Data structure design and algorithms for wavelet-based applications

Guillaume Latu
CEA/IRFM

Collaborators of INRIA CALVI & University of Strasbourg:
Nicolas Besse, Matthieu Haefele, Michaël Gutnic, Eric Sonnendrücker

14/06/2010

Ecole CNRS - Fréjus - juin 2010
Méthodes multirésolution et
méthodes de raffinement adaptatif de maillage
Outline

1. Introduction
2. Algorithmic and performance issues
3. Data structures for multiresolution
4. 1D wavelet algorithms
5. Wavelets for evolution equations
6. Application 1: Vlasov 2D
7. Application 2: Vlasov 4D
Outline

1. Introduction
2. Algorithmic and performance issues
3. Data structures for multiresolution
4. 1D wavelet algorithms
5. Wavelets for evolution equations
6. Application 1: Vlasov 2D
7. Application 2: Vlasov 4D
Multiresolution approaches

- Multiresolution research: disjointed among several disciplines
- A lot of large scale scientific problems require adaptivity
- A need to develop in a collaborative framework:
  - mathematical techniques
  - computational methods
  - softwares

- Source code, slides and course notes available at:

  http://icps.u-strasbg.fr/people/latu/public_html/wavelet/
Application that uses Wavelet

- **Wavelets**: especially useful tool for managing multiresolution

- **Aim of the talk:**
  - design data structures and algorithms in a wavelet-based application

- **Question:**
  - how to reduce the computational cost of application using mesh adaptation
  - how to implement and to use efficiently the discrete wavelet transform in applications
Some wavelet applications

Image compression

- This course focuses on some applications of wavelet
  - common factor: discrete wavelet transform
- A main goal in image field: compression
- Wavelet encoding schemes are used (JPEG2000)
- High compression rates & high SNR
  - Signal-to-Noise Ratio
- Trade-off
  - image quality
  - compression rates
  - computational complexity
Some wavelet applications

Video encoding

- Video encoding with wavelets
  - Image parts with low energy → should use few bits
  - Try to avoid adding noise
  - Coupled to *lossless* encoding

- Time dimension
  - wavelet analysis: 3D
    - or
  - 2D wavelet transform + motion estimation
Some wavelet applications
Application to PDEs

- PDEs
  - help describing physical phenomena, help modeling
  - hard to solve analytically → numerical schemes

- Pb: large computation times
  - work on a reduced model
  - take larger machine: parallelization
  - reduce costs: change numerical schemes & algorithms

- Wavelets, adaptive remeshing
  - track steep front, sharp features
    → increase local resolution
  - smooth areas
    → decimate/remove grid points
  - accuracy control, numerical schemes, ... (mathematics)
  - efficient sparse data structure, algorithms, parallelization (computer science)
Objectives

Efficient sparse representation (1)

- Design sparse data structures
  - many coefficients equal to zero
  - store and work only on non-zero ($nz$) entries

- Issues on sparsity management
  - memory overhead (store $nz$ locations + $nz$ values)
  - dynamic data size
  - important parameter: number of non-zero ($nnz$)
  - complex memory access pattern

- Wavelet-based application
  - reduce the number of operations
    - algorithms choice, sparsity management
  - compact representation in memory
  - multiresolution scheme: have access to one peculiar level
Pitfalls
Efficient sparse representation (2)

- Architectural trends:
  - cost of accessing main memory

- Wide gap
  - available perf. and achieved perf. of software
  - estimation of the gap

- What should we do?
  - reorganize data struct. to improve cache locality
  - clustering, compression of data in memory
  - cache-conscious data structures

- Extensively compare against dense applications
  - optimization and high-performance computations
Pitfalls
Designing correct algorithms

■ Programmer’s problem
  1. design/choose the right algorithm
  2. prove correctness of the algorithm
  3. easy coding and debugging of the algorithm
  4. efficiency of the algorithm

■ Troubles with wavelet-based application
  ■ sharp mathematics
  ■ sparse and complex data in memory
  ■ tricky algorithms

■ Invariants (e.g. mass/average conservation)
  ■ look for invariants that should remain true
  ■ discuss with collaborators to identify them
  ■ help for debugging/proof of correctness
Compilers are good tools **but**

- understand machine-level code & architecture helps!

**Questions:**
- is an *if-else* statement costly or not?
- how much overhead for a function call?
  - for a system call?
1. Introduction

2. Algorithmic and performance issues

3. Data structures for multiresolution

4. 1D wavelet algorithms

5. Wavelets for evolution equations

6. Application 1: Vlasov 2D

7. Application 2: Vlasov 4D
Consider a *caricature* architecture model
- a processor operating at 1 GHz (1 ns clock cycle)
- connected to a DRAM with a latency of 100 ns, no cache

Assume that the processor can execute 1 FLOP per cycle
- FLOP = floating point operation
- peak performance = 1 GFLOPS

Consider adding two arrays on this architecture
- each FLOP requires two data accesses
- *peak* speed of this computation:
  - 1 FLOP (addition) every 200 ns $\rightarrow$ 5 MFLOPS
- achieves small fraction of the peak processor performance

This major problem is known as
*memory wall* or *processor-memory bottleneck*
Computer architecture
Memory hierarchy

When the processor needs to read/write main memory
→ checks whether a copy of that data lies in the cache
→ if not, copy a block of data into the cache
→ replace old data by the new cache lines/block

Lot of time spent to move data from memory to cache

Copying is overhead that slows down the real work

Optimization target:

- reduce the number of transfers between memory and cache
  → Cache-aware algorithm
- be aware of the constraint of main memory bandwidth
- dense calculations often easier to optimize than sparse ones
Spatial locality refers to the use of data elements within relatively close storage

- Classic example: array processing
- Elements $i$ and $i + 1$ are adjacent
- Process element $i$, then process element $i + 1$

Hardware/Compiler mechanisms

- Cache lines
- Look-ahead, prefetch streams
- Burst transfer
Temporal locality means that referenced memory address is likely to be referenced again soon.
- frequently used subroutines/data
- local variables
- example: consider the loop used for a dotproduct, the scalar alpha gets used repeatedly

```
sum = 0.0;
for (k=1; k != n; k++)
    sum = sum + x(k)*y(k);
```

Hardware/Compiler mechanisms
- L1,L2,L3 caches
  → guaranty that data frequently accessed are in a cache
- data alignment can impact cache performance
Reminder

- a C variable has a unique \textit{memory address}
- pointer: a variable that stores a \textit{memory address}
- accessing a variable’s value by a pointer is called \textit{indirection}

\textit{Dynamic} data structure

- size and shape change during run-time
- examples: linked-list, hash table, tree
- pointers useful
  - maintain logical connection between nodes
- may get scattered all over memory 😞
Application behavior (caricature)

- [Fast] a program that often access data in cache memory
- [Medium] a program that accesses sequentially in memory
- [Slow] same program that accesses data randomly in memory

Real-life example: linked-list

- your application contains linked-lists
  - example: to store one level of wavelet coefficients
  - a walk through the list → big jumps in memory
  - performance depends on how the links are set between nodes

Code stored in linkedlist directory
Three possible improvements:

1. Replace linked-list with
   1) array or 2) dynamic array [or 3) chunk list ?]
   - traversal time enhanced
   - reduced time for random access
   - insert/delete time increased

2. Copy/reorder of the linked-list
   - periodically copy the list into a new one
   - reordering the records in memory space
   - a walk through the new list → contiguous accesses

3. Software prefetching
   - requests a data from main memory beforehand
   - example: __builtin_prefetch statement (gcc/icc)
   - expect hiding memory latency
   - sometimes automatically activated by compilers
- **Testbed:** 8-core Intel Nehalem node
  - 2.93 Mhz CPU frequency, 32 KB L1 data cache per core,
  - 256 KB L2 cache per core, 16 MB L3 cache per node

- **Benchmark:** a list containing 8M records (160 MB)

<table>
<thead>
<tr>
<th></th>
<th>time (second)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial traversal (linked list)</td>
<td>0.84 s</td>
<td>1</td>
</tr>
<tr>
<td>Solution 1 (array)</td>
<td>0.056 s</td>
<td>15</td>
</tr>
<tr>
<td>Solution 2 (list after copy/reorder)</td>
<td>0.061 s</td>
<td>13.8</td>
</tr>
<tr>
<td>Solution 3 (list + prefetch)</td>
<td>0.82 s</td>
<td>1.02</td>
</tr>
</tbody>
</table>

**Table:** Time measurements for performing a sum over nodes stored in a linked list or with alternative data structures

- **Benchmark stored in** `linkedlist` directory
Exploiting spatial and temporal locality is critical for
- amortizing memory latency
- increasing effective memory bandwidth

