CPU frequency, 32 KB L1 data cache per core, 256 KB L2 cache per core, 16MB L3 cache per node.

2.2 Computational complexity

2.2.1 Asymptotic analysis

Given a program, its running time is parametrized by several things among which the program input (or the program parameters) plays a major role. Presumably, as input size increases, so does running time (for example think about the loading time of a small file [1 KB] or of a large file [100 GB] in a program). Consequently we often describe running time as a function of input data of size n, denoted $T(n)$. We expect our notion of time to be largely machine-independent, so rather than measuring in terms of elapsed running time, it is more convenient to measure basic steps that the algorithm makes (usually either the number of arithmetic/atomic operations executed, or the number of memory accesses). This will not exactly predict the true running time, since the way the program is written and the quality of the compiler affect performance. As soon as n is large enough, the $T(n)$ function will approximate the true running time by a multiplicative constant factor.

In numerical analysis, it is common to use a notation to describe the asymptotic behaviour of functions, called "*big O notation*" or "*Landau symbol*". This notation is used in two situations:

1. to describe how errors behave as some parameter such as a grid spacing h approaches 0,
2. to describe how the computational cost (such as number of operations or amount of memory space) behaves as the size of the problem input n tends towards ∞.

Hereafter, we give a minimal review of some useful asymptotic notations.

- $O(g(n))$: A function f is $O(g(n))$ iff there exist positive constants c and n_0 such that

$$\forall n \geq n_0, \quad 0 \leq f(n) \leq c\, g(n) \ .$$

- $\Theta(g(n))$: A function f is $\Theta(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that

$$\forall n \geq n_0, \quad 0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n) \ .$$

Example: $n^2 + n$ is $O(n^2)$ as $n \to \infty$, since for large n the quadratic term dominates the linear term.

Example: In sorting a list of n items, a naive algorithm takes $O(n^2)$ work and an optimal algorithm (e.g. *quicksort*) takes $O(n\,log(n))$ work. If you put restrictions on the items - *e.g.* they come from a finite alphabet - you can reduce this to $O(n)$ work (by using *bucket* sort, see [64]).

In modern computers, caches and fast registers induces a bias and these asymptotic estimates can be only roughly correlated with actual compute time. Nevertheless, they are a useful way to distinguish potentially *fast* algorithms from the rest.

Exercise 2. Implement a *bucket* sort [64] on an integer array of N elements. Benchmark the code and check the asymptotic behaviour in $O(N)$.

2.2.2 Renewal of the subject

More and more frequently, computer systems are called on to handle massive amounts of data, much more than can fit in their memory. In this context, the cost of an algorithm is more aptly measured in terms of the amount of data transferred between slow memory (disks, for example) and fast memory, rather than by the number of executed instructions. Similar issues arise when designing algorithms and data structures to exploit cache memory efficiently [16, 20]. It will remain a critical topic to use future generations of supercomputers [49].

A fairly new area of algorithm research called *cache-oblivious* and *cache-aware* algorithms has appeared [3]. The main idea of cache-oblivious algorithms is to achieve optimal use of caches on all levels of a memory hierarchy without knowledge of their size. Cache-oblivious algorithms should not be confused with cache-aware algorithms. Cache-aware algorithms and data structures explicitly depend on various hardware configuration parameters, such as cache size, while cache-oblivious algorithms do not. An example of cache-aware data structure is a B-tree which possesses an explicit parameter B which represents the size of a node[6] (we will develop further on this B-tree structure in another section). The main disadvantage of cache-aware algorithms is that they are based on the knowledge of the memory structure and size, which makes it difficult to port implementations from one architecture to another. Another problem is to adapt these algorithms to work with multiple levels of the memory hierarchy. It can require the tuning of parameters at each level. Cache-oblivious algorithms solve both problems in

[6]the memory footprint size of the node is parametrized by B.

avoiding the need of hardware parameters. Unfortunately they can be even more complex to implement than cache-aware algorithm.

3 Data structures for multiresolution

A small collection of dense and sparse data structures are introduced in this section. These data structures are instructive and will be useful example to design data structures for multiresolution. A set of simple algorithms allows for benchmarking these data types. If some data structure focus on improving performance during runtimes, other ones find ways to reduce memory footprint. Looking for an ideal sparse data structure that provides quick access, compact representation and ease of use, it is a hard way to go.

3.1 Multidimensional array

3.1.1 Memory Layout of 2D arrays and matrices

Basic Linear algebra is a common tool in scientific computing. Consequently there exists some standardization about how to perform operations on vectors and matrices. Thus, the layout for dense matrix storage is somehow always the same. Thereafter, the alternatives for storing a matrix in the C language are described. Assume that we would like to store a 2D array A (a square matrix) having 4 rows and 4 columns.

1. One method of storing the matrix is as an array of arrays (i.e., array of pointers). The matrix variable A is a pointer to an array of pointers (`double**` for example). Each element of the array of pointers points to an array storing a matrix row, as shown in Fig. 1(b). Note that although elements of each row are stored contiguously, the rows can be non-contiguous in memory.

2. The second method, which is common in the scientific computing community, is to store the entire matrix as one single contiguous block of memory. Variable A has the address of the first array element, *i.e.* the address of entry $A[0][0]$. Two flavors exist for this storage: row-major order and column-major order.

   Row-major order (Fig. 1(c)): If the matrix is stored in memory row-wise (elements of the first row followed by elements of the second row, and so on), the storage format is called row-major order. This is how 2D arrays are usually stored in the C language.

```
|----+----+----+----|
|  1 |  2 |  3 |  4 |
|----+----+----+----|
|  5 |  6 |  7 |  8 |
|----+----+----+----|
|  9 | 10 | 11 | 12 |
|----+----+----+----|
| 13 | 14 | 15 | 16 |
|----+----+----+----|
```

(a) Initial Matrix

```
|---|   |----+----+----+----|
|   |-->|  1 |  2 |  3 |  4 |
|---|   |----+----+----+----|
|   | .
|---| .
|   | .
|---|   |----+----+----+----|
|   |-->| 13 | 14 | 15 | 16 |
|---|   |----+----+----+----|
```

(b) Storage using indirections

```
|---+---+---+---+---+---+---+---+-----+----+----+----+----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ... | 13 | 14 | 15 | 16 |
|---+---+---+---+---+---+---+---+-----+----+----+----+----|
```

(c) Storage using row major order

```
|---+---+---+----+---+---+----+----+-----+---+---+----+----|
| 1 | 5 | 9 | 13 | 2 | 6 | 10 | 14 | ... | 4 | 8 | 12 | 16 |
|---+---+---+----+---+---+----+----+-----+---+---+----+----|
```

(d) Storage using column major order

Figure 1: matrix storage

Column-major order (Fig. 1(d)): On the other hand, if the matrix is stored column-wise (elements of the first column followed by elements of the second column, and so on), the storage format is called column-major order. This is how 2D arrays are stored in the Fortran language.

It is important for the programmer to be aware of how matrices are stored in memory while writing performance critical applications. Accessing the matrix elements in the wrong order can lead to poor locality of reference and, therefore, suboptimal performance. For example, if a matrix is stored in row-major format, but the elements are accessed column-wise by incrementing the column index, then memory accesses are *strided* [7], causing poor cache utilization.

3.1.2 Matrix-multiply example

Let's take a look at a few algorithms for multiplying large, dense matrices. Figure 2(a) exemplifies a simple matrix-multiplication algorithm on square matrices of size n^2 (matrices are stored in column-major order). To improve cache usage, a classical improvement is to rearrange calculation by blocking

[7] *stride* in memory between two successive elements is often the size of a row.

```
void mat_mul_basic(double*A, double*tB,
                   double*tC, int N) {
  register double sum;
  register int i,j,k; /* iterators */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++) {
      for (sum=0., k=0; k<N; k++) {
        sum += A[k+i*N]*tB[k+j*N];
      }
      tC[i+j*N] = sum;
    }
}
```

(a) Basic

```
void mat_mul_dgemm(double*A, double*tB,
                   double*tC, int N) {
  double alpha = 1.;
  double beta  = 0.;
#ifdef MKL
  char *notransp  = "N";
  char *transpos  = "T"; /* transpose */
  dgemm(transpos, notransp, &N, &N, &N,
        &alpha, A, &N, tB, &N, &beta,
        tC, &N);
#else
  cblas_dgemm(CblasColMajor, CblasTrans,
              CblasNoTrans, N, N, N, alpha,
              A, N, tB, N, beta, tC, N);
#endif
}
```

(b) BLAS library call

```
void mat_mul_block(double*A, double*tB,
                   double*tC, int N,
                   int blocki, int blockj,
                   int blockk) {
  register int i, j, k;
  /* block iterators */
  int          bki, bkj, bkk;
  /* upper bounds */
  int          maxi, maxj, maxk;
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      tC[i+j*N] = 0.;

  for (bki=0; bki<N; bki+=blocki) {
    maxi=MIN(N,bki+blocki);
    for (bkj=0; bkj<N; bkj+=blockj) {
      maxj=MIN(N,bkj+blockj);
      for (bkk=0; bkk<N; bkk+=blockk) {
        maxk=MIN(N,bkk+blockk);
        for (i=bki; i!=maxi; i++)
          for (j=bkj; j!=maxj; j++)
            for (k=bkk; k!=maxk; k++)
              tC[i+j*N] +=
                A[k+i*N]*tB[k+j*N];
      }
    }
  }
}
```

(c) Blocked

Figure 2: Some matrix-multiply source codes

the loops, see Fig. 2(c). We get six loops in the new version, instead of the initial three loops. The *loop blocking* technique allows for the reuse of the arrays by transforming the loops such that the transformed loops manipulate sub-matrices that fit into some target cache. The idea is to perform as many computations as possible on data already residing in some cache memory, and therefore to minimize the amount of transfer between the main memory and this cache. A third source code is presented in Fig. 2(b), which shows how to use a call to an external optimized routine to compute the matrix multiply. It uses a library (BLAS implementation) that performs basic linear algebra operations such as vector and matrix multiplications[8]. Please note that in the three matrix-multiply routines that are shown, identical operations are done on the same data. Only the sequence in which independent operations are performed and compiler optimizations are different.

[8]Basic Linear Algebra Subprograms (BLAS) is an **A**pplication **P**rogramming **I**nterface standard. Many implementations of this API are available, among which: ATLAS, GotoBLAS, Intel MKL, LAPACK …

Consider the simple algorithm of Fig. 2(a). If matrix B is bigger than the cache, one can figure out that for each `i` value (each line of A that we are considering) the entire matrix B is reloaded from memory to the cache. The reason for that is that, when we have finished to read the last elements of matrix B, the first elements of B have disappeared from the cache. So, the processor has to read matrix B again in order to put its elements into the cache.
Now, let's assume that the size of the blocking factors of the blocked algorithm (that is, the `blocki`, `blockj` and `blockk` parameters of Fig. 2(c)) are well chosen for the largest cache of the machine. One can expect that a B sub-matrix stored in the cache will be used again and again for each row of a A sub-matrix (along dimension k). The direct gain of using data of the B sub-matrix remaining in the cache is that we avoid reloading such data from memory and then we improve the mean memory access times compared to the basic version that reloads matrix B for each line of A. A new measure has been introduced recently that evaluates the number of `Memory Transfers` incurred by problem of size n. The notation used for this is $MT(n)$. For an algorithm to be efficient, the number of memory transfers should be as small as possible. Let us denote MT_{basic} and $MT_{blocked}$ the number of memory to cache transfers for the basic and blocked algorithms[9]. From the explanation above, it is clear that $MT_{blocked}(n) < MT_{basic}(n)$.

There are several ways to estimate the blocking factors for the algorithm of Fig. (2(c)). ATLAS library[10] has chosen, for example, to benchmark extensively the target machine in order to obtain them. During the installation of ATLAS on a machine, this software tunes itself to provide best performances.

Several time measurements of the three versions of the matrix-multiply were performed on a Intel Nehalem-EP node. Table 3.1.2 shows execution times for one matrix-multiplication of two matrices of size 2048^2 computed on a single core of the node. The performance is also evaluated in terms of FLOPS. A FLOPS is equal to one floating-point arithmetic operation per second. This unit of computer speed is frequently used in scientific computing community. To compute the number of GFLOPS ($= 10^9$ FLOPS) obtained, we need the number of floating point operations for a single matrix multiply. This number is the sum of 2048^3 additions and 2048^3 multiplications, so we get 2×2048^3 operations. For the Nehalem machine, the the-

[9]One can evaluate MT, for example if n is large enough: $MT_{basic}(n) = 2\,n^3 + n^2$, $MT_{blocked}(n) = \frac{2\,n^3}{m} + n^2$.

[10]Automatically Tuned Linear Algebra Software library, yet another implementation of BLAS.

oretical peak performance is 16 GFLOPS (double precision computations). In Table 3.1.2, performance of matrix-multiply subroutines are shown.

	Basic algo.	Blocked algo.	BLAS call
Time	19.3 s	5.66 s	1.68 s
GFLOPS	0.89	3.04	10.2

Table 2: Performance of a matrix-multiply on two square matrices of size 2048^2 on single core of a Nehalem node.

The main goal of this paragraph was to exemplify a cache optimization technique. The blocked algorithm is in-between the performances of the basic algorithm and the one of the optimized BLAS call. We notice that taking into account the cache issue with a blocked algorithm greatly improves performance. Then, to go from the blocked algorithm to the BLAS implementation, other optimizations should be done, such as multi-level cache blocking, innermost loop optimization, compiler best options choice, among other things. Yet, these elements are beyond the scope of this paper.