How to improve spatial and temporal locality?
- data layout, data access pattern, computations organization

Optimized data-structure requires
- deep understanding of an application’s code
- knowledge of the underlying cache architecture
- significant rewriting of code
- perhaps get good advices or help
Computational complexity
Asymptotic analysis (1)

- **Computational complexity**
  - predict run-time depending on problem size (approximately)

- **Framework**
  - main parameter of a program: input size $n$
  - running time: $T(n)$
  - computational cost: $C(n)$
  - when $n$ large enough: $T(n) \approx \alpha C(n)$

- **Numerical analysis, Big O notation**
  - how the computational cost behaves asymptotically?
  - count the number of operations or amount of memory
  - input $n$ grows to $\infty$
Asymptotic notations

- $O(g(n))$: A function $f$ is $O(g(n))$ iff there exist positive constants $c$, and $n_0$ such that
  \[
  \forall n \geq n_0, \ 0 \leq f(n) \leq c \cdot g(n)
  \]

- $\Theta(g(n))$: A function $f$ is $\Theta(g(n))$ iff there exist positive constants $c_1, c_2$, and $n_0$ such that
  \[
  \forall n \geq n_0, \ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)
  \]

- Example: $n^2 + n = O(n^2)$ as $n \to \infty$
Sorting a list of $n$ items,

- naive algorithm takes $O(n^2)$ work
- optimal algorithm (e.g. quicksort) takes $O(n \log(n))$
- item from a finite alphabet $\rightarrow O(n)$ work (bucket sort)

Exercise: Implement a bucket sort on an integer array of $N$ elements. Benchmark the code, check asymptotic behaviour in $O(n)$. 
Computational complexity
Renewal of the subject

- Pb: gap between cache and memory bandwidths
- Introducing a new measure
  - count memory transfers $MT(n)$ for problem of size $n$
  - i.e. count main memory accesses, not cache accesses!
  - expecting $MT(n)$ to be small

- New area of algorithm research
  - cache-friendly data access pattern and algorithms
  - cache-aware algorithms (depends on cache parameters)
    - e.g. B-tree
  - cache-oblivious algorithms (optimal use of cache)
Outline

1. Introduction
2. Algorithmic and performance issues
3. Data structures for multiresolution
4. 1D wavelet algorithms
5. Wavelets for evolution equations
6. Application 1: Vlasov 2D
7. Application 2: Vlasov 4D
Basic Linear algebra tools widely used, many libraries
- Dense: BLAS, LINPACK, ScaLAPACK
- Sparse: HIPS, HYPRE, pARMS, Pastix, MUMPS, SUPERLU

Standard layouts for dense matrix storage
- matrix $A$ is a pointer to an array of pointers 😞
- row-major order: stored row-wise in memory
  → C language
- column-major order: stored column-wise in memory
  → Fortran
Multidimensional array
Memory Layout of 2D arrays and matrices (2)

(a) Initial Matrix

(b) Storage using indirections

(c) Storage with row major order

(d) Storage with column major order

- Accessing matrix elements in the wrong order can lead to poor spatial locality
Multidimensional array
Matrix-multiply example (1)

```c
22 void mat_mul_basic(double* A, double*tB, double*tC, int N) {
23     register double sum;
24     register int i, j, k; /* iterators */
25     for (j=0; j<N; j++)
26         for (i=0; i<N; i++) {
27             for (sum=0., k=0; k<N; k++) {
28                 sum += A[k+i*N]*tB[k+j*N];
29             }
30             tC[i+j*N] = sum;
31         }
32 }
```

**Figure:** Basic algorithm

Code stored in `matmul` directory
Multidimensional array
Matrix-multiply example (2)

```c
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
}
```

**Figure:** BLAS library call
Multidimensional array
Matrix-multiply example (3)

```c
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    tC[i+j*N] = 0.;

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

**Figure:** Blocked algorithm
Multidimensional array
Matrix-multiply example (4)

Worst case ($m$ the common size of blocking factors)

- $MT_{basic}(N) = 2N^3 + N^2$
- $MT_{blocked}(N) = 2N^3/m + N^2$
Multidimensional array
Matrix-multiply example (5)

- \( C(N = 2048) = 2048^3 \) additions + \( 2048^3 \) multiplications
- Theoretical peak perf.: 12 GFLOPS (64-bit computations)

<table>
<thead>
<tr>
<th></th>
<th>Basic algo.</th>
<th>Blocked algo.</th>
<th>BLAS call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>19.3 s</td>
<td>5.66 s</td>
<td>1.68 s</td>
</tr>
<tr>
<td>GFLOPS</td>
<td>0.89</td>
<td>3.04</td>
<td>10.2</td>
</tr>
</tbody>
</table>

**Table**: Performance of a matrix-multiply on two square matrices of size \( 2048^2 \) on a Nehalem node.

- Cache optimizations useful
- Other optimizations needed to reach BLAS perf
Dynamic array
Principle

- Dynamic array reconciles two antinomic points
  - Performance of sequential access in static arrays
  - Avoid memory waste with data resize

- How does it work?
  - allocate a fixed-size array, split it into 2 parts
    - part 1: store array elements (size: actual size)
    - part 2: reserved but unused (capacity size - actual size)
  - whenever an element is added
    - if (actual size = capacity size)
      allocate a new array in doubling the capacity
    - append the element and increment actual size
Dynamic array
Benchmark

- 8 threads working simultaneously (bandwidth saturation)
- averaging on several successive runs
- caches are not flushed between successive runs

<table>
<thead>
<tr>
<th>Array length</th>
<th>Cumulative bandwidth static array</th>
<th>Cumulative bandwidth dynamic array</th>
<th>Known peak bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>32K</td>
<td>370 GB/s</td>
<td>370 GB/s</td>
<td>-</td>
</tr>
<tr>
<td>64K</td>
<td>220 GB/s</td>
<td>220 GB/s</td>
<td>-</td>
</tr>
<tr>
<td>512K</td>
<td>150 GB/s</td>
<td>150 GB/s</td>
<td>-</td>
</tr>
<tr>
<td>8MB</td>
<td>31 GB/s</td>
<td>31 GB/s</td>
<td>32 GB/s</td>
</tr>
</tbody>
</table>

Table: Cumulative bandwidth obtained during the computation of a sum of array elements (Nehalem 8-core node)
Hash table

Principle (1)

- Hash table → functionality of a set, sparse array
  - map
    - keys (identifiers), e.g. person’s name, to associated values, e.g. telephone number
  - operation on elements: insertion, deletion, retrieval with average cost of $O(1)$!
  - for large set:
    - save space, perform well, simple to use

- Central mechanism: Hash function
  - function that return a slot index depending on a key
  - ideally map each key to a unique slot/bucket index
  - different keys could give same slot index: collision
How to deal with collisions?

one way: instead of values, have linked-list of values

Figure: A chained hash table with five pairs (key-value) and one collision located at bucket (id 3).
void testhash() {
    GHashTable htab;
    HASH_KEYTYPE key;
    HASH_KEYTYPE *pkey;
    HASH_VALTYPE *pval;
    HASH_VALTYPE sum, maxi;
    GHashTableIter iter;
    
    /** Allocate and fill hash table **/
    /** ... **/
    /** Traversal of the hash table **/
    sum = 0.;
    g_hash_table_iter_init (&iter, htab);
    while (g_hash_table_iter_next (&iter, (gpointer) &pkey,
        (gpointer) &pval))
        sum += *pval;

    /** Perform lookups in the hash table **/
    maxi = 0.;
    for (key = 0; key < MAXSIZE; key += STRIDE) {
        pval = (HASH_VALTYPE*)
            g_hash_table_lookup(htab, (void*)&key);
        maxi = (fabs(*pval) > maxi ? fabs(*pval) : maxi);
    }
}
Collisions & chaining: big overhead

Solutions:

- good hash function: avoid collisions, uniform scattering
  - double hashing
  - perfect hashing
  - universal hashing

- dynamic hash: increase table size when collisions occurred
  - reduce collisions
  - overhead: copy of the table
Hash table

Drawbacks

- Complex to setup
  - achieving $O(1)$ for insertion/retrieval: not simple

- Not so cheap
  - the cost $O(1)$ of a good hash function could be high
  - no quick way to locate an entry near another one
  - poor spatial locality in key space
  - memory access patterns that jump around

- Library use
  - overhead of a function call
  - need home-made implementation? macro, inline function

- Consider alternatives: dynamic arrays, search trees
Hash table

Traversal of a sparse array (1)

- Benchmark of a sparse array
  - many entries are undefined, only some entries are set
  - code stored in hash directory

- Benchmark settings
  - sparse array contains $S$ values
  - range of values in the sparse array $[0, N-1]$
  - inverse fill ratio equal to $\alpha = \lfloor N/S \rfloor = 15$

- Two implementations for sparse array traversal
  - one based on hash table (glib library)
    - space complexity & algo. complexity $O(S)$
  - one based on simple static array
    - entry of the array: (real value) or (NOT A VALUE constant)
    - space complexity & algo. complexity $O(N)$
Run-time for three traversals $N = 8 \times 10^7$

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 and select only defined values (complexity in time $O(N)$).
3. Traverse the whole dense array and make a lookup to the hash table at each position (complexity in time $O(N)$).

Timings on Nehalem 8-core node

- run-time $2 \approx$ run-time $1$
  iterator of the hash-table $\approx (\alpha = 15) \times$ array access time
- run-time $3 \approx 20 \times$ run-time $2$
  lookup to the hash-table $\approx 20 \times$ array access time
Tree
Definitions

- **Constitution**
  - set of *nodes* connected with edges → graph with no cycle
  - *internal node*: has a set of children nodes
  - *leaf node*: no child, one parent node
  - *root*: common ancestor node

- **Notation**
  - *siblings*: children of one node
  - *degree* of a node: number of children
  - leaves have *height* 0
  - root has *depth* 0

- **Root** $r$ and its sub-trees $T_1, T_2, \ldots, T_k$
A binary tree consists of a root node and two disjoint binary sub-trees, called the left and right sub-trees.
full binary tree can be \textit{implicitly} stored as an array (\textit{Ahnentafel} list)

- a node is stored at index $k$, left child at $2k + 1$, right child at $2k + 2$

- \textit{eliminate} memory \textit{overhead} due to pointers

\begin{verbatim}
|---|---+---|---+---+---+---|---+---+---+---+---+---+---+---|
| A | B | E | C | F | D | G | |
|---|---+---|---+---+---+---|---+---+---+---+---+---+---+---|
\end{verbatim}

\textbf{Figure:} Implicit storage of the binary tree
■ **Issues**

  ■ no bound on the number of children  
    → dynamic list needed  
  ■ internal node must have access to its children  
    → pointers, indirection required

■ **Classic representations of general trees**

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

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

■ **Code stored in gtree directory**
Questions: computation & memory cost?

Answers:
- try to avoid random accesses
- try to reduce numerous indirection levels
- try to remove some pointers

More efficient general tree data structure

```c
struct darray_gtree3 {
    struct gtree3 *nodesDArray;
    int size;
    int capacity;
};
```

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

Benefits: Fortran, contiguous, logical links, copy cost ...
Testing general tree data structures

- **Testbed**: 8-core Intel Nehalem node
  - 2.93 Mhz CPU frequency, 32 KB L1 data cache per core,
  - 256 KB L2 cache per core, 16 MB L3 cache per node

- **Benchmark**: perform traversals of general trees
  - **parameter**: number of nodes (1K→64K)
  - breadth-first algorithm (useful in wavelet-based app.)

<table>
<thead>
<tr>
<th>Nb nodes</th>
<th>Cumulative bandwidth</th>
<th>Cumulative bandwidth</th>
<th>Cumulative bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>breadth-first algo.</td>
<td>breadth-first algo.</td>
<td>array traversal</td>
</tr>
<tr>
<td></td>
<td>gtree1</td>
<td>gtree3</td>
<td>gtree3</td>
</tr>
<tr>
<td>1K</td>
<td>4.4 GB/s</td>
<td>4.8 GB/s</td>
<td>90 GB/s</td>
</tr>
<tr>
<td>8K</td>
<td>2.9 GB/s</td>
<td>3.4 GB/s</td>
<td>60 GB/s</td>
</tr>
<tr>
<td>32K</td>
<td>1.2 GB/s</td>
<td>2.7 GB/s</td>
<td>36 GB/s</td>
</tr>
<tr>
<td>64K</td>
<td>0.6 GB/s</td>
<td>1.6 GB/s</td>
<td>12 GB/s</td>
</tr>
</tbody>
</table>

**Exercise**: Why array traversal (column 4) has not the same bandwidth as the one previously observed for arrays (from 31 GB/s to 370 GB/s)?
CPU mechanism
pipelining

- CPUs breaks up instructions into smaller steps
  - example of a substep decomposition
    1. Fetch Instruction
    2. Decode Instruction
    3. Execute Instruction
    4. Write Back (store result)

- working on $N$ instructions at each cycle (overlap substeps)

<table>
<thead>
<tr>
<th>Inst. Id.</th>
<th>Pipeline Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------</td>
<td>----------------</td>
</tr>
<tr>
<td>1</td>
<td>FI</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

- Nehalem $\rightarrow$ 16-stage pipeline
- if-statement, conditional branches implies pipeline flush 😞
In rearranging the data in memory space, one could:

- increase the spatial locality and temporal locality
- build cache-aware, cache-oblivious data layout
- reduce memory-bandwidth requirement
- reduce memory latencies
- reach extreme compression
- fast traversal

Could not achieve all objectives simultaneously:

- memory access, fast traversal: B-tree, B+tree
  principle: big node contains several original nodes
  → reduced number of indirection levels
- compression: implicit binary tree
- pointer elimination: significance map
Wavelet tree in image processing
Introduction to EZW scheme

- **Embedded Zerotree wavelet (EZW) [by J.M. Shapiro]**
  - use 2D discrete wavelet transform on images
  - fast compression technique
  - high compression ratio
  - *computationally simple* technique
  - *bit stream* ordered by bit importance
    - large coeff. first $\rightarrow$ progressive transmission achievable
  - multi-resolution: encoder/decoder could stop at any point

- **Sketch of the EZW encoder**
  1. Wavelet transform of the raw image
  2. *Quantization* step (EZW algorithm) $\rightarrow$ bit stream
  3. *Entropy coding* (*lossless* compression of the bit stream)
Bit-plane principle

- Consider the encoding of $m$ entries of $n$ bits
- First, encode most significant bit (0) of each the $m$ entries
- Send bit 1 of each of the $m$ entries
- ... 
- Last, encode least significant bit $n-1$ of the entries

Example on a grayscale (8-bit) image (© CAES CNRS)
Wavelet tree in image processing
EZW - Algorithm (1)

- Basis of EZW = wavelet transform + bit-plane encoder
  1. wavelet coefficients are bit-plane encoded
  2. a few bit-planes suffice to get qualitative images
     → high compression ratio

- Most significant bit-planes are very sparse (few large coeff.)
  → strategy: preserve sparsity in order to compress
    - only bits corresponding to large coeff. are encoded
      → but, how to tell the locations of these bits?
    - encode bits location in a clever way
      → use a tree description to describe implicitly these locations
Global algorithm

**Input**: Wavelet representation \( W \)

**Output**: Bit stream \( B \)

\[
\text{SubList} \leftarrow \emptyset; \quad \text{/* Subordinate List */}
\]

**for** \( j \leftarrow 7 \) **to** 0 **by** -1 **do**

**/* Update Significance Map during Significance Pass */**

\[
\text{SigMap} \leftarrow \text{coeff. of } W \text{ such as } w_{x,y} < 2^j \text{ and } (x,y) \notin \text{SubList};
\]

Output in B: \( \text{SigMap} \) with a tree encoder;

**/* Refinement Pass */**

\[
\text{SubList} \leftarrow \text{SigMap} \cup \text{SubList};
\]

Output in B: bit-plane \( j \) for all \( w_{x,y} \in \text{SubList} \);

**Significance pass**, tree encoding with a four-letter alphabet

- 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 and all its descendant also (Zero Tree root),
- label \( z \) if a coefficient is insignificant but all its descendants are not insignificant (Insignificant Zero).
Wavelet tree in image processing

EZW - Algorithm (3)

[Significance pass 1] pnztptttttzttttttptt
[Refinement pass 1] 1010
[Significance pass 2] ztnptttttttt
[Refinement pass 2] 100110
[Significance pass 3] zzzzzppnpnptntpntnntttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt

Figure: Example of a stream outputted by EZW algorithm of a an image of size $8 \times 8$
Wavelet tree in image processing

EZW - Entropy coding - lossless compression

**Modeling**
- compute probabilities for each coeff. (finite alphabet)
- higher is the proba. of the coeff., shorter is the bit sequence
- coeff. with proba. $p$ gets a bit sequence of length $-\log(p)$

**Coding**
1. output the dictionary: list of $(coeff., bit\ sequence)$
2. loop on input coeff., output their bit sequence

**Examples of entropy coders**
- Huffman, Arithmetic, Shannon–Fano
Outline

1 Introduction
2 Algorithmic and performance issues
3 Data structures for multiresolution
4 1D wavelet algorithms
5 Wavelets for evolution equations
6 Application 1: Vlasov 2D
7 Application 2: Vlasov 4D
Haar
Notations

- Sampled input function
  \[ c^n = \{ c^n_k \mid 0 \leq k < 2^n \} . \]

|  \( c^n_0 \) | \( c^n_1 \) | \( c^n_2 \) | \( c^n_3 \) | \( c^n_4 \) | \( c^n_5 \) | \( c^n_6 \) | \( c^n_7 \) | \( c^n_8 \) | \( c^n_9 \) | \( c^n_{10} \) | \( c^n_{11} \) | \( c^n_{12} \) | \( c^n_{13} \) | \( c^n_{14} \) | \( c^n_{15} \) |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|

Input signal \(( N = 2^4 \))

- One Haar representation: coeff. \( c^n_0 \) and \( (d^n_j)_{k \in [0, 2^{j-1}]} \), \( j = 0, n-1 \)
defined recursively by difference and average operators

\[
\begin{align*}
d^n_{k-1} &= c^n_{2k+1} - c^n_{2k}, \\
c^n_{k-1} &= c^n_{2k} + \frac{d^n_{2k+1}}{2} .
\end{align*}
\]

<table>
<thead>
<tr>
<th>( c^n_0 )</th>
<th>( d^n_0 )</th>
<th>( d^n_1 )</th>
<th>( d^n_2 )</th>
<th>( d^n_2 )</th>
<th>( d^n_2 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
<th>( d^n_3 )</th>
</tr>
</thead>
</table>

Haar coefficients \(( N = 2^4 \))
Two main type of storage (many flavors exists for multi-$d$)

- Store coefficients *level-by-level*, denoted *Mallat representation*
- Store coefficients at the location they are computed in the *in-place* algorithm

\[
\begin{array}{cccccccccccccccc}
& c^0_0 & d^0_0 & d^{1}_0 & d^{1}_1 & d^{2}_0 & d^{2}_1 & d^{2}_2 & d^{3}_0 & d^{3}_1 & d^{3}_2 & d^{3}_3 & d^{3}_4 & d^{3}_5 & d^{3}_6 & d^{3}_7 \\
\end{array}
\]

*Mallat storage*

\[
\begin{array}{cccccccccccccccc}
& c^0_0 & d^3_0 & d^2_0 & d^3_1 & d^{1}_0 & d^{3}_2 & d^{2}_1 & d^{3}_3 & d^0_0 & d^3_4 & d^{2}_2 & d^{3}_5 & d^{1}_1 & d^{3}_6 & d^{2}_3 & d^3_7 \\
\end{array}
\]

*In-place storage*

$d^i_j$ has the vector index $(1 + 2.k)2^{j_{\text{max}} - j}$ in the *in-place* version
void haar_fdwt(double* vec, int N) {
    register int itf, itc;
    register int idetail, iaverage;
    /* 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 (iaverage = 0;
            iaverage < N; iaverage += itc) {
            /* At index 'idetail': the difference 
               at 'iaverage': the average */
            idetail = iaverage + itf;
            /* PREDICT */
            vec[idetail] =
                vec[idetail] - vec[iaverage];
            /* UPDATE */
            vec[iaverage] =
                vec[iaverage] + .5 * vec[idetail];
        }
    }
}

void haar_fdwt(double* vec, double *ts, int N) {
    register int half, k;
    register int idetail, iaverage;
    /* loop from finest level to coarsest level */
    for (half = N/2; half >= 1; half /= 2) {
        /* Copy input 'vec' to temporary '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 'iaverage': the average */
            iaverage = k;
            idetail = half + k;
            /* PREDICT */
            /* UPDATE */
            vec[iaverage] =
                ts[2*k] + .5 * vec[idetail];
        }
    }
}

(a) In-place storage

(b) Mallat representation

At level $j$, we have $itf = 2^{nlevel-j}$, $itc = 2 \cdot itf$, $half = 2^j$, $2^j$ details are computed at each level $j \in [0, nlevel - 1]$. 
Dyadic grid and localization of $c_*$ in red and $d_*$ in yellow
Haar
Inverse wavelet transform