3.2 Dynamic array

Fixed-size arrays lack the possibility of being resizable, but provide very fast access to their elements. The goal of a *dynamic array* data structure is to give the possibility to resize the array whenever required, together with access performance close to the one of classical fixed-size arrays. The simplest dynamic array is constructed by first allocating a fixed-size array and then dividing it into two parts [66]: the first one stores the elements of the dynamic array and the second one is reserved (allocated but unused). We can then add elements at the end of the first part in constant time by using the reserved space, until this space is completely filled up. The number of elements currently stored in the dynamic array is its *actual size*, while the size of the underlying array in memory is called the *capacity*, which is the maximum possible size without reallocating. Resizing the underlying array is an expensive operation, typically it involves: allocating memory space, copying the entire contents of the array. To avoid incurring the cost of resizing many times, dynamic arrays usually resize by a large amount, such as doubling in size (geometric expansion), and use the reserved space for future expansion. The operation of adding an element to the end might work as follows:

```
inline void dynarray_append (dynarray *a, element *e) {
  if (a->size == a->capacity) {
    if (a->capacity == 0) a->capacity = 1;
    else a->capacity = 2 * a->capacity;

    // realloc: allocate new array and copy the contents of previous array
    a->array = realloc(a->array, a->capacity*sizeof(element));
  }
  a->array[a->size] = *e;
  (a->size)++;
}
```

Dynamic arrays benefit from many of the advantages of arrays, including good locality of reference and data cache utilization, compactness (low memory usage compared, for example, to linked list, which require extra pointers), and random access with no extra cost. They usually have only a reasonable additional overhead in term of memory footprint compared to fixed size arrays. This makes dynamic arrays an attractive tool for building *cache-friendly* data structures.

Benchmarks on the 8-core Nehalem-EP node have been made for different sizes of array. Depending of the array length, the array can fit in L1, L2 or L3 caches (respectively sized 8×32KB, 8×256KB, 2×8MB). Table 3 presents the measured bandwidth for reading *static arrays* or *dynamic arrays*. The benchmark has been performed with 8 threads working simultaneously on 8 cores, in order to saturate the node bandwidth capacity. The cumulative bandwidth given corresponds to the sum of thread bandwidths. An average on several successive runs has been performed. The caches are *not* flushed between two successive runs, thus increasing temporal locality.

Array length	Cumulative bandwidth static array	Cumulative bandwidth dynamic array	Known peak bandwidth
32 KB	370 GB/s	370 GB/s	-
64 KB	220 GB/s	220 GB/s	-
512 KB	150 GB/s	150 GB/s	-
8 MB	31 GB/s	31 GB/s	32 GB/s

Table 3: Cumulative bandwidth obtained during the computation of a sum of array elements (on Nehalem-EP node with 8 cores).

Dynamic and static arrays reach the same bandwidth reading. It is worth noticing that the cumulative bandwidth of L1 cache is 370 GB/s whereas the main memory bandwidth is 31 GB/s. The order of magnitude between these two bandwidths intuitively shows that it is critical to design algorithm and data structures that are able to use cache efficiently.

3.3 Hash table

3.3.1 Principle

In computer science, a hash table [67] or hash map is a data structure that uses a hash function to map identifying values, known as keys, (*e.g.* a person's name) to their associated values (*e.g.* his/her telephone number). The hash function is used to transform the key into the index of an array element. This array element (denoted the *bucket*) stores the associated value. Note that a hash table essentially has the same functionality as a *set*, allowing for element insertion, element deletion, and element retrieval. If the set is very small, a simple array or a linked list can be sufficient to represent a set. Yet, for large sets, these simple solutions will most probably require too much computing and memory resources whereas hash tables will save memory space and will be efficient. Many people consider that hash table are simple to use and fast.

Ideally, the hash function should map every possible key to a unique bucket index (or slot index), but this ideal is unfrequently achievable in practice. Most hash table designs assume that hash collisions (the situation where different keys happen to have the same hash slot/index) are normal occurrences and must be accommodated in some way. A classical approach to deal with that is to have a linked list of several key-value pairs that are *hashed* to the same location (*bucket*). By doing so, we avoid storing values directly in buckets, and we use per-bucket linked lists instead. To find a particular value, its key is hashed to find the index of a bucket, and the linked list stored into the bucket is scanned to find the exact key. These data structure that uses linked lists are called *chained hash tables*. They are popular because they require only basic data structures with simple algorithms, and they support the use of simple hash functions that may induce collisions (see example shown in Fig. 3).

In a good hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table resulting in a cost $O(1)$. Many hash tables allow arbitrary insertions and deletions of key-value pairs, also at constant average cost per operation. In many situations, hash tables are as fast as search trees or any other table lookup structures, and sometimes are even faster. For this reason, and their simplicity of use, they are widely used in many kinds of computer softwares.

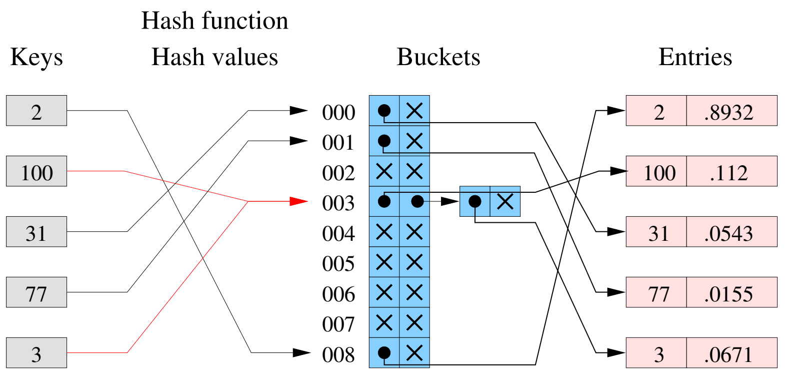


Figure 3: A chained hash table with five pairs (key-value) and one collision located at bucket (id 003).

3.3.2 Hash functions

In principle, a hashing function returns a bucket index directly; in practice, it is common to use the return value modulo the number of buckets as the actual index.

The chaining and the use of linked list represents a big overhead in a hash table. Therefore, it is important that the chains always remain short (that is, the number of collisions remains low). A good hashing function would ensure that it distributes keys uniformly accross the available buckets, thus reducing the probability of collisions [38, 67].

Another way to keep chains short is to use a technique known as dynamic hashing: adding more buckets when the existing buckets are all used (that is when collisions become inevitable, the hash table is resized up), and using a new hashing function that distributes keys uniformly into all of the buckets. All keys need to be redistributed and copied, since the corresponding indices will be different with the new hashing function.

3.3.3 Drawbacks

A hash table is a natural and quite easy way to represent a set or a sparse array. If the hash function is effective (that is, induces few collisions), then accessing an element in the array is effectively $O(1)$. However, achieving this level of performance is not simple. Furthermore, the constant before the $O(1)$ can be quite large. Thereafter is given a list of some drawbacks one can expect of hash tables:

- Choosing an effective hash function for a specific application is more an art than a science. Furthermore, if we choose a bad hash function, hash tables become quite inefficient because there are many collisions.

- Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly high.
- There is no efficient way to locate an entry whose key is closest to a given key. In situations where spatial correlations exist between neighboring keys, it can be penalizing. In comparison, ordered search trees have lookup and insertion cost proportional to $log(n)$, but allow finding the nearest key at about the same cost, and ordered enumeration of all entries at constant cost per entry.
- Hash tables in general exhibit poor spatial locality: the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger cache misses and slow down memory accesses.
- There is a significant runtime overhead in calling a subroutine [11], including passing the arguments, branching to the subprogram, saving local context, and branching back to the caller. The overhead often includes saving and restoring certain processor registers, allocating and deallocating memory space. So, if you use a library that furnishes an implementation of a hash table, you may pay a lot for all function calls made to the library subroutines. Thus, you may want to design a *home made* hash table, in order to reduce the costs associated by function calls (using *macros* or *inlining* techniques).

3.3.4 Traversal of a sparse array

In the `hash` directory, you will find a short program that benchmarks a hash table. The `glib` implementation of hash tables has been chosen [21]. This C utility library is portable across different operating systems. Few studies report performance of available hash table libraries, so our choice of `glib` as a reference library assume that it is a representative case.

The source code has user-defined *macros* to access the hash table (see `hash.h`). These macros look like functions but do not have the overhead associated with classical function calls[12]. Suppression of the overhead due

[11] A technique used to eliminate this overhead is called *inline expansion* or *inlining.*

[12] Macros looks like functions that are expanded and processed by the preprocessor, the compiler does not see them. Just before the compiler does its job, the preprocessor is

to function calls is required in order to analyse correctly the timing results for accessing the hash table. Furthermore, these macros provide a generic interface for transferring information between the application on the first hand and the hash table implementation on the other hand. Also, it provides an easy way to quickly change the hash table implementation: one just has to modify a few macros and few functions in `hash.h` and `hash.c` in order to use a new library or a home-made hash table.

In the directory `hash`, some *sparse array* implementations are benchmarked. Let us define a sparse array as a data structure where S is the number of items to store and N is the size of the range of key values k ($k \in [0, N-1]$). In the proposed test, the sparse array has an inverse fill ratio (the ratio of N over S which is a measure of sparsity) equal to $\alpha = \lfloor N/S \rfloor = 15$. Two implementations of sparse array are considered and several traversals are performed on them. The first implementation is based on dense array, most entries of which are not set (practically, all unset entries have a default value `NOTAVALUE`). The second implementation uses a `glib` hash table [21, 35]. The first solution won't save memory (space complexity is $O(N)$), while the second one will (space complexity is $O(S)$).

The test program initializes a dense array of size N by assigning a value to an entry over $\alpha = 15$ (the total number of assigned entries is equal to S). Also, the hash table is filled with the same S values. Once the initialization is done, we measure the execution time for summing all S values stored both in the dense array and in the hash table. To perform this sum, the three traversals are compared, with respect to execution time:

1. Iterate over the S records of the hash table, using a `glib` iterator (complexity in time $O(S)$).

2. Traverse the whole dense array, test each entry, and select only defined values (*i.e* the values not equal to `NOTAVALUE` - complexity in time $O(N)$).

3. Traverse the whole dense array and make a lookup to the hash table at each position and try to retrieve the associated value (complexity in time $O(N)$).

automatically called and performs simple text substitutions into the source code. Macros can be replaced by a function call with inline code. Macros and inline functions eliminate the overhead of a function call. Inline functions can be preferable to macros because they offer the type-checking services of functions and they avoid collisions in the naming of macro parameters and local variables.

4. Traverse the dense array by successive pseudo-random accesses and try to retrieve the associated value (complexity in time $O(N)$).

For the Nehalem machine that we previously described, the execution times of the two first solutions are quite the same (for $N = 8.10^7$). This means that the access time of an entry in the hash table via the iterator is 15 times larger than the access time of a single dense array access.
The execution time of the third traversal is approximately 20 times bigger than the second traversal time. We can therefore estimate that the lookup operator of the hash table has an average time cost which is 20 times the one of a single array access time (through sequential accesses). The fourth traversal time is almost the same as the third one, meaning that reading access of the hashtable is roughly equal to one random access time.

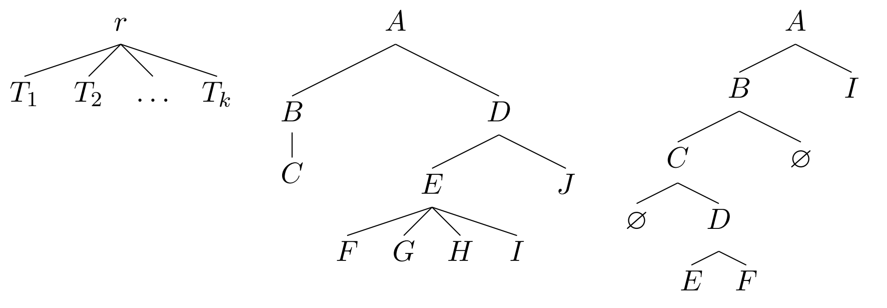
The same benchmark has been undertaken on other machines, and similar behaviour is observed: the access time to the hash table is between 13 and 30 times slower than an array access (a sequential access, not a random access). We can still improve this access time by designing a home-made implementation of hash table with specific features and optimized hash function for a target application. But the difference coming from a quick sequential access versus a slow random access will still remain anyway.

3.4 Tree

3.4.1 Definition

Formally a tree (actually a rooted tree) is defined recursively as follows [40]. In a tree, there is a set of items called nodes. At the top of the tree, one distinguished node is called the root r, and below one can find a set of zero or more nonempty subsets of nodes, denoted $T_1, T_2, \ldots, T_k$, where each T_i is itself a tree. These are called the *sub-trees* of the root.

The root of each sub-tree $T_1, T_2, \ldots, T_k$ is said to be a child of r, and r is the parent of each root. The children of r are said to be siblings of one another. Trees are typically drawn like graphs, where there is an edge (sometimes directed) from a node to each of its children. See figure 4(a).



(a) Root r and its k subtrees (b) Example of a general tree (c) Example of a binary tree

Figure 4: Tree examples

The degree of a node in a tree is the number of children it has. A leaf is a node of degree 0. A path between two nodes is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that u_i is a parent of u_{i+1}. The length of a path is the number of edges on the path (in this case $k-1$). The depth of a node in the tree is the length of the unique path from the root to that node. The root is at depth 0. The height of a node is the length of the longest path from the node to a leaf. Thus all leaves are at height 0. If there is a path from u to v we say that v is a descendants of u. Similarly, u is an ancestor of v.