```c
void haar_idwt(double* vec, int N) {
    register int itf, itc;
    register int idetail, iaverage;
    /* 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 (iaverage = 0;
            iaverage < N; iaverage += itc) {
            /* At index 'idetail': the difference
               at 'iaverage': the average */
            idetail = iaverage + itf;
            /* UPDATE */
            vec[iaverage] = vec[iaverage] - .5 * vec[idetail];
            /* PREDICT */
            vec[idetail] = vec[idetail] + vec[iaverage];
        }
    }
}
```

```c
void haar_idwt(double* vec, double *ts, int N) {
    register int half, i;
    register int idetail, iaverage;
    /* 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 stored and at 'iaverage' is the average
               will be stored */
            iaverage = i;
            idetail = half + i;
            /* UPDATE */
            ts[2*i] =
                vec[iaverage] - .5 * vec[idetail];
            /* PREDICT */
            ts[2*i+1] =
                vec[idetail] + ts[2*i];
        }
        /* Copy temporary 'ts' to input 'vec' */
        for (i=0; i < 2*half; i++) vec[i] = ts[i];
    }
}
```

(c) In-place storage  (d) Mallat representation

Code stored in **haar_mallat**, **haar_inplace** directories
(e) In-place storage

Exercise: Much 'simpler' DWT are shown here compared to previous slides.
But are these functions \textit{cache-aware} compared to previous ones.
Evaluate $MT(N)$ for the different forward transforms.
Haar

Thresholding (Mallat representation)

```c
/* Thresholding of coefficients */
void haars_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 += 1) {
        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] = 0.0;
            } else {
                nnz_level++;
            }
        }

        printf("level %d threshold %20e
                 nnz %10d\n", level, threshold_level, nnz_level);
        nnz_tot += nnz_level;
    }

    printf("Number of non-zero coefficients :\n           %13d over %13d (%.7f percent)\n", nnz_tot, N, (100.*nnz_tot)/N);
}
```
Outline

1. Introduction
2. Algorithmic and performance issues
3. Data structures for multiresolution
4. 1D wavelet algorithms
5. Wavelets for evolution equations
6. Application 1: Vlasov 2D
7. Application 2: Vlasov 4D
Traveling signal

- Aims
  - simulate traveling signal at constant speed (transport equation)
  - use Haar 1D, periodic domain and adaptive grid
  - compare dense storage versus hash table

- Global algorithm

Read 1D input signal $In$;
Build Wavelet coeff. and threshold $W_{old} \leftarrow THR(DWT(In))$;
for $n = 1$ to $n \leq nb\_steps$ do
  Translate signal $W_{wold}$ and compute a new adaptive grid $G_{new}$;
  Build well structured tree-grid $G_{new} \leftarrow TREE(G_{new})$;
  Compute signal $F_{new}$ on grid $G_{new}$ (interpolating on $F_{old}$);
  Adaptive wavelet transform $W_{new} \leftarrow ADWT(F_{new})$;
  Thresholding $W_{new} \leftarrow THR(W_{new})$;
  Swap $W_{new}$ and $W_{old}$;
  Swap $F_{new}$ and $F_{old}$;
```c
#ifdef _HASH /* HASH TABLE */

#define FTYPE hashtable
#define _GETF(_hatab, _key, _val) {
  _VALTYPE* _pval =
    ((double*)g_hash_table_lookup(_hatab->hash, (void*)&_key));
  _val = ((_pval == NULL) ? NOTAVALUE : *(_pval));
}
#define _SETF_NEW(_hatab, _key, _val) {
  _hatab->keys[_hatab->cursize] = _key;
  _hatab->vals[_hatab->cursize] = _val;
  g_hash_table_insert(_hatab->hash,
    (gpointer)&(_hatab->keys[_hatab->cursize]),
    (gpointer)&(_hatab->vals[_hatab->cursize]));
  (_hatab->cursize)++;
}
/* ... */
#else /* DENSE ARRAY */

#define FTYPE double
#define _GETF(_hatab, _key, _val) _val = _hatab[_key];
#define _SETF_NEW(_hatab, _key, _val) _hatab[_key] = _val;
/* ... */
#endif