3.4.2 Implementation

One difficulty in designing a data structure for general tree is that there is no bound on the number of children a node can have. A constraint is that each node must have access to pointers to all its children. A common representation of general trees is to store two pointers within each node: the *leftChild* and the *rightSibling* (see figure 5 at left). Another way to represent a node is to store an array of children (if the maximum number of children is known). On the right of figure 5, the data structure that uses the array of children simplifies the management of data structure. It avoids some pointers manipulation compared to the first solution with the chained nodes (that looks like linked-list). Nevertheless, memory space is wasted because a fixed size array is used whenever the tree actually has no child. The code corresponding to the general tree data structures shown here are stored in the `gtree` directory.

```
struct gtree1 {
 struct data   object;
 struct gtree1 *leftChild;
 struct gtree1 *rightSibling;
};
```

```
struct gtree2 {
 struct data   object;
 struct gtree2 childArray[MAX_NB_CHILD];
};
```

Figure 5: Data structures for general tree

To design an efficient data structure, one has to answer the following questions: May the number of children per node vary over a large range (possibly with no upper limit)? What kind of structure is taken to store dynamic list of children (*e.g.* linked-list, dynamic array)? Does the data representation permit: efficient search, node insertion, node deletion, quick traversal?

```
struct gtree3 {
 struct data object;
 int         leftChild;
 int         rightSibling;
 int         parent;
};
```

```
struct darray_gtree3 {
  struct gtree3 *nodesArray;
  int size;
  int capacity;
};
```

Figure 6: Efficient data structures for general trees

Depending on the data type, the performance of operations on the general tree may vary. In figure 6, an advanced data structure is shown. It can be implemented with several langages (including C and Fortran). The main idea is to store nodes of the tree in a dynamic array. This representation provides for a quick traversal of the tree in simply scanning the array of nodes (without following links between nodes). Indeed for some operations, it is not required to perform a postfix, prefix, breadth-first or another traversal specifically. Then, a simple scan of the array prevents one to perform a costly effective tree traversal that requires to follow the links between nodes. All these indirections are penalizing in term of execution time because they induce random accesses. The table 4 shows the memory bandwidth achieved using the simple `gtree1` and the sophisticated `gtree3` data structures[13] (using the same machine as before).

[13]The bandwidth were estimated using only the size of `object` field encompassed within a node which is here a field of size 8 Bytes (`double` type).

Nb nodes	Cumulative bandwidth **breadth-first algo. gtree1**	Cumulative bandwidth **breadth-first algo. gtree3**	Cumulative bandwidth **array traversal gtree3**
1K	4.4 GB/s	4.8 GB/s	90 GB/s
8K	2.9 GB/s	3.4 GB/s	60 GB/s
32K	1.2 GB/s	2.7 GB/s	36 GB/s
64K	0.6 GB/s	1.6 GB/s	12 GB/s

Table 4: Performance for traversal of different general tree data struture

Columns 2 and 3 give the performance of the breadth-first traversal of data structures `gtree1` and `gtree3` respectively. In column 4, the bandwidth of the array traversal in a `gtree3` tree is shown.
The *breadth-first algorithms* used to walk through `gtree1` and `gtree3` are identical. The main reason that explains a higher bandwidth in columns 3 than in column 2 is twofold:

1. in `gtree3`, the indirections are specified by means of 32-bit integers [14], whereas in `gtree1` the indirections use 64-bit pointers[15]. The larger integer type you use for the indirections, the larger bandwidth consumption (and execution time overhead) you get.

2. in `gtree3`, all nodes are in a relatively compact memory area (all nodes of the tree benefit from contiguous memory allocation); but in `gtree1` the nodes can be scattered over a large memory space because each node is allocated separately. Thus, much more random accesses can be generated in the `gtree1` case than in the `gtree3` case. It follows that the bandwidth of `gtree1` must be lower or equal to the bandwidth of `gtree3`.

Concerning column 4, the used algorithm is a loop that iterates through the array of nodes. Here, no indirection at all are followed and the access pattern in memory is contiguous. It is the first reason why this algorithm has larger memory bandwidth than the breadth-first algorithm of column 3. The second reason, is that in the array scan we avoid the use of many conditionals (`if` statements) that are needed in the breadth-first algorithm. And one has to notice that `if` statements can drastically slow down code.

[14]If you manipulate only trees with a few nodes you can even take 16-bit integer.
[15]The machine used for the benchmarks uses a 64-bit operating system.

Remark: When a processor sees a conditional instruction appear in the instruction stream, a mechanism called `branch prediction` is activated. Sometime, the processor can guess which branch of the `if` (`i.e.` the `then` part versus the `else` part) is taken. In this case, the program may keep going at full speed. However, if the processor does not guess beforehand the right branch of the conditional, there is a big performance hit. The reason behind that is that many processors partially overlap instructions, so that more than one instruction can be in-progress simultaneously. An internal engine called a *pipeline* takes advantage of the fact that every instruction can be divided into smaller identifiable steps, each of which uses different resources on the processor. At each processor cycle, the pipeline can contain several instructions in-progress. A bad guess of the `branch prediction` mechanism leads to clear out the pipeline and to prevent the execution of several instructions in parallel. As a consequence, a program tends to slow down its execution whenever conditional statements are encountered.

Considering wavelet algorithms specifically, breadth-first traversal of a tree is very often needed. Then, one may expect to find a way to perform breadth-first traversal with maximal performance. A proposal is made in the `gtree.c` example. Assume we have a `gtree3` data structure as input. A standard breath-first traversal is done; for each node encountered during the traversal, a copy of the node data is inserted in a new tree. In the end, the new tree has its nodes ordered the same way as a breadth-first traversal performed on the initial tree. Considering the new tree, a simple array scan (performance of column 4) will exactly mimic a breadth-first traversal without the cost induced by indirections (same idea as the compacted data-structures described in [57]).

3.4.3 Binary tree

The binary tree is a fundamental data structure in computer science. One of the most useful types of binary tree is a binary search tree, used for rapidly storing sorted data and rapidly retrieving stored data[16].

A binary tree is defined recursively as follows. A binary tree can be empty. Otherwise, a binary tree consists of a root node and two disjoint

[16]A binary search tree is a binary tree in which the value stored at each node of the tree is greater than the values stored in its left sub-tree and less than the values stored in its right sub-tree.

binary trees, called the *left* and *right* sub-trees. Notice that there is a difference between a node with a single left child, and one with a single right child. In the previous setting of general trees no distinction has been made between these two situations.

A binary tree can be *implicitly* stored as an array (this method is also called *Ahnentafel* list). We begin by storing the root node at index 0 in the array. We then store its left child at index 1 and its right child at index 2 The children of the node at index 1 are stored at indices 3 and 4 and the children of the node at index 2 are stored at indices 5 and 6. This can be generalized as follows: if a node is stored at index k then its left child is located at index $2k+1$ and its right child at $2k+2$. This form of implicit storage thus eliminates *overheads* of the tree structure[17] (indirection or pointer storage).

A quadtree is a tree data structure in which each internal node has exactly four children instead of two for the binary tree. Quadtrees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions.

3.4.4 Optimization of the tree structure

Many improvement can be done to adapt the data layout of trees to a particular use. In rearranging the nodes in memory, one can increase the spatial locality and temporal locality. Several authors have described cache-aware or cache-oblivious data structures in order to maximize cache effects and to reduce memory bandwidth requirements [32].
Depending on the usage one foresee for the tree data structure, one may also be interested in reducing the memory consumption. In the signal processing and image compression communities, many sophisticated tools have been developed for this purpose. Two main issues have been addressed:

1. pointer elimination by the way of a specific data structure that store a compact representation of the tree (like *significance map* for example, see p.34),

2. compression of the information carried by the node.

The B-tree is an example of an optimized tree structure [11, 65]. Let us assume that we have a binary tree. We transform it in a B-tree by encapsulating a set of nodes of the original binary tree into a big node in

[17]This implicit storage is really advantageous for trees that tend to be balanced.

the new B-tree. The large size of the nodes in a B-tree makes them very cache efficient. It reduces the average number of nodes we need to access when walking from the root node to a leaf, and thus the average number of indirections.
Each internal node of a B-tree will contain several objects. Usually, the number of objects is chosen to vary between B and 2B (where the constant B is fixed by the user).

In a B-tree implementation, one can also optimize memory storage. A classical approach is that each node contains an implicit binary tree (see the previous section on binary tree) along with pointers to the child nodes. In reducing the storage induced by pointers, the memory overhead due to the tree becomes quite small.

Remark: Another approach to the B-tree (but quite more complex) is to store the tree in memory according to a *recursive* layout called the *van Emde Boas* layout. Conceptually, the *recursive* nature of the decomposition permits to handle trees with *cache-oblivious* property. It means that whatever the capacity of cache memories is, this structure handles them efficiently (we do not need to know the effective cache size and tune any parameter like B). Many other cache-optimized tree structures exist and have been designed for specific applications.

3.5 Wavelet trees in image processing

3.5.1 Introduction to the EZW scheme

Embedded Zerotree wavelet (EZW) coding has been introduced by J. Shapiro [50]. It is an effective and computationally simple technique for image compression (see [10] for a review of other coding strategies). Designing new compression algorithms was a very active research topic in the 1990s. This scheme has interrupted the simultaneous progression of efficiency and complexity of algorithms proposed by other researchers of the community. Indeed, this technique not only is competitive in performance with the most complex techniques, but is also extremely fast in execution.

Furthermore, the EZW algorithm has the property that the bits of the encoded image are generated in order of importance. The bit stream begins with the largest coefficient and ends up with smallest ones. This technique, called embedded coding, is very useful for progressive image transmission and video compression. A compact multi-resolution representation is provided by this coding. The encoder can terminate the encoding at any point according to an initially prescribed target rate or target distortion metric.

Also, the decoder can cease decoding at any point in the bit stream in order to recover the image at a given resolution [44]. A major objective in a progressive transmission scheme is to select the most important information - which yields the largest distortion reduction - to be transmitted first. Since most of the coefficients will be zero or close to zero thanks to the wavelet representation[18], the spatial locations of the significant coefficients make up a large portion of the total size of a typical compressed image. The outline of the image compression process is the following: 1) Wavelet transform of the 2D image, 2) *Quantization* step (applying the EZW algorithm, see subsection 3.5.2), 3) Entropy coding that provides lossless compression (see subsection 3.5.3).

A main component of the EZW algorithm, as many other image compression schemes, is the *bit-plane* encoder. Such a bit-plane encoder takes as input a set of binary numbers (having the same size of n bits). The principle is to first select the most significant bit (n) of all binary numbers. Then, in a second pass, the encoder selects the $(n-1)$-th bit of all numbers. Iteratively, all j-th bits are considered until the lowest significant bit is reached. In that procedure, a *bit-plane* represents all the bits having the same position j in the considered set of binary numbers. The main advantage of this type of encoder is the possibility of decoding only first bit-planes during the decompression. The larger the number of discarded bit-planes, the higher the distortion due to a larger quantization error [63].

The SPIHT strategy (Set Partitioning in Hierarchical Trees) is a more recent work that shares many of the EZW characteristics.

3.5.2 EZW Algorithm

In the EZW algorithm, all bit-planes are encoded one after the other, beginning with the most significant bit and ending with the lowest bit [58]. In order to reduce the number of bits to encode, not all coefficients is encoded in each bit-plane (only significant coefficients above a given threshold are selected). The bit-plane encoding will be done in the so-called *refinement pass*. Because a partial subset of coefficients are encoded in each bit-plane, one has to specify also what the 2D locations of transmitted bits are in the refinement pass. These locations will be encoded during the *significance pass*.

In short, for each bit-plane j, the encoder sends two series of bits. The first one encodes the location of new significant coefficients we must add

[18]2D wavelet transform in the case of images.

(*significance pass*). The second one gives the j-th bit of all coefficients distinguished as significant (*refinement pass*). To start the algorithm, an initial threshold is set, for example $2^{j_{max}}$. A coefficient in the wavelet 2D decomposition is significant if its absolute value is greater than the threshold. For every pass j a threshold is chosen against which all the wavelet coefficients are measured (*e.g.* 2^j). If a wavelet coefficient is larger than the threshold, its location is put in a *subordinate list* of significant coefficients and "removed" from the image (we will no longer look at it in next significance pass). The *subordinate list* is an incremental list of coefficient locations, at each significant pass some locations are added to the list. When all the wavelet coefficients have been visited, the threshold is lowered (*e.g.* 2^{j-1}) and the image is scanned again.

During the *significance pass*, wavelet coefficients are analyzed (recall that coefficients already in the subordinate list are considered equal to zero) and labeled with the following rules:

- label **p** if the coefficient is significant and positive,
- label **n** if the coefficient is significant and negative,
- label **t** if a coefficient is not significant as well as all its descendants (Zero Tree root),
- label **z** if a coefficient is insignificant but all its descendants are not insignificant (Insignificant Zero).

The scanning of coefficients is performed in such a way that no child node is scanned before its parents (in order to detect easily Zero Tree Root and Insignificant Zero nodes).