FTYPE *sparse_array;
```
```c
void sparse_print1d(FTYPE *sdata, int size, double dx, char *fname) {
    FILE *fd;
    _KEYTYPE i;
    _VALTYPE val;
    if ((fd = fopen(fname, "w")) == NULL) {
        printf("file %s could not be opened\n", fname);
        exit(-1);
    } else {
        for (i = 0; i < size; i++) {
            _GETF(sdata, i, val);
            if (val != NOTAVALUE)
                fprintf(fd, "%f %.3e\n", dx*i, val);
        }
    }
    fclose(fd);
}
```
```c
#traveling

/**
 * @brief Struct and typedef definitions for the traveling signal code.
 */

typedef struct 

struct shashable {
    _KEYTYPE *keys;
    _VALTYPE *vals;
    _TABTYPE *hash;
    int *ends;
    int maxsize;
    int cursize;
};

void wav_adapt_predict(FTYPE *sdata, int itc, int N, int dir) {
    _KEYTYPE i, idetail;
    int itf = itc / 2;
    double predictVal, detailVal, *pdetail;

    #ifdef _HASH
        int p, pstart, pend, ilevel;
        ilevel = wav_log2(itc);
        if (ilevel == 0) pstart = 0;
        else pstart = sdata->ends[ilevel - 1];
        pend = sdata->cursize;
        for (p=pstart; p<pend; p++) {
            i = sdata->keys[p];
            predictVal = sdata->vals[p];
            #else
            for (i = 0; i < N; i += itc) {
                GETF(sdata, i, predictVal);
            }
        #endif
        /* detail at index idetail*/
        idetail = i + itf;
        _GETF(sdata, idetail, detailVal);
        /* Verify if detail or coarse is NOTAVALUE */
        if ((predictVal == NOTAVALUE) &&
            (detailVal != NOTAVALUE)) {
            printf("! (perfect tree) in adapt_pred\n";
                 "i %d itc %d\n", (int)i, (int)itc);
            exit(2);
        }
        if (detailVal != NOTAVALUE) {
            _GETPOINTER(sdata, idetail, pdetail);
            *pdetail = dir * predictVal;
        }}

void wav_adapt_ftransform(FTYPE *sdata, int N) {
    int j;
    for (j = 2; j <= N; j = j * 2) {
        wav_adapt_predict(sdata, j, N, 1);
        wav_adapt_update(sdata, j, N, 1);
    }
}
```
Traveling signal
Build well structured wavelet tree

```c
int inline wav_iLevel(int input, int nlevel) {
    int j = 0x1<<nlevel, lev;
    for(j |= input, lev = 0;
        ((j&0x1) == 0); j >>= 1, lev++);
    return(lev);
}

int inline wav_itf(int input, int nlevel) {
    int j = 0x1<<nlevel, itf;
    for(j |= input, itf=1;
        ((j&0x1) == 0); j >>= 1, itf <<= 1);
    return(itf);
}

void inline wav_getmask(int itc, int *itcMask) {
    int i = (-1), j;
    for (j=1; j<itc; j*=2, i<<=1);    
    *itcMask = i;
}

/* Scan levels to build a correct wavelet tree */
for (itf = 1, itc = 2; itc <= N; itf *=2, itc*=2) {
    #ifdef _HASH
        for (p=0; p<snew->curSize; p++) {
            i = snew->keys[p];
            valdet = snew->vals[p];
            myitf = wav_itf(i, nlevel);
            if ((myitf == itf) && (valdet != NOTAVALUE)) {
    #else
        for (i = itf; i < N; i+= itc) {
            _GETF(snew, i, valdet);
            if (valdet != NOTAVALUE) {
    #endif
                ioca = i-itf;
                _GETF(snew, ioca, val);
                if (val == NOTAVALUE) {
                    _SETF_NEW(snew, ioca, 0.);
                }
            }
        }
    }
```
Traveling signal
Dynamic remeshing in time: advection+refinement

/* scan all levels */
for (itf = 1, itc = 2, level = 0; itc <= N;
    itf *= 2, itc *= 2, level++) {
    atf = itf / 2; if (atf < 1) atf = 1;
    /* higher bits mask corresponding to atf */
    wav_getmask(atf, &maskatf);
    /* traversal at level 'itc' */
#else HASH
    pstart = pend;
    pend = swav->ends[level];
    for (p=pstart; p<pend; p++) {
        i = swav->keys[p];
        val = swav->vals[p];
    }
#else
    for (i = itf; i < N; i+= itc) {
        _GETF(swav, i, val);
    } } } }

if (val != NOTAVALUE) {
    /* advect grid point */
    int disp = (i + floordisp + N)%N;
    /* get a 'divide atf' grid point */
    int disp &= maskatf;
    /* SAME AS: int disp /= atf; int disp += atf; */
    /* refinement procedure, insert
     patchsize points for each initial point */
    beginpatch = N+int disp - atf;
    for (c = 0; c < patchsize; c++) {
        j = (beginpatch + c*atf)%N;
        _GETF(snew,j,val);
        if (val == NOTAVALUE) {
            _SETF_NEW(snew,j,0.);
        } } }}}}
Outline

1 Introduction
2 Algorithmic and performance issues
3 Data structures for multiresolution
4 1D wavelet algorithms
5 Wavelets for evolution equations
6 Application 1: Vlasov 2D
7 Application 2: Vlasov 4D
Application 1: Vlasov 2D

OBIWAN : Vlasov solver using a Wavelet based Adaptive Mesh Refinement

Matthieu Haefele, Guillaume Latu
Michael Gutnic, Eric SonnenDrücker

CALVI project
Application 1: Vlasov 2D

Cadre, modélisation

Vlasov equation

\[ \frac{\partial f}{\partial t} + \vec{v} \cdot \nabla_x f + \left( E + \vec{v} \times B \right) \cdot \nabla_v f = 0 \]

- \( f(\vec{x}, \vec{v}, t) \): particle distribution function at time \( t \) in phase space, \( (\vec{x}, \vec{v}) \in \mathbb{R}^d \times \mathbb{R}^d \) with \( d=3 \)
- \( E(\vec{x}, t), B(\vec{x}, t) \): electromagnetic field

Applications:
- Plasmas physic
- Particle physic
Cadre, modélisation

**Reduced model (d=1)**

- **Objective:**
  Validate a new adaptive numerical scheme

\[
\frac{\partial f}{\partial t} + v \frac{\partial f}{\partial x} + \left( E_{\text{self}}(x, t) + E_{\text{app}}(x, t) \right) \frac{\partial f}{\partial v} = 0
\]

- **Non linear PDE**
Splitting de l’équation

Pour décrire une évolution en temps, on peut résoudre l’équation en deux temps

Splitting en \( V \) (\( x \) constant)
\[
\frac{\partial f}{\partial t} + E_{applied + self} (x, t) \frac{\partial f}{\partial v} = 0
\]

Splitting en \( X \) (\( v \) constant)
\[
\frac{\partial f}{\partial t} + v \frac{\partial f}{\partial x} = 0
\]

On peut aussi résoudre l’équation sans splitting
Application 1: Vlasov 2D

Cadre, modélisation

**Global algorithm**

\[ E_{0}, f_{0} \]

\[ f_{i} \rightarrow f_{i+1} \]

\[ \rho_{i+1} \rightarrow E_{\text{self}}^{i+1} \]

\[ E_{\text{app}}^{i+1} \]

Electrical field solved with Poisson equation

\[ \frac{dE}{dx} = \rho(x, t) = \int f(x, v, t) dv \]
Property of the Vlasov equation

- $f$ is constant along particular curves of the phase space: the characteristics
- Characteristics can be computed explicitly
- One 2D advection
- OR
- Two 1D advections (splitting method)

Two major kind of numerical schemes
- Particle-In-Cell methods [Birdshall'85]
- Grid methods [Filbet'01]
Cadre, modélisation

**Schéma numérique**

- Pour chaque point \( p_i \) du maillage en \( t_1 = t_0 + dt \) :
  - *Rechercher* l'origine \( o_i \) de la caractéristique en \( t_0 \)
  - *Interpoler* la valeur en \( o_i \) au pas de temps précédent \( t_0 \)

- Version sans splitting -> interpolation 2D

\[ f(.,t) \text{ connu} \]
\[ \text{Interpolée}(f(o_i, t_0)) = f(p_i, t_1) \]
Cadre, modélisation

**Problématique**

- Limitations codes denses utilisant des grilles :
  - Nécessité d'une grande quantité de mémoire
    - Ex en 2D : grille de $16384^2 \rightarrow 2$ Go
    - Ex en 4D : grille de $256^4 \rightarrow 32$ Go

- Calculs « inutiles » dans certaines régions comportant peu d’informations

- Utilisation maillage adaptatif :
  - Structures de données et calculs *profitent* du creux possibilité de réduire coûts en mémoire et en calcul.
Appearance of fine structures  → fine grids needed

Idea: use a multiresolution analysis to adapt the numerical method
Semi-Gaussian beam evolution in periodic focusing channel

- Potassium ions
- Beam energy 80 keV
- Periodic focusing field of the form $\alpha(1 + \cos 2\pi z/S)$.
- Tune depression 0.17
Application 1: Vlasov 2D
Application 1: Vlasov 2D

"fSparASCII0492.out"
Schéma adaptatif

**MultiResolution Analysis (MRA)**

- For a grid $G_j$
  - $c^j_k = f(x^j_k)$
  - Projection operator: $c^j_k = c^{j+1}_{2k}$
  - Prediction operator: $c^{j+1}_{2k+1} = P_{2N+1}(x^{j+1}_{2k+1})$
    with $P$ the Lagrange polynomial
  - The detail is defined as $d^j_k = c^{j+1}_{2k+1} - P(x^{j+1}_{2k+1})$
- We have defined a MRA!

$$f(x) = \sum_{k=0}^{N} c^0_k \varphi^0_k(x) + \sum_{j=1}^{n\text{level}} \sum_{k=0}^{N} d^j_k \psi^j_k(x)$$

**$d$ is small ($d < \varepsilon$) where $f$ is regular**
Application 1: Vlasov 2D

Schéma adaptatif

**MultiResolution Analysis (MRA)**

- Wavelet transform algorithm

- Need a well formed tree of wavelet coefficients
Application 1: Vlasov 2D

Schéma adaptatif

**MultiResolution Analysis (MRA)**

- **Error control**

\[ \left\| F_{\text{dense}} - F_{\text{adapt}} \right\|_{L_1, L_2, L_\infty} < \varepsilon \]

Dense representation

Wavelet representation
Schéma adaptatif

**Grille adaptative 2D**

Niveau 0

Niveau 1

Niveau 2
Schéma adaptatif

**Schéma 2D**

produit tensoriel de transformations 1D

Point appartenant
à une ligne grossière

Point appartenant
à une colonne grossière

Point appartenant
ni à une colonne
ni à une ligne grossière
Etude algorithmique

3 types de données utilisés

- Structure arborescente $G$ :
  nécessaire pour la méthode basée sur ondelettes
  nombre de nœuds : $\#G$

- Représentation en ondelettes : $D$
  ensemble de coefficients (il y en a $\#D$)

- Fonction de densité $F$ :
  connue pour certains points (il y en a $\#F$)
Application 1: Vlasov 2D

Algorithme adaptatif

Splitting adaptatif

Splitting

1A) Prédiction : points du maillage adaptatif advectés,
(on ne connaît pas encore la valeur)
Ajout de points autour de ces coeff. advectés
(raffinement du maillage)
Structuration de l’arborescence d’ondelette

1B) Advection arrière
(reconstruction des valeurs des densités
puis interpolation : Inverse Wavelet Transform)

1C) Calcul des coefficients d’ondelette \((t+dt)\)
Adaptative wavelet Transform + Compression
Algorithme adaptatif

1A) Prédiction et Raffinement

Idée générale :
- Pour tous les détails (temps \( t \)) dont le coefficient > seuil
- Suivre les caractéristiques (advection avant)
- Déterminer les points dont on souhaite connaître le détail
- Ajouter ces points dans une structure adaptative \( G \)

Advection en avant

Prédiction

Points ajoutés au même niveau que le point de départ (●)
Points ajoutés au niveau plus fin pour raffiner le maillage(●)
1A) Structuration des coefficients

Losque l’on calculera le détail au point (●)
Où faudra-t-il connaître la valeur de $f$ ?

Implicite à la méthode : Il faut que le maillage contienne une arborescence de coefficients d'ondelettes
Ajout de points dans $G$ pour obtenir cette structure d'arbre
1B) Advection arrière

Idée générale :

- Pour tous les points \( p_i \) de \( G \)
- Rechercher l'origine \( o_i \) de la caractéristique
- Interpoler la valeur en \( o_i \) à l'aide d'une représentation de la fonction de densité au pas de temps précédent

Problème : on ne possède pas les valeurs de la distribution \( f \)
Algorithme adaptatif

1B) Reconstruction de $f$

Informations disponibles :
- Coefficients d’ondelette $D_{t-dt}$ au pas de temps précédent
- Certains points de la distribution $f$ au pas de temps précédent

Deux solutions :
- Inverse Wavelet Transform
  Reconstituer $f$ dense à partir de $D_{t-dt}$ puis interpoler
  Coût en calcul et en mémoire
- Interpolation dans l'espace des ondelettes
  Calcul complexe
  Difficulté à optimiser
Algorithme adaptatif

### 1C) IWT

- Reconstruction de $f$ dense à partir de sa décomposition en ondelettes

- Reconstruire les différentes résolutions de $f$ avec une boucle sur les niveaux $j$ du plus grossier $j=0$ vers le niveau fin $j=\text{max}-1$
  - Interpolyer la valeur aux points de niveau $j-1$