For each pass (bit-plane j), a stream of bit is generated:

- First, a sequence of **p**, **n**, **t**, **z** letters is encoded (this stream is called *significance map*). It implicitly gives the locations where the new coefficients lie. This representation of the locations of new coefficients is far more compact compared to simply list the location indices in the image.
- Second, The j-th bit of all coefficients appearing in the *subordinate list* is extracted. This list of bits is encoded

An example of a stream generated with the EZW algorithm [58] is given in Fig. 3.5.2.

```
[Significance pass 1] pnztpttttztttttttptt
[Refinement   pass 1] 1010
[Significance pass 2] ztnptttttttt
[Refinement   pass 2] 100110
[Significance pass 3] zzzzzppnppnttnnptpttnttttttttptttptttttttttpttttttttttt
[Refinement   pass 3] 10011101111011011000
[Significance pass 4] zzzzzzztztznzzzzpttptpptpnptntttttptpnpppptttttptptttpnp
[Refinement   pass 4] 11011111011001000001110110100010010101100
[Significance pass 5] zzzzztzzzzztpzzzttpttttnptppttptttnppnttttpnnpttpttppttt
[Refinement   pass 5] 10111100110100010111110101101100100000000110110110011000111
```

Figure 7: Example of a stream output by EZW algorithm of a an image of size 8×8

The EZW scheme shows us how to overcome storage problems through the design of a combination of complex data structure and algorithms that minimize the size of the wavelet representation. But the stream we have obtained in Fig. 3.5.2 can be further compressed using a method called entropy coding.

3.5.3 Entropy coding

Entropy coding is the process of taking all the information generated by a process and compressing it losslessly into an output stream. It enables to represent a bit stream with a smaller size for storage or for transmission. This method is widely used for data compression.

One of the main types of entropy coding creates and assigns a unique prefix code to each unique symbol that occurs in the input stream. These entropy encoders then compress data by replacing each fixed-length input symbol by corresponding variable-length prefix codeword. The length of each codeword is approximately proportional to the negative logarithm of the probability. Therefore, the most common symbols use the shortest codes. Popular entropy coder are for example *Huffman coding*, or *Arithmetic coding*. Classically, arithmetic coding further compresses the stream generated by the EZW algorithm.

In this section we had a look to the EZW scheme that combines several compression techniques.

4 Some wavelet algorithms

Algorithms closely related to the discrete wavelet transform are shown in this section. In these algorithms, wavelet representations are stored into non-sparse data structures. Sparse data structures usage will be shown in the next section. Compactly supported wavelets are considered here, and the lifting scheme is employed [14, 15, 30, 55, 56].

4.1 Haar

4.1.1 Notations

The *Haar* transform is a very simple example of wavelet transform. This transform holds many of the common issues arising in wavelet transforms, and requires just a little bit of programming effort. Let us give some notations introduced in a founding paper entitled 'Building your own wavelets at home' [55].

Consider the sampled function $c^{\,n}$ of 2^n sample values $c^{\,n}_{\,k}$:

$$c^{\,n} = \{c^{\,n}_{\,k} \mid 0 \le k < 2^n\}\,.$$

Apply a difference and an average operator onto each pair $(c_{\,2k}, c_{\,2k+1})$, and denote the results by $d^{\,n-1}_{\,k}$ and $c^{\,n-1}_{\,k}$:

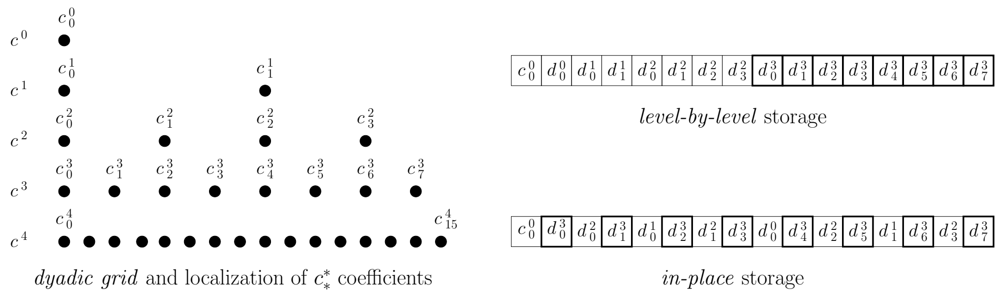
$$d^{\,n-1}_{\,k} = c^{\,n}_{\,2\,k+1} - c^{\,n}_{\,2\,k}\ ,\quad c^{\,n-1}_{\,k} = \frac{c^{\,n}_{\,2\,k} + c^{\,n}_{\,2\,k+1}}{2}\ . \tag{1}$$

or considering the *lifting* methodology [55], it can also be written:

$$d^{\,n-1}_{\,k} = c^{\,n}_{\,2\,k+1} - c^{\,n}_{\,2\,k}\ ,\quad c^{\,n-1}_{\,k} = c^{\,n}_{\,2\,k} + \frac{d^{\,n-1}_{\,k}}{2}\ . \tag{2}$$

The input function c_n is split into two parts: on one hand $d^{\,n-1}$ with 2^{n-1} differences, and on the other hand $c^{\,n-1}$ with 2^{n-1} averages. One can think of the averages $c^{\,n-1}$ as a coarser resolution representation of the input $c^{\,n}$, and $d^{\,n-1}$ as details that allow to recover $c^{\,n}$ from $c^{\,n-1}$. If the original function is very smoothly varying, then the details are small and can be cut off in order to get a more compact representation (we can store only coefficients $d^{\,n-1}$ above a given threshold).

We can apply the same transform iteratively on $c^{\,n-1}$, $c^{\,n-2}$, ... , $c^{\,1}$. We end up with the following representation: $c^{\,0}$ is the average of the input function, and detailed information is stored in $\left(d^{\,j}\right)_{j=0\dots n-1}$. This representation occupies the same space as the original input (N coefficients), but the difference is that we have a representation with many coefficient near 0 if the original function is smooth enough. The complexity of the transform computation is $\Theta(N)$. The localization of c_*^* coefficients are shown in Fig. 8 (left side).



c^0
c^1
c^2
c^3
c^4
c_0^0
c_0^1 c_1^1
c_0^2 c_1^2 c_2^2 c_3^2
c_0^3 c_1^3 c_2^3 c_3^3 c_4^3 c_5^3 c_6^3 c_7^3
c_0^4 c_{15}^4
dyadic grid and localization of c_*^* coefficients
c_0^0 d_0^0 d_0^1 d_1^1 d_0^2 d_1^2 d_2^2 d_3^2 d_0^3 d_1^3 d_2^3 d_3^3 d_4^3 d_5^3 d_6^3 d_7^3
level-by-level storage
c_0^0 d_0^3 d_0^2 d_1^3 d_0^1 d_2^3 d_1^2 d_3^3 d_0^0 d_4^3 d_2^2 d_5^3 d_1^1 d_6^3 d_3^2 d_7^3
in-place storage

Figure 8: Dyadic grid and two classic storage layouts in memory for the wavelet representation

4.1.2 Storage issue

Two type of data storage can be foreseen for the wavelet representation. On the first hand, we can store coefficients and details in increasing level order, from 0 to level $n-1$ (see Fig. 8, upper right side) and in each level coefficients are sorted by index. This way, we store contiguously all wavelet coefficients of the same level, one level after the other. In the literature, several notation are used for this storage layout: *level-by-level* or *mallat* representation [7, 36]. Another storage type called *in-place* is presented on Fig. 8, lower right side. This storage has become popular with the arrival of the wavelet *lifting* scheme [14, 55]. It mainly avoids moving of data in memory during the *forward* and *inverse* wavelet transform (as we shall see). For this ordering of data, coefficient d_k^j has the vector index $(1+2\,k)\,2^{j_{max}-j}$ (with $j_{max}=3$ in the example), wheras coefficient c_k^j has the vector index $k\,2^{j_{max}-j+1}$. For this storage setting: coefficient $c_{2\,k+1}^j$ lives at the same location as d_k^{j-1}, and coefficient $c_{2\,k}^j$ lives at the same index as c_k^{j-1}. A set of *in-place* algorithms allows for that kind of storage, during one single traversal they replace coefficients from one level by coefficients of another level.

The finest coefficients have been surrounded for the two storage layouts of Fig. 8. Let us remark that for the *level-by-level* solution all these finest coefficients are kept together, but in the *in-place* solution, they are scattered accross memory.

4.2 Forward Wavelet transform

```
void haar_fdwt(double* vec, int N) {
  register int itf, itc;
  register int idetail, icoarse;
  /* loop from finest level to coarsest level */
  for (itc = 2, itf = 1;
       itc <= N; itc *= 2, itf *= 2) {

    /* loop on all coefficients at this level */
    for (icoarse = 0;
         icoarse < N; icoarse += itc) {
      /* At index 'idetail': the difference
              at 'icoarse': the average */
      idetail        = icoarse + itf;
      /* PREDICT */
      vec[idetail]  =
        vec[idetail] - vec[icoarse];
      /* UPDATE */
      vec[icoarse] =
        vec[icoarse] + .5 * vec[idetail];
    }
  }
}
```

(a) In-place storage

```
void haar_fdwt(double* vec, double *ts, int N) {
  register int half, k;
  register int idetail, icoarse;
  /* loop from finest level to coarsest level */
  for (half = N/2; half >= 1; half /= 2) {
    /* Copy input 'vec' to tempory 'ts' */
    for (k = 0; k < 2*half; k++)
      ts[k] = vec[k];

    /* loop on all coefficients at this level */
    for (k = 0; k < half; k ++) {
      /* At index 'idetail': the difference
              at 'icoarse': the average */
      icoarse       = k;
      idetail       = half + k;
      /* PREDICT */
      vec[idetail]  = ts[2*k+1] - ts[2*k];
      /* UPDATE */
      vec[icoarse] =
        ts[2*k] + .5 * vec[idetail];
    }
  }
}
```

(b) Mallat representation

Figure 9: Forward wavelet transform - Haar

From Equation (2), one can deduce the algorithm of the Haar forward transform. For a detailed description on the design of this algorithm, we refer the reader to [55]. This algorithm is shown in Fig. 9 for the two storage layouts that has been just introduced. The input vector `vec` of size `N` is processed in order to provide the wavelet representation also in `vec` at the end of each function call. Let us notice that the *Mallat* representation requires a temporary array (parameter `ts` of the function). It is possible to avoid this supplementary storage at the expense of a reordering algorithm that represents a significant computational overhead.

4.3 Inverse wavelet transform

```
void haar_idwt(double* vec, int N) {
  register int itf, itc;
  register int idetail, icoarse;
  /* loop from coarsest level to finest level */
  for (itc = N, itf = N/2;
       itc >= 2; itc /= 2, itf /= 2) {
    /* loop on all coefficients at this level */
    for (icoarse = 0;
         icoarse < N; icoarse += itc) {

      /* At index 'idetail': the difference
              at 'icoarse': the average */
      idetail = icoarse + itf;
      /* UPDATE */
      vec[icoarse]
        = vec[icoarse] - .5 * vec[idetail];
      /* PREDICT */
      vec[idetail]
        = vec[idetail] + vec[icoarse];
    }
  }
}
```

(a) In-place storage

```
void haar_idwt(double* vec, double *ts, int N) {
  register int half, i;
  register int idetail, icoarse;
  /* loop from coarsest level to finest level */
  for (half = 1; half <= N/2; half *= 2) {
    /* loop on all coefficients at this level */
    for (i = 0; i < half; i++) {
      /* At index 'idetail' the detail will be
         store and at 'icoarse' is the average
         will be store */
      icoarse  = i;
      idetail  = half + i;
      /* UPDATE */
      ts[2*i]   =
        vec[icoarse] - .5 * vec[idetail];
      /* PREDICT */
      ts[2*i+1] =
        vec[idetail] + ts[2*i];
    }
    /* Copy tempory 'ts' to input 'vec' */
    for (i=0; i < 2*half; i++) vec[i] = ts[i];
  }
}
```

(b) Mallat storage

Figure 10: Inverse wavelet transform - Haar

The loop structures of the previous *forward transform* and of *inverse transform* are very similar as it is illustrated in Figure 10. But these loops differ in three ways:

1. the scanning of level goes from fine to coarse (forward transform) or coarse to fine (inverse transform),
2. the `update` and `predict` steps are not performed in the same order,
3. there is a sign modification in both `predict` and `update` steps.

With the Mallat storage, the array copy induces an overhead that does not exist in the in-place version. Nevertheless, on most machines execution times of forward and inverse transforms are quite the same with the two storage flavors. Besides the copy overhead, the Mallat storage induces more cache locality at the coarsest level. Sometimes, on some machines, the Mallat version is even faster than the in-place one.