  - Ajouter les détails de ce niveau si ils sont présents

- Dépendance de données
  - La reconstruction d'un point au niveau le plus fin nécessite de reconstruire des points aux différentes résolutions
Etude algorithmique

Algorithmme global

- Pas de temps $t$
  - Splitting $V$

$D_t \rightarrow G$  1A) Prédiction : coeff. d’ondelette (en $t$) advectés en $V$,

$G \rightarrow G$ Ajout de points autour de ces coeff. advectés
  (raffinement)

$G \rightarrow G$ Structure d’arbre complétée

$(G, D_t) \rightarrow F_{t + dt}$  1B) Calcul de la densité sur les nœuds de l’arbre
  (reconstruction des valeurs des densités puis interpolation : IWT)

$(G, F_{t + dt}) \rightarrow D_{t + dt}$  5C) Calcul des coeff. d’ondelette ($t + dt$) AWT

- Splitting $X$ (idem Splitting $V$)

- Calcul du champ
Complexités algorithmiques (1)

1) Prédiction et 2) Raffinement

- Parcours d’une arborescence de coefficients d’ondelette ($D_t$)
- Complexité linéaire en $\#D_t$

3) Construction d’une structure arborescente

- Parcours par niveau (du + fin vers le + grossier)
- Dépendance des données :
  - un niveau $j$ utilise celui qui vient d’être traité précédemment $j+1$
  - Complexité linéaire en $\#G$
Complexités algorithmiques (2)

4) Calcul de la densité pour les nœuds de l’arbre :
Deux possibilités :
- A) reconstruction de chaque densité pour #G points, complexité :
  
  
  [(taille du filtre: 4 ou +)*(nb dimensions: 2)]^(niveau du point)
- B) reconstruire tous les points (le calcul des points grossiers sont factorisés)
  parcours par niveau (du + grossier vers le + fin), complexité :
  (nb points grille fine)*(taille du filtre)*(nb dimensions)

Comparaison :
- A) complexité totale moindre si très peu de points,
  mais de nombreux accès aléatoires (coûteux en temps d’accès mémoire)
- B) Si les points sont stockées dans un tableau en mémoire,
  possibilité de profiter de la rapidité des mémoires caches
Etude algorithmique

Complexités algorithmiques (3)

5) Calcul des coefficients d’ondelette \((t+dt)\).

- Parcours par niveau (du + fin vers le + grossier) de \(F_{t+dt}\).
  Pour chaque point, calcul du détail (ou du coefficient d’ondelette), complexité :
  \[
  \#G \times (\text{taille du filtre: 4 ou +})
  \]

Points communs des étapes précédentes :

- Complexité des étapes 1) 2) 3) 5) en \#G : nombre de nœuds
- Pour chaque point, on doit pouvoir accéder rapidement aux points de même niveau, du niveau au-dessus, niveau au-dessous.
- Nécessité de réaliser des parcours par niveau
  \[
  \text{aspect séquentiel qui nécessitera des synchronisations des algorithmes parallèles}
  \]
Optimisations mises en œuvre

α) Parties coûteuses → améliorer le code

β) Changer les structures de données, pour :
- ajout rapide d’un nœud de l’arbre (2 ou 3 références mem.)
- lecture rapide des structures (aléatoire ou en séquence)
- prendre en compte explicitement le « creux »
- parcours par niveau très rapide
- structures efficaces avec peu ou beaucoup de données
Optimisation

Optimisation parties coûteuses α (1)

Prises de performances (profiling)

- grille fine de grande taille (2048x2048), peu de détails dans $D_t$
  - 92% du temps → [4-calcul de la densité]
  - 7% du temps → calcul du champ
- grille fine de grande taille (2048x2048), nombreux détails dans $D_t$
  - 45% du temps → [4-calcul de la densité]
  - 4% du temps → calcul du champ
  - 51% du temps → autres calculs (#G)
Optimisation

Optimisation parties coûteuses α (2)

Principes utilisés pour réaliser les optimisations :

- Appels de fonction coûteux :
  allocation de variables, changements de contextes
- Les caches (L1, L2) accélèrent l’accès aux données
  - chargement par lignes de cache (localité spatiale)
  - localité temporelle
- L’accès à des suites de données contiguës en mémoire est accélérée par l’architecture des ordinateurs actuels.

Réduction du temps de : [4-calcul de densité]

Réécriture partielle du code :

- réduction du nb d’appels de fonctions
  → utilisation de macros
- favoriser les lectures/écritures de suites d’octets contigus en mémoire (localité spatiale) → optimisation nids de boucles (inversions)
Optimisation

Optimisation parties coûteuses α (3)

Performances sur un pas de temps (juillet 2004) :
le temps de [4-calcul de densité] est divisé
par un facteur 4

grille fine de grande taille, peu de détails dans $D_t$
(1.8s pour 2500 nœuds contre 6.3s avant –
Athlon MP 2000+)

71 % du temps $\rightarrow$ [4-calcul de la densité]
28 % du temps $\rightarrow$ calcul du champ

grille fine de grande taille, nombreux détails dans $D_t$
(8s pour 250 000 nœuds contre 13s avant)

15 % du temps $\rightarrow$ [4-calcul de la densité]
6 % du temps $\rightarrow$ calcul du champ
Optimisation

Structures de données (1)

Les tables de hachages utilisées sont :

- efficaces pour →
  - prendre en compte explicitement le « creux »
  - l’ajout rapide d’un nœud de l’arbre (accès aléatoire)
  - accès aléatoire rapide

- peu performantes pour →
  - parcours par niveau avec accès aux niveaux adjacents
  - rester efficace avec beaucoup de données

Concernant les points négatifs, on souhaite bénéficier des caches :

- les points proches spatialement doivent l’être dans la structure
- les parcours des niveaux grossiers sont extrêmement fréquents, il doivent être stockés de manière « dense »
Optimisation

**Structures de données (2)**

- **Solution : Stockage creux avec 1 niveau d’indirection**

  ![Diagram showing a grid with pointers and values](image)

  - Tableau de valeurs (niveaux 0 à L)
  - Tableau de pointeurs
  - Tableaux de valeurs (cellules) (niveaux L+1 à nblevel)

- **Avantages et inconvénients :**
  - faible nombre d’indirections (une) → lecture en accès aléatoire, coût faible
  - gestion manuelle de la zone mémoire des cellules
    → l’accès à des cellules voisines bénéficie des localités spatiales et temporelles
    → reste efficace avec de nombreuses cellules
  - surcoût de la méthode → le tableau de pointeurs
    → la gestion de l’indirection
Optimisation

**Structures de données (3)**

- Changement structures de données → refonte,
- Modification de certains algo. pour version //
  - [4-calcul de densité] est réécrit : algo. par bloc
    → surcoût dû à des calculs redondants
    → accélération possible : bloc tient dans le cache

- Le calcul du champ n’est plus fait sur la grille fine,
  mais directement à partir des ondelettes
Performances sur un pas de temps (janvier 2005) : le temps de chaque partie a été réduit, sauf [5-Calcul des coefficients d’ondelette] (travail en cours)

grille fine de grande taille, peu de détails dans $D_t$
(1.3s pour 2500 nœuds contre 1.8s avant)

74 % du temps → [4-calcul de la densité]
5 % du temps → calcul du champ

grille fine de grande taille, nombreux détails dans $D_t$
(2.8s pour 250 000 nœuds contre 8s avant)

42 % du temps → [4-calcul de la densité]
4 % du temps → calcul du champ
Conclusion Optimisations α, β

Amélioration notable sur un pas de temps grille fine de grande taille, beaucoup de détails
obiwan 13s → 2.8s obitwo
Numerical scheme

Idea: Apply the semi-Lagrangian method on an adaptive mesh

Init: electrical field $E_0$, distribution function $F_0$, wavelet coefficient $D_0$

For all steps $t$ required:

1 Splitting in $v$-direction
   1A Build the adaptive grid $G_t$
      - Prediction step
      - Make a well formed tree
   1B Compute the distribution function $F_t$
      - Inverse Wavelet Transform
      - Backward advection on $A_t$
   1C Adaptive Wavelet Transform and compress

2 Splitting in $x$-direction (idem)

3 Compute electrical field $D_t$
Algorithmic complexities

**Hypothesis**: At time step $t$, each adaptive struct. holds

$\sim S$ coefficients, with $S < N^2$

Algorithmic complexities (reads, writes, operations)

<table>
<thead>
<tr>
<th>Steps</th>
<th>#reads</th>
<th>#operations</th>
<th>#writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build the adaptive grid (1A or 2A)</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
</tr>
<tr>
<td>Compute distrib. function (1B or 2B)</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
</tr>
<tr>
<td>Compute wavelet transform (1C or 2C)</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
<td>$O(S)$</td>
</tr>
<tr>
<td>Compute electrical field (3)</td>
<td>$O(N+S)$</td>
<td>$O(N+S)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

Each part of a time step should be done in parallel
Algorithmes et structure de données

**Data structures**

- **Constraints:**
  1. Traversal of large adaptive structure
  2. Some values (levels 0, 1) are used very often

- **Previously,** hash tables were used → not so efficient

- **Solution:**
  → sparse structure with one indirection level

- Dense 2D array for values (levels 0 to \( L \))

- Dense 2D array for indirection pointers (levels 0 to \( L \))

- Fine blocks of \( K^2 \) values (levels \( L+1 \) to \( nblevel \))
Data structures

Management of sparsity → not optimal

Low cost to read one element in sparse structure
→ 1 memory access for levels $0 \rightarrow L$