4.4 Thresholding

To denoise or compress a signal, one can use a wavelet thresholding technique. It involves two steps: 1) taking the wavelet transform (*i.e.*, calculating the wavelet coefficients); 2) discarding (setting to zero) the coefficients with relatively small or insignificant magnitudes (using a specific criteria as we shall see). By discarding small coefficients, one actually avoids wavelet basis functions which have coefficients below a certain threshold. One can either set a global threshold which is the same for all coefficients, or set a level-dependent threshold. There is an extensive academic literature on the thresholding subject.

```
void haar_thresholding(double *vec, int N,
                       double normcoef,
                       double threshold) {
  int nnz_tot; /* number of non-zero coeff. */
  register int i;
  double *threshold_level;
  /* number of non-zero coeff. for each level */
  int    *nnz_level;
  /*  number of level, actual level */
  int     nblevel, level;
  nblevel         = (int)round(log2(N));
  threshold_level =
    (double*) malloc (nblevel*sizeof(double));
  nnz_level       =
    (int*) malloc (nblevel*sizeof(int));
  for (level = 0; level < nblevel; level++) {
    threshold_level[level] =
      threshold_func(threshold,norm,level);
    nnz_level[level] = 0;
  }
  for (i = 1; i < N; i +=1) {
    level = haar_level(i, nblevel);
    if (fabs(vec[i]) < threshold_level[level]) {
      vec[i] = 0.;
    } else {
      nnz_level[level]++;
    }
  }
  nnz_tot = 0;
  for (level = 0; level < nblevel; level++) {
    printf("level %4d threshold %20e nnz %10d\n",
           level,threshold_level[level],
           nnz_level[level]);
    nnz_tot += nnz_level[level];
  }
  printf("Number of non-zero coefficients :"\
         "%13d over %13d (%.7f percents)\n",
         nnz_tot,N,(100.*nnz_tot)/N);
  free(nnz_level);
  free(threshold_level);
}
```

(a) In-place storage

```
/* Thresholding of coefficients */
void haar_thresholding(double *vec, int N,
                       double norm,
                       double threshold) {
  register int level, i;
  /* number of non-zero coeff. at one level */
  register int nnz_level;
  int nnz_tot; /* total nb. of non-zero */
  register int half;
  register double threshold_level;
  nnz_tot = 0;
  for (level=0, half=1; half < N;
       half *= 2, level ++) {
    threshold_level =
      threshold_func(threshold,norm,level);
    nnz_level = 0;
    for (i = half; i < 2*half; i++) {
      if (fabs(vec[i]) < threshold_level) {
        vec[i] = 0.;
      } else {
        nnz_level++;
      }
    }
    printf("level %4d threshold %20e nnz %10d\n",
           level,threshold_level,nnz_level);
    nnz_tot += nnz_level;
  }
  printf("Number of non-zero coefficients :"\
         "%13d over %13d (%.7f percents)\n",
         nnz_tot,N,(100.*nnz_tot)/N);
}
```

(b) Mallat storage

Figure 11: Wavelet thresholding - Haar

5 Applications

Lifting was originally developed by Sweldens to adjust wavelet transforms to complex geometries and irregular sampling. A main feature of lifting is that it provides an entirely spatial-domain interpretation of the transform, as opposed to the more traditional frequency-domain based constructions. In this framework, algorithms can be adapted in order to exploit spatial locality and to boost performance in reducing the amount of data transfers (memory loads from random access memory to cache memory). We will use this formalism to illustrate the use of some sparse data structures in wavelet-based applications. A short analysis of the performance one can achieved for some 2D discrete wavelet transforms will be given. To begin with, a short overview of the lifting scheme is presented here; much of notations and materials come from [30].

5.1 Lifting overview

Lifting scheme consists of iteration of the following three operations:

- **Split** Divide the original 1D data vector $x[i]$ into two disjoint subsets. Even and odd indices are distinguished. The even index points are stored into x_e data set and odd points into x_o data set, such as

$$x_e[i] = x[2\,i], \quad x_o[i] = x[2\,i+1]$$

- **Predict** Generate the wavelet coefficients $d[i]$ as the error in predicting $x_o[i]$ from $x_e[*]$ using prediction operator $\mathcal{P}$ translated at index n:

$$d[i] = x_o[i] - \mathcal{P}(x_e[*])[i]$$

  If the underlying sampled function is locally smooth, the residuals $d[i]$ will be small.

- **Update** Combine $x_e[i]$ and $d[*]$ to obtain scaling coefficients $c[i]$ that represent a coarse approximation to the original vector $x[i]$. This is accomplished by applying an update operator $\mathcal{U}$ translated at location n to the wavelet coefficients and adding the result to $x_e[i]$:

$$c[i] = x_e[i] + \mathcal{U}(d[*])[i]$$

Because each transforms are invertible, no information is lost during the *split-predict-update* sequence. Going backward, one can reconstruct perfectly the initial data vector $x[*]$ thanks to $c[*]$ and $d[*]$ vectors. To do that inverse transform, one has just to apply a mirrored *update-predict-merge* sequence using the same $\mathcal{U}$ and $\mathcal{P}$ operators as in the forward transform. A simple example of lifting is the construction of the Deslauriers-Dubuc family of wavelets. In this transform, a single prediction step is followed by a single stage update step (boundary conditions are not discussed here):

$$d[i] = x_o[i] - \frac{-x_e[i-1] + 9\,x_e[i] + 9\,x_e[i+1] - x_e[i+2]}{16} \ ,$$

$$c[i] = x_e[i] + \frac{d[i-1] + d[i]}{4} \ .$$

The prediction and update operators are completly defined by a set of *filter coefficients*, respectively p_k and u_k, as follows:

$$\mathcal{P}(x[*])[i] = \textstyle\sum_k p_k\, x[i+k] \qquad (3)$$

$$\mathcal{U}(x[*])[i] = \textstyle\sum_k u_k\, x[i+k] \qquad (4)$$

For the Deslauriers-Dubuc example, the coefficients for predict and update operators are

$$(k, p_k) \in \left\{(-1, \tfrac{1}{16}), (0, \tfrac{9}{16}), (1, \tfrac{9}{16}), (2, -\tfrac{1}{16})\right\} \ ,$$

$$(k, u_k) \in \left\{(-1, \tfrac{1}{4}), (0, \tfrac{1}{4}), \right\} \ .$$

The Haar transform, that has been shown previously, fits easily in this lifting methodology, as many other wavelet filters do [15]. The filter coefficients of the Haar transform introduced previously (p. 36) are:

$$(k, p_k) \in \{(0, 1)\} \ , \qquad (k, u_k) \in \left\{(0, \tfrac{1}{2})\right\} \ .$$

5.2 Implementation of a 2D wavelet transform

Traditional wavelet-based adaptive methods can be further classified either as wavelet-Galerkin methods [28] or wavelet-collocation methods [27, 62]. The major difference between these two is that wavelet-Galerkin algorithms solve the problem in the wavelet coefficient space and, in general, can be considered as gridless methods. On the other hand, wavelet-collocation methods solve the problem in the physical space, using a dynamically adaptive grid.

The approach used in the following is known as adaptive wavelet-collocation method (AWCM). Wavelet coefficients are found based on the values of a function at certain locations (grid points), called collocation points. In typical algorithms, a set of collocation points is defined in such a way that the collocation points of the coarser level of resolution is a subset of the collocation points of the finer resolution. Often enough, these methods utilize interpolating wavelets instead of other types of wavelets [9, 19, 26, 27, 29, 34, 39, 56, 59, 61, 62]. In this framework, wavelets are typically used to adapt the computational grid (and hence compress the solution), while finite differences are used to approximate derivatives, if ever needed. Mathematically, interpolating wavelets can be formulated in a lifting biorthogonal setting.

Multi-dimensional wavelets can be constructed as tensor products of corresponding one-dimensional wavelets. It yields a separable multi-dimensional transform. On a 2D array, a two-dimensional wavelet transform is then simple. The tensor-product transform amounts to first computing one-dimensional wavelet transforms of each row of the input, collecting the resulting coefficients into the same data structure, and finally computing the one-dimensional transform of each column. An implementation of a 2D wavelet transform is presented in Figures (12,13,14).

```
/*
  Wavelet transform.
  In-place wavelet algorithm is used here.
  A dense 2D array (fwav) of N*N elements is
  passed to the 2D transform function.
  The result of the wavelet scheme is calculated
  in-place (without any array temporaries) in the
  array 'fwav'.
*/
void wav_ftransform( _VALTYPE *fwav, int N ) {
  int itc;
  for (itc = 2; itc <= N; itc = itc * 2) {
    wav_predict_x  ( fwav, itc, N, 1 );
    wav_update_x   ( fwav, itc, N, 1 );
    wav_predict_y  ( fwav, itc, N, 1 );
    wav_update_y   ( fwav, itc, N, 1 );
  }
}
```

(a) Forward transform

```
/*
  Wavelet inverse transform.
  Same comments as  wav_ftransform function
*/
void wav_itransform( _VALTYPE *fwav, int N ) {
  int itc;
  for (itc = N; itc >= 2; itc = itc / 2) {
    wav_update_y   ( fwav, itc, N, -1 );
    wav_predict_y  ( fwav, itc, N, -1 );
    wav_update_x   ( fwav, itc, N, -1 );
    wav_predict_x  ( fwav, itc, N, -1 );
  }
}
```

(b) Inverse transform

Figure 12: 2D wavelet transform - Lifting scheme plus tensorial product

The principle of the forward 2D transform algorithm is to perform a traversal of each level j from the finest to the coarsest in both directions. For the sake of simplicity, the implementation shown here uses periodic boundary conditions and a square domain (N grid points in each direction). We assume the number of wavelet levels $j_{max} = log(N) - 1$ to be equal in each direction.

We introduce some variables $itc(j)$ and $itf(j)$ associated to each level j: $itc = 2^{j_{max}-j+1}$ and $itf = 2^{j_{max}-j}$. Variable itc stores the distance along one direction between two consecutive details of level j. In subsection 4.1.2, that introduces storage issue, we have already see where are stored details d_k^j in the 1D case (in-place storage). The index of detail d_k^j in a 1D vector is $(1+2\,k)\,2^{j_{max}-j}$; it is straightforward that this index can be also expressed as: $itf + k \times itc$. To traverse all details of one level j, it is computationally cheaper to use this last expression. In the source code of Fig. 13(a), the loop on `ix` index uses it explicitly. Also, in the forward and inverse transforms of Fig. 12, the main loop that scans all levels is done on variable itc instead of j for convenience.

```c
/* Predict function in x-direction
   fwav: input/output array
   itc: level+1 which is considered
   N: square root of the 2D domain size
   dir: (1=forward) or (-1=inverse) transform
   wav_filterPred: predict filter
   wav_np: size of the predict filter
 */
inline void wav_predict_x(_VALTYPE* fwav, int itc,
                          int N, int dir ) {
  register int ix, iy, icoarse;
  register const int itf = (itc>>1); /* itc/2 */
  register int k, Nplus = N*N;
  register double predictVal;
  /* Scan all iy at level 'itf' */
  for (iy = 0; iy < N; iy += itf) {
    /* Apply predict operator in x-direction */
    for (ix = itf; ix < N; ix += itc) {
      /* At index ix is the detail point we are
         looking at, compute predict value */
      predictVal = 0.;
      for (k = 0; k < wav_np; k++) {
        icoarse = (Nplus + ix +
                   (k + wav_offp)*itc - itf)%N;
        predictVal += wav_filterPred[k] *
          fwav[_XY(icoarse,iy,N)];
      }
      /* Add or retrieve the predict value to
         the 'odd' points whether the forward
         or inverse transform is applied */
      fwav[_XY(ix,iy,N)] += - dir * predictVal;
    }
  }
}
```

(a) Predict along x-direction

```c
/* Update function in x-direction
   fwav: input/output array
   itc: level+1 which is considered
   N: square root of the 2D domain size
   dir: (1=forward) or (-1=inverse) transform
   wav_filterPred: predict filter
   wav_np: size of the predict filter
*/
inline void wav_update_x(_VALTYPE* fwav, int itc,
                         int N, int dir ) {
  register int ix, iy, idetail;
  register const int itf = (itc>>1); /* itc/2 */
  register int k, Nplus = N*N;
  register double updateVal;
  /* Scan all iy at level 'itf' */
  for (iy = 0; iy < N; iy += itf) {
    /* Apply update operator in x-direction */
    for (ix = 0; ix < N; ix += itc) {
      /* At index ix is the coarse point we
         are looking at, compute udapte value */
      updateVal = 0.;
      for (k = 0; k < wav_nu; k++) {
        idetail = (Nplus + ix +
                   (k + wav_offu)*itc - itf)%N;
        updateVal += wav_filterUpd[k] *
          fwav[_XY(idetail,iy,N)];
      }
      /* Add or retrieve the update value to
         the 'even'points whether the forward
         or inverse transform is applied */
      fwav[_XY(ix,iy,N)] += dir * updateVal;
    }
  }
}
```

(b) Update along x-direction

Figure 13: Predict and update operators in x-direction

Concerning the predict operator (x-direction), shown in Fig. (13(a)), one can see two loops in `ix` and `iy`. These loops perform a traversal of all details (in x-direction) over the whole 2D domain at one given level $j_x = j_{max} + 1 - log(itc)$. The inner loop in k apply the *predictor* operator $\mathcal{P}$ using: 1) filter coefficients stored into `wav_filterPred`, and 2) coarse coefficients indexed by `icoarse`. The size of the predict filter is equal to `wav_np`, and an offset `wav_offp` is used in order to translate the filter operator at the right position. All these parameters concerning the predict filter are set at the initialization stage depending on the effective filter the user wants.

The skeleton of the update operator in x-direction (Figure 13(b)) is similar to the predict operator. Nevertheless, there remain two main differences:

first, the update phase uses its own filter `wav_filterUp`; second, the inner loop accesses to fine coefficients indexed by `idetail` instead of coarse coefficients.

Algorithms presented on Fig. 14 illustrate the predict/update steps at one level `itc` in y-direction. They match the corresponding algorithms in x-direction except a few differences.

```
/* Predict function in y-direction
   fwav: input/output array
   itc: level+1 which is considered
   N: square root of the 2D domain size
   dir: (1=forward) or (-1=inverse) transform
   wav_filterPred: predict filter
   wav_np: size of the predict filter
*/
inline void wav_predict_y(_VALTYPE* fwav, int itc,
                          int N, int dir ) {
  register int ix, iy, icoarse;
  register const int itf = (itc>>1); /* itc/2 */
  register int k, Nplus = N*N;
  register double predictVal;

  /* Apply predict operator in y-direction */
  for (iy = itf; iy < N; iy += itc) {
    /* Scan all ix at level 'itf' */
    for (ix = 0; ix < N; ix += itf) {
      /* At index iy is the detail point we are
         looking at, compute predict value */
      predictVal = 0.;
      for (k = 0; k < wav_np; k++) {
        icoarse = (Nplus + iy +
                   (k + wav_offp)*itc - itf)%N;
        predictVal += wav_filterPred[k] *
          fwav[_XY(ix,icoarse,N)];
      }
      /* Add or retrieve the predict value to
         the 'odd' points whether the forward
         or inverse transform is applied */
      fwav[_XY(ix,iy,N)] += - dir * predictVal;
    }
  }
}
```

(a) Predict along y-direction

```
/* Update function in y-direction
   fwav: input/output array
   itc: level+1 which is considered
   N: square root of the 2D domain size
   dir: (1=forward) or (-1=inverse) transform
   wav_filterUpd: update filter
   wav_nu: size of the update filter
*/
inline void wav_update_y(_VALTYPE* fwav, int itc,
                         int N, int dir ) {
  register int ix, iy, idetail;
  register const int itf = (itc>>1); /* itc/2 */
  register int k, Nplus = N*N;
  register double updateVal;

  /* Apply predict operator in y-direction */
  for (iy = 0; iy < N; iy += itc) {
    /* Scan all ix at level 'itf' */
    for (ix = 0; ix < N; ix += itf) {
      /* At index iy is the coarse point we
         are looking at, compute update value */
      updateVal = 0.;
      for (k = 0; k < wav_nu; k++) {
        idetail = (Nplus + iy +
                   (k + wav_offu)*itc - itf)%N;
        updateVal += wav_filterUpd[k] *
          fwav[_XY(ix,idetail,N)];
      }
      /* Add or retrieve the update value to
         the 'even' points whether the forward
         or inverse transform is applied */
      fwav[_XY(ix,iy,N)] += dir * updateVal;
    }
  }
}
```

(b) Update along y-direction

Figure 14: Predict and update operators in y-direction

The setting of interpolating wavelets within the lifting framework is described in [56]. Example of the effective predict and update filters to use are also given.

5.3 Adaptive remeshing in 2D

Wavelet-based scheme can be used to solve partial differential equations (PDE) that involve different spatial and temporal scales [61]. The adaptive framework helps modelling time-evolution problems with multilevel features. Wavelets have the ability to dynamically adapt the computational grid to local structures of the solution. Wavelets appear to combine the advantages of both spectral (Fourier space solver) and finite-difference bases. One can expext that numerical methods based on wavelets attain both good spatial and spectral resolution.

For time-evolution problems, numerical schemes must be able to catch all structures that change over time. In the following, we assume that we are using an explicit time-marching scheme. In order to minimize the number of unknowns, the basis of active wavelets and, consequently, the computational grid should adapt dynamically in time to reflect changes in the solution between two consecutive time steps. The adaptation of the computational grid is based on the analysis of wavelets structure. We may drop fine scale wavelets with smallest coefficients in the regions where the solution is smooth. As a consequence, a grid point should be omitted from the computational domain, if the associated wavelet is omitted from the wavelet approximation. To ensure accuracy, the basis should also consider wavelets whose coefficients can possibly become significant during one time step of the time-integration scheme. Thus, at any instant of time, the basis should not only include wavelets whose coefficients are above a prescribed threshold parameter, but also include wavelets belonging to an adjacent zone. The size of the adjacent zone is determined specifically depending on the problem to be solved [26, 27, 59]. The method can be viewed as an adaptive mesh method, where the mesh is automatically refined around sharp variations of the solution. An advantage of the method is that we never have to care about where and how to refine the mesh. All that is handled by thresholding the wavelet coefficients. This method is well suited for large problems with a solution that is largely compressed in a wavelet basis.

In wavelet-based application with dynamic adaptive remeshing, one can use both the physical representation of the signal and its wavelet representation. An operator can be cheaper to compute in one of these two representations. With this approach one may transform back and forth between the physical domain and the wavelet domain at each time step, which however introduces some overhead. In this context, a specific computation kernel is required: the *adaptive* wavelet transform (with two flavors *forward* and *inverse*). This algorithm has the ability to apply the wavelet transform to a sparse representation of a signal. A sketch of a global algorithm using time-marching scheme and adaptive remeshing is shown in Fig. 15.

```
Initialize input signal $f$;
$W_{old} \leftarrow$ THR(FWT($f$)) ;          // Get wavelet coeff. and threshold them
for $n = 1$ to $nb\_time\_steps$ do
    $M_{new} \leftarrow$ ENRICH($W_{old}$) ;                   // Compute a new adaptive mesh
    $M_{new} \leftarrow$ TREE($M_{new}$) ;                    // Build a proper tree structure
    $F_{new} \leftarrow$ EVOLVE($M_{new}, F_{old}, W_{old}$) ;        // Apply time-marching scheme
    $W_{new} \leftarrow$ AFWT($F_{new}$) ;                    // Adaptive wavelet transform
    $W_{new} \leftarrow$ THR($W_{new}$) ;                                   // Threshold
    Swap $W_{new}$ and $W_{old}$;
    Swap $F_{new}$ and $F_{old}$;
```

Figure 15: A time-marching scheme with wavelet-based adaptive remeshing

In this algorithm, F_{new} (as F_{old}) refers to a set of active points and the corresponding signal values, W_{new} (as W_{old}) the same set of active points and the associated wavelet coefficients, and finally M_{new} a set of locations where we want to know wavelet coefficients (*i.e.* the adaptive mesh). Let us describe the algorithm. Initially, a forward wavelet transform (FWT) is applied on the input signal which is known on a fine uniform mesh. The resulting wavelet representation is thresholded (THR) and the remaining set of wavelet coefficients plus their locations are stored into a sparse data structure: W_{old}. Next, we enter the loop over time. In order to capture all physical phenomena that can appear during the next time step, the set of points included in W_{old} is enriched (ENRICH operator). It accounts for possible translation or creation of finer scales in the signal between two subsequent time steps. The TREE step adds the relevant grid points in order to get a *proper* wavelet tree structure [9]. The TREE operation will assure [19] us that for each point P stored into M_{new}: all the points at coarser scales that will be required to compute the wavelet coefficient at P are also in M_{new}. This is quite essential in order to simplify the adaptive forward transform (AFWT) that will be performed afterwards (and sometimes also to simplify the EVOLVE operator). Then, the evolution operator (EVOLVE) computes the values of the signal at next time step at the locations stored into M_{new} (the operator can use values in F_{old} and in W_{old}). The adaptive wavelet transform (AFWT) evaluates all wavelet coefficients W_{new} thanks to the set of signal values contained into F_{new}. The thresholding step (THR) remove coefficients in W_{new} that are above a level dependent threshold. Finally, at

[19] Note that it is possible to design applications without this feature ([27]), but recursive and costly interpolations using coarser scales may happen during the adaptive forward transform.

the end of loop over time, some data structures are swapped in order to prepare a new time step.

5.4 Sparse data structures

We want to use wavelets to obtain sparse representations of signals, so we need a data structure to store wavelet data that takes advantage of this sparsity [28]. This is especially important for adaptive solving of PDEs because one hopes to save both memory and computation. It is expected that the storage amount will be, in average, linear with respect to the number of stored values in the sparse representation. Assuming the memory required to store S coefficients in the sparse structure is approximately $c_s\, S$, we would like c_s as small as possible.

We also need an efficient representation of this sparse data structure, in order to keep low the overheads due to sparsity management. There are several criteria that should be met. As we just said, the data structure must induce a minimal overhead in term of memory storage, but it must also permit fast accesses to the wavelet coefficients. Nevertheless, both objectives are contradictory to a certain extent. Often, sparse data types will either propose to reduce the memory overheads and give a very compact memory footprint (for example EZW coding), or, on the other side, put forward very fast access to stored data. Lastly, it is desirable that the data structure will be simple to use.

In order to compare a wavelet-based method against a standard dense scheme, the design of the wavelet data structure is a rightfully key issue. Even if the algorithmic complexity gives a cheaper cost to wavelet-based method ($O(S)$ with S the number of wavelet coefficients) compared to non-sparse one ($O(N)$ with N the number of grid points), a wrong choice of the sparse data structure may drastically hinder performance of the wavelet approach. One essential point is that dense schemes have a real big advantage over wavelet scheme, they can use arrays as main data structures. Now, reading or writing sequentially dense data structures such as arrays is expected to reach near the maximum available main memory bandwidth. So, designing sparse data structures that can compete with this sequential access pattern on large vectors is quite a difficult task.

In the sequel, two sparse data structures will be compared: hash table, versus tree-based data type.

Tree data structure As stated in [28] and reported in subsection 3.4.4 (p. 31), using non-well-designed trees can lead to significant resource waste. With straightforward tree structures, much indirection pointers or logical indirections are needed that increase the total memory cost. Besides, accesses to deepest leaves necessitate visiting many nodes of the tree[20], hence increasing the average cost access to stored data.

A standard strategy to reduce these costs is to gather nodes of the tree into blocks of several nodes, sometimes calls supernode (block of B wavelet coefficients is chosen here). The reason behind choosing a block representation is to spread the cost of memory management and pointer references over all wavelet coefficients inside the block. In the following, we will denote by *packed Tree* or *P-tree* this type of tree that uses the same kind of strategy as the *B-tree* data structure. The number of wavelet coefficients in each block, B, has to be chosen with two competing aims in mind. If on one hand B is small enough one can get a tree with no zero-valued wavelets inside each block (in the extreme case $B = 1$, this property is always true), but on the other hand if B is large we get another benefit: less memory overhead due to indirections. So, B has to be optimized depending on the data to be stored, in order to both reduce sparsity inside blocks and the global overhead due to indirections.

At least two implementation alternatives may be foreseen for this P-tree data structure. On the first hand, a block of the P-tree can gather multiple nodes at one level of resolution. On the second hand, one can put nodes of the same spatial area but belonging to different levels in a block of the P-tree. In order to reduce the maximal depth of leaves, we have chosen to group wavelet coefficients of different levels together. With this approach, we are able to guarantee that the wavelet tree will have a very small depth. This property allows for very fast access to leaves by reducing the number of indirections to follow [24, 25]. In the following, the P-tree data structure is defined with only two levels of nodes. One node gathers all coefficients of coarsest levels from level 0 to level $j_{limit-1}$ over all spatial domain. Then, nodes of finer levels encapsulate coefficients of levels from j_{limit} to j_{max}. A fine node has a fixed size of $(2^{max-limit})^2$ and represent a square region in the 2D index space. If one coefficient is non-zero in a fine node, the node has to exist and coarser levels is connected to this node through one indirection. But if a fine node does not carry any non-zero coefficient, the node is not allocated at all. This type of trees in which some of the branches or nodes are discarded are called *pruned tree*. The cost to read or write a coarse element in the sparse structure consists in just one memory access. And it costs two memory accesses to read an element in fine blocks because one

[20]the number of visited nodes is exactly the depth of a given leaf.

has to follow one indirection [24]. Nevertheless, coarse node and fine nodes contain frequently zero values; so, the management of the sparsity is not optimal. This sparse structure with one level of indirection enables both to take into account the sparsity and to have a fast access to each element. For small or large number of points, the traversal algorithm always benefits from spatial locality. One has to choose a correct j_{limit} value that minimizes the memory footprint (it depends on the data used).

Hash table Standard hash table libraries provide a simple way to deal with sparse data structures. No specific optimization is necessary to obtained a memory footprint of the sparse data structure linear in S. There is one bad point with hash tables storing large data sets, their inability to deliver fast accesses to stored elements because they rely on random access memory. As stated in section 3.3.3, there are some other parameters to look at: 1) if a library is used, access to a stored element can involve one costly function call, 2) in many hash tables collisions occur that hinder performance. In the sequel, the `glib` library implementation of hashtable is used.

5.5 Implementation of adaptive transform

Implementations of an adaptive wavelet transform are given in this subsection. Compared to subsection 5.2 where algorithms compute wavelet transforms on non-sparse data structure, the two-dimensional wavelet transform is here applied on a sparse representation of a 2D signal of size S. The algorithmic complexity of this adaptive transform is $O(S)$.

In order to oversimplify both algorithmics and numerics, the algorithm of Fig. 16(a) use lifting scheme with no update operator. So, compared to algorithm of Fig. 12(b), you only find in Fig. 16(a), calls to predict steps in x and y directions. But apart from that, and the explicit usage of sparse data structure, the principle of the inverse transform algorithm remains the same as previously seen. Hence, in Figures 16(b) and 16(c) there is a traversal of all details at one specific level and the application of predict operator as it was in Figure 13(a).

```
void wav_adapt_itransform( FTYPE *sold, int N) {
  const int dir = -1;
  int itc;
  /* loop on levels from coarse to fine */
  for (itc = N ; itc >= 2 ; itc >>= 1){
    wav_adapt_predict_y(sold, itc, N, dir);
    wav_adapt_predict_x(sold, itc, N, dir);
  }
}
```

(a) Inverse adaptive transform

```
void
wav_inside_pred_y(FTYPE *sdata, int ix,
                  int iy, int itf, int itc,
                  int N, int dir, int Nplus,
                  _VALTYPE *pdetail) {
  register int k, icoarse;
  double   predictVal, readval;
  /* iy is the detail index we are looking at */
  predictVal = 0;
  /* Apply predict operator in y-direction */
  for (k = 0; k < wav_np; k++) {
    icoarse = (Nplus + iy +
               (k + wav_offp)*itc - itf)%N;
    _GETF(sdata,ix,icoarse,N,readval);
    predictVal += wav_filterPred[k] * readval;
  }
  *pdetail += - dir * predictVal;
}
```

(b) Prediction operator in y-direction

```
void wav_adapt_predict_y(FTYPE *sdata, int itc,
                         int N, int dir) {
  _KEYTYPE ix, iy, idetail;
  _VALTYPE *pdetail;
  int      itf = itc / 2, Nplus = N*N;
  int      p, pstart, pend, ilevel;
  ilevel = mylog2(itf);
  if (ilevel == 0) {
    pstart = 0; pend = sdata->ends[0];
  } else {
    pstart = sdata->ends[ilevel -1];
    pend = sdata->ends[ilevel];
  }
  /*  Iterate over non-zero wavelet coefficients
      at level parameterized by itf */
  for (p = pstart; p < pend; p++) {
    /* Get 'idetail': a key from the list. */
    idetail    = sdata->keys[p];
    /* Extract indices (ix,iy) from 'idetail' */
    _INVKEYXY(idetail,ix,iy);
    /*  With 'iy', determine wether the point
        (ix,iy) is a detail at level 'itf' */
    if ((iy&itf) != 0) {
      /* Get pointer to memory area 'pdetail'
         where detail val. will be stored  */
      _GETPDETAIL(sdata,ix,iy,p,N,pdetail);
      /* Compute the prediction and store
         difference at 'pdetail' */
      wav_inside_pred_y(sdata, ix, iy, itf, itc,
                        N, dir, Nplus, pdetail);
    }
  }
}
```

(c) Walk trough all space to predict in y-dir.

Figure 16: 2D inverse adaptive wavelet transform

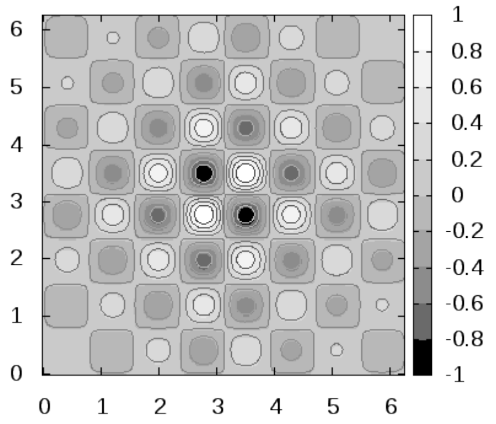
A key point to achieve this transform efficiently is to traverse the sparse data structure as quickly as possible. In order to do that, one can think of constructing a supplementary 1D array storing the non-zero wavelet locations. Even if the building of this array induces small computation and memory overheads, the array provides 1) a simple way to traverse the sparse data structure and 2) a quick sequential access to all non-zero locations.

The array of non-zero locations is named `sdata->keys` in the code. The macro `_INVKEYXY` allows one to retrieve indices `ix` and `iy` wich are encoded into one single element of this array. The array is organized and ordered such as all entries of one given wavelet level is stored contiguously. This strategy is useful to improve the efficiency of the algorithm, because traversals are level-wise. Following this idea, at the beginning of the function presented in Fig. 16(c), the two indices `pstart` and `pend` defines an interval associated to a level j (also characterized by `itc`). This interval corresponds to starting and ending indices in array `sdata->keys` where non-zero locations of level j are stored.

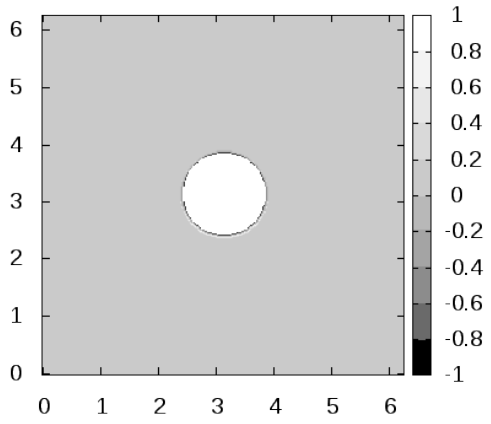
The macros `_GETF` and `_GETPDETAIL` permit respectivelly to get the value of a coefficient at one location, and to get the pointer towards a coefficient at one location. The advantage of having all these macros is that they involve no function call overhead, and they hidden the different data structure that can be used underneath. Practically, depending on one compiler options, the code associated with the macros will be based on whether a hashtable or on a P-tree for example.

5.6 Performance of sparse datatype in adaptive transform

This subsection is devoted to the performance comparison of sparse data structures. Different measures are shown concerning the execution of adaptive transform on a Nehalem-EP machine. Three implementations are considered based on several sparse data types: a P-tree version, a hashtable version, a dense data structure version. Roughly speaking, algorithms used in these versions are similar, but the macros that access sparse data are specifically designed. Source codes corresponding to this experiment are stored into `wav2d` directory.



(a) Test case A (`nb_mode=4`)



(b) Test case B

Two reference test cases have been used to conduct the experiments. In both test cases the size of the domain were 2π in each direction. First, the test case A is a tensor product of sine functions (along x and y dimensions) that multiplies a decreasing exponential (see Fig. 17(a)). The user can set a parameter `nb_modes` that fixes the number of modes in each direction. Second, the test case B define a disc in the center of the domain with the value 1 inside and 0 outside. The radius of the circle has been set to 1.5 for this experiments (see Fig. 17(b)). For test cases A and B, several mesh sizes were investigated as described in table 5. In the sequel, we will note *sparsity rate* (abbreviated *srate* and also called *fill ratio*) the ratio of non-zero elements to total elements. In the table 5, the *srate* percentage corresponds to S, the number of non-zero elements, over the total number of points in the uniform mesh N^2. It is worth noticing that data stored are quite sparse, we keep less than one percent of uniform mesh points in each of the six configurations.

Case name	N	`nb_modes`	N^2	S	srate
A1	4096	32	17.10^6	33.10^3	0.20%
A2	16384	128	268.10^6	524.10^3	0.20%
A3	32768	256	1073.10^6	2097.10^3	0.20%
B1	4096	-	17.10^6	64.10^3	0.38%
B2	16384	-	268.10^6	258.10^3	0.10%
B3	32768	-	1073.10^6	516.10^3	0.05%

Table 5: Settings and sizes of test cases used in experiments

Using adaptive inverse wavelet transform algorithm, benchmarks have been performed for these six settings (the reader is referred to the `bench.sh` script in the `wav2d` directory). Single floating-point precision is taken for wavelet data structure coefficients. Three data structures are compared on the basis of runtime and amount of memory required to store sparse data. To determine the actual memory usage taken for sparse data, a snapshot of the process memory consumption is taken just before and just after the filling of the main sparse data structure. Substracting the two values, one can estimate the amount memory required by the sparse data. And this estimation is still correct even if a library is used (which is the case when using the hashtable in the proposed implementation).

The `glib` implementation uses *open addressing* and *dynamic resizing* for the hashtable in order to enhance its efficiency. Note that the hashtable implementation is parameterized by a user-selected hash function that impacts performance. The hash function indirectly determine the average number of read accesses that are involved per hashtable lookup. Bad hash functions, lead to frequent collisions; then, each lookup can generate several read accesses to reach the slot where the target key-value pair is stored. The hash function we used in our implementation is a combination of the multiplicative and division methods described in [31] (section 6.4). The user-specified hash function is a multiplicative scheme with the golden ratio as multiplicative constant. In addition, the division method is automatically employed in the `glib` library. The number of read accesses per lookup is satisfactory and lies in average in between 2.10 and 2.61 for the six settings. Note that the ideal case (almost perfect hash function) where nearly no collision occurs would give only 2 read access per lookup: one read to compare the provided key to the stored key, one read to retrieve the stored value.

	A1 case		A2 case		A3 case	
	Time	Memory	Time	Memory	Time	Memory
Dense array	28.6 ms	64 MB	492 ms	1 GB	2432 ms	4 GB
Hashtable	6.7 ms	3.59 MB	129 ms	56.3 MB	496 ms	101.6 MB
P-tree	1.5 ms	0.65 MB	24.3 ms	10.1 MB	99 ms	40.1 MB

	B1 case		B2 case		B3 case	
	Time	Memory	Time	Memory	Time	Memory
Dense array	31.2 ms	64 MB	474 ms	1 GB	2082 ms	4 GB
Hashtable	16.3 ms	7.1 MB	72.9 ms	13.7 MB	200 ms	56 MB
P-tree	4.1 ms	4.9 MB	17.6 ms	70.7 MB	44.8 ms	280 MB

Table 6: Runtime measurements for adaptive inverse wavelet transform

For the A-series (A1, A2, A3), the behaviour of the three data structures is quite simple to analyse. The sparse storages of hashtable and P-tree save actually much space compared to a dense array that keeps all N^2 floating point values. The P-tree achieves a lower memory usage than the hashtable, but this ordering is not typical.

The speedup between runtimes of P-tree and hashtable is in the interval from 4.5 to 5.5. As already said at page 26, the hashtable has two major drawbacks: random accesses in memory narrow down performance, and function calls induced by using a library can add an extra overhead. On the other hand, the P-tree provides fast read access to any element with a cost limited to two memory reads at most, and above all, has spatial locality property. Concerning the B-series (B1, B2, B3), the P-tree data structure takes much more memory than hashtable. This situation is the most usual case. The runtimes show also a speedup from 4 to 4.5 for the P-tree data type over the hashtable data type. For largest test case A3 and B3, the random access penalty of hashtable become predominant because compressed sparse data does not fit entirely in the L3 cache of 16 MB. As a consequence, the gap between runtimes of P-tree and hashtable is as large as a speedup factor of 5 (case A3) and a speedup of 4.5 (case B3).

As far as fast reading access is needed, the presented hashtable performance is below P-tree performance. At least two ways can be considered to improve the hashtable version: the elements can be stored with specific patterns in order to improve the spatial locality [41] (this is partly done in storing elements level by level in the proposed implementation), the overhead of function calls can be lowered using homemade hashtable instead of library use. Nevertheless, concerning the first improvment, pure sequential memory accesses can not be achieved in traversing a hashtable: it would contradict the principle of easy and cheap element insertion that induces holes in the hashtable that can host new added values, also another objective is to scatter data evenly through hash table. Therefore, for large data set, hashtable can not compete with P-tree reading access time. Let us have a look now at memory storage cost, the P-tree usually consumes more than hashtable, even if complex tree-based data structure can be designed to decrease the number of zero coefficients that are stored.

5.7 Related works and discussion

There is a large body of work on wavelet adaptive and adaptive remeshing technics applied to time evolution problems. Interpolating wavelets papers [1, 13, 19, 34, 37, 46] and lifting scheme [14, 15, 30, 55, 56] are playing a central role in this context, because they provide a competitive tool to reduce both computational and memory resources. The strategy of mixing interpolating wavelets and adaptive remeshing to solve PDEs has been applied to several domain specific problems, such as: Navier-Stokes equations [47], academic equation sets [26, 45, 59], reaction-diffusion equations [18], fluid dynamics [60], parabolic PDEs [42], hyperbolic PDEs [12, 27, 28, 62], Vlasov solvers [4, 22–25, 52].

It is noticeable that there are additional computational costs associated with the use of the adaptive multi-resolution wavelet methodology. As it can be seen in the algorithm of Fig. 15, three steps are added compared to a uniform mesh framework: construct a new adaptive mesh, compute the wavelet transform, apply the threshold. Furthermore, applying the time-marching step involves a traversal of the adaptive mesh that adds overhead if caution is not taken. Finally, as we have seen, the design of the sparse data structure underlying wavelet representation has also a big impact on performance.

Data structures adapted to wavelet implementation have been extensiveley studied in the computer graphics community [48]. Wavelets are used in different subfields of this community, such as: image compression, automatic level-of-detail control for editing and rendering curves and surfaces, global illumination, volume rendering, rendering of complex scenes in real time. Depending on their usage, data structures used in wavelet processing are tuned specifically. Main trends are to tune data types for high compression rates, or for fast running times. But, generally speaking, the most space efficient methods for storing wavelet representation always leads to a rather high overhead for lookups.

Few authors have emphasize on sparse data structures that are adapted to wavelet storage and that have also good properties regarding fast random access [24, 33, 34, 41]. More work needs to be done to handle efficiently wavelet representations in multidimensionnal settings where data sets can be large and do not fit into caches. In order to achieve lower runtimes, wavelet-based applications have to simultanously reduce the number of operations to be done and get a quick access to processed data. Even if an asymptotic reduction of algorithmic cost is obtained through adaptive strategy, suitable data structure are strongly required in order to achieve effective reduced runtimes compared to non-adaptive classical approaches.

6 Conclusion

This course gives a deeper understanding of the role of datatype performance in adaptive wavelet-based applications. It presents some benchmark technics and code fragments useful to learn about sparse data structure, adaptive algorithms, and how to evaluate their performance. Material and practical examples are given, and they provide good introduction for anyone involved in the development of adaptive applications.

A key ingredient for the realization of an adaptive scheme that solve an evolution problem is the organization of the data, that is how to store wavelet coefficients. The data must be organized such at, the benefits of the adaptive method in term of algorithmic cost reduction is not wasted by overheads of data management [2]. Uniform non-adaptive method based on arrays can benefit from sequential accesses in memory. With these linear accesses, one can expect to reach the maximum available main memory bandwidth and excellent read access mean time. By contrast, sparse data representation needed for storing wavelet coefficients can not achieve such high performance. When evaluating a wavelet-based application, one has to ensure that the overhead induced by the sparse data structures is much smaller than the gain of efficiency brought by the adaptive approach. It is not straightforward to design suitable data structures for wavelet-based algorithms and to benchmark them in order to obtain a quantitative validation of the approach against a classical non-adaptive scheme. The difficulty comes partly from the intertwining between the numerical scheme, algorithms and datatypes used. It is not unusual that you obtain a large reduction over the number of floting point operations thanks to an efficient adaptive numerical scheme, but you loose at the same time all this benefit because of slow data accesses or sparse data management.

Access time to main memory is a bottleneck for many of today's software. As a programmer, we have seen the important need to understand the memory hierarchy because it has a big impact on the performance of your applications. We have looked at the task of redesigning and reorganizing data structures in order to improve cache locality (spatial and temporal). Pointers and indirections should be employed carefully because they slow down mean access time and add a memory cost. As far as performance is concerned, the implementation of a sparse datatype requires to design an optimized data layout and algorithms with certain access patterns to fasten runtimes. The speedup between optimized implementation and simple one can be as large as a factor 15.

Usage and optimization of typical data structures for sparse storage have been discussed. Comparisons of tree-based data structures and hashtables have been shown. Hashtables are quite simple to use and lead to memory consumption linear with the number of elements stored. Nevertheless, tree-based data structures are versatile and optimized version can lead to impressive resource savings. On the first hand, the EZW coding based on a tree structure allows for a very compact memory footprint of one wavelet representation. On the other hand, we have exhibit P-tree data type that ensures fast read access time to coefficients of a wavelet tree. Both hashtables and tree-based data structures have to be considered within wavelet applications with repect to resource consumption or software engineering objectives.

Wavelet-based scheme is useful in solving partial differential equations (PDE) that involve different spatial and temporal scales. Wavelet-collocation methods is one of the methods available for solving the problem in the physical space, using a dynamically adaptive grid. Wavelets have the ability to dynamically adapt the computational grid to local structures of the solution. Some algorithms and data types usable in such framework are presented in this document. Performance comparisons between hashtable and P-tree datatypes in a 2 dimensional setting are illustrated. The use of macros helps evaluate and compare the different sparse datatypes.

The efficiency of wavelets is well established in many fields. In the scope of computer science, wavelet based multiresolution representation became the state of the art technique in image processing and has also been recognized in computer graphics [53]. However, more research and development is required before wavelets become core technology in numerical simulations and in solvers for partial differential equations. A main issue among others remains to improve memory footprint and access time performance for wavelet representations [8].

References

[1] P. S. Addison. *The Illustrated Wavelet Transform Handbook: Introductory Theory and Applications in Science, Engineering, Medicine and Finance.* IOP Publishing Ltd., 2001.

[2] A. Barinka, T. Barsch, P. Charton, A. Cohen, S. Dahlke, W. Dahmen, and K. Urban. Adaptive wavelet schemes for elliptic problems - implementation and numerical experiments. *SIAM J. Scient. Comput*, 23:910–939, 1999.

[3] M. A. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-Trees. *SIAM Journal on Computing*, 35(2):341–358, 2005. `http://www.cs.sunysb.edu/~bender/pub/sicomp05-BenderDeFa.ps`.

[4] N. Besse, G. Latu, A. Ghizzo, E. Sonnendrucker, and P. Bertrand. A wavelet-MRA-based adaptive semi-lagrangian method for the relativistic Vlasov-Maxwell system. *Journal of Computational Physics*, 227(16):7889 – 7916, 2008. `http://www.sciencedirect.com/science/article/B6WHY-4SGD4YJ-1/2/41fd1c69bacca48b1f3ba82c8ae6850a`.

[5] J. Bruce, W. Ng Spencer, and D. T. Wang. *Memory systems: cache, DRAM, disk.* Elsevier, 2008.

[6] R.E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003.

[7] D. Chaver, C. Tenllado, L. Piñuel, M. Prieto, and F. Tirado. Vectorization of the 2D wavelet lifting transform using SIMD extensions. *Proc. 17th IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2003.

[8] A. Cohen. Adaptive methods for pde's - wavelets or mesh refinement ? In *Proceedings of the International Conference of Mathematics*, 2002. `www.ann.jussieu.fr/~cohen/pekin.ps.gz`.

[9] A. Cohen. Adaptive multiscale methods for evolution equations. In *proceedings of ENUMATH conference, (Ischia Porto 2001)*, 2002. `http://www.ann.jussieu.fr/~cohen/procischia.ps.gz`.

[10] A. Cohen, I. Daubechies, O. Guleryuz, and M. Orchard. On the importance of combining wavelet-based non-linear approximation with coding strategies. *IEEE Trans. Inform. Theory*, 48(7):1895–1921, 2002. `http://www.ann.jussieu.fr/~cohen/cdgo.ps.gz`.

[11] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms (3rd ed.)*. MIT Press, 2009.

[12] P. Cruz, M. A. Alves, F.D. Magalhães, and A. Mendes. Solution of hyperbolic pdes using a stable adaptive multiresolution method. *Chemical Engineering Science*, 58(9):1777 – 1792, 2003.

[13] P. Cruz, A. Mendes, and F.D. Magalhães. Using wavelets for solving PDEs: and adaptive collocation method. *Chemical Engineering Science*, 56:3305–3309, 2001.

[14] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl.*, 4(3):247–269, 1998.

[15] Ingrid Daubechies and Wim Sweldens. Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl*, 4:247–269, 1998.

[16] E. Demaine and C.E. Leiserson. MIT - Introduction to Algorithms, 2007. `http://www.catonmat.net/blog/summary-of-mit-introduction-to-algorithms`.

[17] J. Dolbow, Khaleel M. A., and J. Mitchell. Multiscale mathematics initiative: A roadmap. Technical report, Tech. Rep. PNNL-14966 Pacific Northwest National Laboratory, 2004. `http://www.sc.doe.gov/ascr/Research/AM/MultiscaleMathWorkshop3.pdf`.

[18] Margarete O. Domingues, Sônia M. Gomes, Olivier Roussel, and Kai Schneider. An adaptive multiresolution scheme with local time stepping for evolutionary pdes. *J. Comput. Phys.*, 227(8):3758–3780, 2008.

[19] D. L. Donoho. Interpolating wavelet transforms. Technical report, Preprint, Department of Statistics, Stanford University, 1992. `http://www-stat.stanford.edu/~donoho/Reports/1992/interpol.pdf`.

[20] J. Erickson. Topics in analysis of algorithms. `http://compgeom.cs.uiuc.edu/~jeffe/teaching/473/`.

[21] GNOME. Glib reference manual. `http://library.gnome.org/devel/glib/2.24`.

[22] M. Gutnic, M. Mehrenberger, E. Sonnendrücker, O. Hoenen, G. Latu, and E. Violard. Adaptive 2d vlasov simulation of particle beams. In *Proceedings of ICAP 2006*, 2006. `http://epaper.kek.jp/ICAP06/PAPERS/THMPMP02.PDF`.

[23] Michael Gutnic, Matthieu Haefele, Ioana Paun, and Eric Sonnendrücker. Vlasov simulations on an adaptive phase-space grid. *Comput. Phys. Commun.*, 164:214–219, 2004.

[24] M. Haefele, G. Latu, and M. Gutnic. A parallel vlasov solver using a wavelet based adaptive mesh refinement. In *ICPP 2005 Workshops*, pages 181–188. IEEE Computer Society, 2005. `http://icps.u-strasbg.fr/people/latu/public_html/docs/icpp-hpsec-05.pdf` `http://dx.doi.org/10.1109/ICPPW.2005.13`.

[25] Matthieu Haefele. *Simulation adaptative et visualisation haute performance de plasmas et de faisceaux de particules*. PhD thesis, Laboratoire des Sciences de l'Images, de l'Informatique et de la Télédétection, LSIIT, 2007.

[26] Ami Harten. Adaptive multiresolution schemes for shock computations. *Journal of Computational Physics*, 115(2):319 – 338, 1994.

[27] Mats Holmstrom. Solving hyperbolic PDEs using interpolating wavelets. *SIAM Journal on Scientific Computing*, 21(2):405–420, 1999.

[28] Mats Holmström and Johan Waldén. Adaptive wavelet methods for hyperbolic PDEs. *J. Sci. Comput.*, 13(1):19–49, 1998.

[29] S. Jain, P. Tsiotras, and H.-M. Zhou. A hierarchical multiresolution adaptive mesh refinement for the solution of evolution PDEs. *SIAM Journal on Scientific Computing*, 31(2):1221–1248, 2008.

[30] Roger L. Claypoole Jr., Geoffrey M. Davis, Wim Sweldens, and Richard G. Baraniuk. Nonlinear wavelet transforms for image coding via lifting. *IEEE Transactions on Image Processing*, 12(12):1449–1459, 2003.

[31] D. Knuth. *The Art of Computer Programming (3rd ed.)*, volume 3. Addison-Wesley, 1998.

[32] Piyush Kumar. Cache oblivious algorithms. In *Algorithms for Memory Hierarchies. Editors: Meyer, U. and Sanders, P. and Sibeyn, J.*, volume LNCS 2625, pages 193–212. Elsevier, 2003. `http://www.compgeom.com/co-chap/co.pdf`.

[33] Sylvain Lefebvre and Hugues Hoppe. Compressed random-access trees for spatially coherent data. In *Rendering Techniques (Proceedings of the Eurographics Symposium on Rendering)*. Eurographics, 2007.

[34] R. Lippert, T. Arias, and A. Edelman. Multiscale computation with interpolating wavelets. *Journal of Computational Physics*, 140(2):278–310, 1998.

[35] Linux magazine 88. Dissection de glib. `http://www.unixgarden.com/index.php/programmation/dissection-de-glib-les-tables-de-hachage`.

[36] S. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:674–693, 1989. `http://www.dii.unisi.it/~menegaz/docs&papers/mallat-89.pdf`.

[37] Stéphane Mallat. *A Wavelet Tour of Signal Processing, Third Edition: The Sparse Way.* Academic Press, 2008.

[38] Abhijit Menon-Sen. How hashes really work, 2002. `www.perl.com/lpt/a/679`.

[39] R.J.E. Merry. Wavelet theory and applications, a literature study. Technical report, Master's thesis, Eindhoven University of Technology, 2005. `alexandria.tue.nl/repository/books/612762.pdf`.

[40] D. Mount. Data structures - course. `http://www.cs.umd.edu/~mount/420/`.

[41] F. F. Rodler and Pagh R. Fast random access to wavelet compressed volumetric data using hashing, 2001. `http://www.brics.dk/RS/01/34/`.

[42] Olivier Roussel, Kai Schneider, Alexei Tsigulin, and Henning Bockhorn. A conservative fully adaptive multiresolution algorithm for parabolic pdes. *Journal of Computational Physics*, 188(2):493 – 523, 2003.

[43] Y. Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003. `http://www-users.cs.umn.edu/~saad/books.html`.

[44] A. Said and W. A. Pearlman. A new, fast, and efficient image codec based on Set Partitioning In Hierarchical Trees. *IEE Trans. on Circuits and Systems for Video Technology*, 6(3):243–250, 1996.

[45] J. C. Santos, P. Cruz, F.D. Magalhães, and A. Mendes. 2-D wavelet-based adaptive-grid method for the resolution of PDEs. *AIChE Journal*, 49:706–717, 2003.

[46] K. Schneider, M. Farge, F. Koster, and M. Griebel. Adaptive wavelet methods for the Navier-Stokes equations. *Numerical Flow Simulation II*, 75:303–318, 2001.

[47] K. Schneider, M. Farge, F. Koster, and M. Griebel. Adaptive wavelet methods for the navier-stokes equations. In *Notes on Numerical Fluid Mechanics (Ed. E. H. Hirschel) - Springer*, pages 303–318, 2001.

[48] P. Schroder. Wavelets in computer graphics. In *Proceedings of the IEEE*, volume 84(4), pages 615–625. IEEE, 1996.

[49] John Shalf, Sudip Dosanjh, and John Morrison. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, chapter 1, pages 1–25. Springer Berlin / Heidelberg, 2011.

[50] J. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEE Trans. on Signal Processing*, 41(12):3445–3461, 1993. `http://www.cs.tut.fi/~tabus/course/SC/Shapiro.pdf`.

[51] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.

[52] E. Sonnendrücker, M. Gutnic, M. Haefele, and G. Latu. Vlasov simulations of beams and halo. In *PAC 2005 proceedings*, 2005.

[53] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[54] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, second edition, 1997.

[55] W. Sweldens and P. Schroder. Building your own wavelets at home. *ACM SIGGRAPH Course notes*, 1996. `http://cm.bell-labs.com/who/wim/papers/athome/athome.pdf`.

[56] Wim Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Applied and Computational Harmonic Analysis*, 3(2):186 – 200, 1996.

[57] Chilimbi Trishul M., Hill Mark D., and Larus James R. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–74, 2000.

[58] C. Valens. EZW encoding. `http://pagesperso-orange.fr/polyvalens/clemens/ezw/ezw.html`.

[59] Oleg V. Vasilyev and Samuel Paolucci. A fast adaptive wavelet collocation algorithm for multidimensional PDEs. *Journal of Computational Physics*, 138(1):16 – 56, 1997. `http://www.sciencedirect.com/science/article/B6WHY-45KV05C-2/2/ba67a076220d0acba3173d425d3c4acf`.

[60] O.V. Vasilyev. Solving multi-dimensional evolution problems with localized structures using second generation wavelets. *Int. J. Comp. Fluid Dyn., Special issue on High-resolution methods in Computational Fluid Dynamics*, 17(2):151168, 2003.

[61] O.V. Vasilyev, D.A. Yuen, and S. Paolucci. The solution of PDEs using wavelets. *Computers in Phys.*, 11(5):429–435, 1997.

[62] Johan Walden. Filter bank methods for hyperbolic PDEs. *SIAM Journal on Numerical Analysis*, 36(4):1183–1233, 1999.

[63] Wikipedia, bit-plane. `http://en.wikipedia.org/wiki/Bit_plane`.

[64] Wikipedia, bucket sort. `http://en.wikipedia.org/wiki/Bucket_sort`.

[65] Wikipedia, b-tree. http://en.wikipedia.org/wiki/B-tree.

[66] Wikipedia, dynamic arrays. http://en.wikipedia.org/wiki/Dynamic_array.

[67] Wikipedia, hash table. http://en.wikipedia.org/wiki/Hash_table.