→ 2 memory accesses for levels $L+1 \rightarrow nblevel$

Spatial and temporal locality improved
significant reduction of exec. time (vs. hash tables)
Parallélisation

**Data partitioning**

- Example of sparse data to distribute:

  ![Data partitioning diagram](image)

  - no fine block
  - fine blocks (red+ brown colors)

- MRA involves: complex and large mem. access patterns
  → On several procs: distant accesses in reading, writing
Application 1: Vlasov 2D

Parallélisation

Target paradigm & machine

- Complex data dependencies & medium grain parallelism:
  -> a programming model without explicit comm.
  -> targeted a shared memory architecture

- OpenMP chosen (one thread per processor)

- Two machines (CINES, Montpellier, France):
  - IBM SP3 NH2 (one node of 16 procs. used)
  - SGI Origin 3800 (up to 64 procs. used)
Data & computation placement

- In each step of our algorithms, the $x$-loop is parallel
  → A thread uses extensively the same set of blocks
- Block cyclic distribution of columns of fine blocks onto threads
Parallélisation

**Optimization of subdomain size**

- Subdomains lead to different computation costs → implies load imbalance → try to increase the number $B$ of subdomains
- Most of our algorithms benefit from spatial locality → numerous subdomains increase the number of distant memory accesses → memory bandwidth limitation → try to decrease the number $B$ of subdomains
- Find a balance for the value of $B$
Parallélisation

Parallel overheads

- Load imbalance at each parallel step
- Numerous readings on distant memory at each parallel step
- Writings on distant memory & concurrent accesses in “prediction” and “well formed tree” steps
- Two all-to-all implicit communication patterns during “splitting in x-direction” step
Parallélisation

Performance analysis

Timing profile and speedups of one typical time step on a standard test case (B=64) on SGI Origin 3800 machine

<table>
<thead>
<tr>
<th>Steps</th>
<th>Number of procs.</th>
<th>1</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>1A Prediction</td>
<td></td>
<td>0.7659</td>
<td>14.6</td>
<td>24</td>
<td>45</td>
</tr>
<tr>
<td>1A Well formed tree</td>
<td></td>
<td>1.4023</td>
<td>15.4</td>
<td>30</td>
<td>49</td>
</tr>
<tr>
<td>1B Compute distrib. function</td>
<td></td>
<td>5.1605</td>
<td>14.8</td>
<td>28</td>
<td>54</td>
</tr>
<tr>
<td>1C Wavelet transform</td>
<td></td>
<td>0.6782</td>
<td>17.3</td>
<td>33</td>
<td>62</td>
</tr>
<tr>
<td>2A Prediction</td>
<td></td>
<td>1.1447</td>
<td>14.6</td>
<td>25</td>
<td>47</td>
</tr>
<tr>
<td>2A Well formed tree</td>
<td></td>
<td>2.4335</td>
<td>15.3</td>
<td>28</td>
<td>52</td>
</tr>
<tr>
<td>2B Compute distrib. function</td>
<td></td>
<td>4.9924</td>
<td>13.3</td>
<td>23</td>
<td>33</td>
</tr>
<tr>
<td>2C Wavelet transform</td>
<td></td>
<td>0.8598</td>
<td>14.3</td>
<td>23</td>
<td>34</td>
</tr>
<tr>
<td>3 Compute Field</td>
<td></td>
<td>0.2523</td>
<td>15.5</td>
<td>24</td>
<td>45</td>
</tr>
</tbody>
</table>
Parallélisation

Performance analysis

Overall performance of one simulation on a standard test case (B=64)

<table>
<thead>
<tr>
<th>Number of procs.</th>
<th>1</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (sec.)</td>
<td>speedup</td>
<td>speedup</td>
<td>speedup</td>
</tr>
<tr>
<td>SGI Origin 3800</td>
<td>15539</td>
<td>14,2</td>
<td>25,1</td>
<td>42,2</td>
</tr>
<tr>
<td>IBM SP3 NH2</td>
<td>19644</td>
<td>14,6</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Main problems:

- Load balancing
- Output writing time (on hard drive)
Parallélisation

**Comparaison dense vs. adaptatif**

Deux codes : LOSS2D vs. Obiwan2D

Temps pour 1 itération, 8 processeurs sur un cas test Semi Gauss périodique
[ maillage = $2^K \times 2^K$]
[ précision $10^{-5}$ en adaptatif]

<table>
<thead>
<tr>
<th>K</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taille maillage (MO)</td>
<td>8</td>
<td>32</td>
<td>128</td>
<td>512</td>
<td>2048</td>
</tr>
<tr>
<td>LOSS2D (sec.)</td>
<td>0.11</td>
<td>0.44</td>
<td>2.70</td>
<td>24.20</td>
<td>138.60</td>
</tr>
<tr>
<td>Obiwan2D (sec.)</td>
<td>0.33</td>
<td>0.83</td>
<td>2.46</td>
<td>3.70</td>
<td>8.90</td>
</tr>
</tbody>
</table>

Avantage indéniable à l’adaptatif lorsque K est élevé.
Applicaton 1: Vlasov 2D (with N. Besse)

Parametric Instability (2)

\[ \text{time} = 86.778 \omega_p^{-1} \]

\[ \text{time} = 104.130 \omega_p^{-1} \]

Figure: Evolution of the distribution function in the phase space \((x, p_x)\) during the saturation phase for the parametric instability.
Vertical: \(p_x\)-axis, Horizontal: \(x\)-axis

Figure: Evolution of the distribution function in the phase space \((x, p_x)\) during the saturation phase for the parametric instability.
Vertical: \(p_x\)-axis, Horizontal: \(x\)-axis
Application 1: Vlasov 2D (with N. Besse)

Self-Induced transparency and KEEN waves

- Plot of $f(t, x, p_x)$
- Self-Induced transparency
- time $= 205.4 \omega_p^{-1}$

- Plot of $f(t, x, p_x)$
- KEEN waves
- time $= 1230.4 \omega_p^{-1}$

$P_{osc} = 1.25$, $T_e = 100\, kev$, $n_0/n_c = 1.20$, Mesh: $2^8 \times 3(x) \times 2^6 \times 3(p_x)$
Application 1: Vlasov 2D (with N. Besse)

Laser wake

- Plot of $f(t, x, p_x)$
- Laser wake
- Time $= 109.8 \omega_p^{-1}$

Laser wake

- Plot of $f(t, x, p_x)$
- Laser wake
- Time $= 175.8 \omega_p^{-1}$

$P_{osc} = \sqrt{3/2}$, $T_e = 3\text{kev}$, $n_0/n_c = 0.1$, Mesh: $2^{10+3} \times 2^{8+3}$
Perspectives Obiwan 4D (2006)

Problèmes actuels :

- Surcoût trop important de la méthode adaptative interpolations -> faible pourcentage du temps total
- Creux des structures “moins flagrant” qu’en 2D
- Remplissage important
- Parallélisation 1D (pour l’instant) peu scalable

Idées pour concurrencer LOSS4D :

1. Algorithmes économes en bande passante mémoire
2. Compression 2D et non 4D
Outline

1. Introduction
2. Algorithmic and performance issues
3. Data structures for multiresolution
4. 1D wavelet algorithms
5. Wavelets for evolution equations
6. Application 1: Vlasov 2D
7. Application 2: Vlasov 4D
In the sequel we shall consider only the collision-less Vlasov-Maxwell equations

$$\partial_t f + \mathbf{v} \cdot \nabla_x f + \mathbf{F} \cdot \nabla_v f = 0$$

$$\mathbf{F} = \mathbf{E} + \mathbf{v} \times \mathbf{B}$$

with \((E, B)\) the electro magnetic fields, solutions of Maxwell equations. The source terms are computed by

$$\rho = \int f \, d\mathbf{v}, \quad \mathbf{J} = \int f \mathbf{v} \, d\mathbf{v}$$

In some cases Maxwell’s equations can be replaced by a reduced model like Poisson’s equation
Curse of dimensionality:

$N^d$ grid points needed in $d$ dimensions on uniform grids.

Number of grid points grows exponentially with dimension → killer for Vlasov equation where $d$ up to 6.

Memory needed

- In 4D, $256^4$ grid $\rightarrow$ 32 GB
- In 6D, $64^6$ grid $\rightarrow$ 512 GB

Adaptive algorithm is a must in higher dimensions
The Vlasov equation (2D space, 2D velocity, d=4) is split. We solve it, using elementary 1D advection equations:

\[
\begin{align*}
\partial_t f + v_x \partial_x f &= 0 & (\hat{x} \text{ operator}) \\
\partial_t f + v_y \partial_y f &= 0 & (\hat{y} \text{ operator}) \\
\partial_t f + \dot{v}_x \partial_{v_x} f &= 0 & (\hat{v}_x \text{ operator}) \\
\partial_t f + \dot{v}_y \partial_{v_y} f &= 0 & (\hat{v}_y \text{ operator})
\end{align*}
\]

One time step of simulation is composed of:
- successive splittings in each dimension (advection steps)
- field solving using the source term \( \rho = \int f \, dv \)
- Main cost of the application: interpolations
Adaptive scheme: wavelets
Hierarchical approximation

- Decomposition of \( f(x, y, v_x, v_y) \) in hierarchical basis

\[
f(z) = \sum_{k} c_{j_0 k} \phi_{j_0 k} + \sum_{j=\text{coarse level}} \sum_{k} d_{j k} \psi_{j k}
\]

- Coefficients \( c_{j_0 k} = f(z_{j_0 k}) \)

- Detail coefficients

\[
d_{j k} = f(z_{j k}) - \text{Lagrange interp. in } z_{j k} \text{ at level } j - 1
\]

Details \( d \) are small if \( f \) is locally smooth

- Only grid points where \( f \) varies most are kept, others are eliminated

- Usually:

\[
\text{Nb. of } c \text{ and } d \text{ coeff. } \ll \text{Nb. of points in uniform grid } N^4
\]
Input: $shifts_X (v_X)$ (displacements in $X$ direction)
Input: $D^t$ (Wavelet coefficients, details)
Output: $D^{t+\varepsilon}$

Adapted Grid Prediction: $shifts_X, D^t \mapsto Adapt. Grid^{t+\varepsilon}$
Maketree: $Adapt. Grid^{t+\varepsilon} \mapsto Adapt. Grid^{t+\varepsilon}$
Backward Advection: $-shifts_X, Adapt. Grid^{t+\varepsilon}, D^t \mapsto F^{t+\varepsilon}$
Wavelet Transform: $F^{t+\varepsilon} \mapsto D^{t+\varepsilon}$

Algorithm 1: Adaptive advection
Parallel algorithm for Wavelet Transform

**Input:** $F^{\text{t+e}}$ (distribution function known on the adaptive grid)

**Output:** $D^{\text{t+e}}$ (wavelet coefficients, details)

1. $D^{\text{t+e}} \leftarrow F^{\text{t+e}}$; $iF \leftarrow 1$; $iC \leftarrow 2$;
2. for $j \leftarrow nblev$ to 1 by -1 do
   a. for $d \leftarrow 1$ to 4 do
      i. $s[1:4] \leftarrow 0$; $t[1:4] \leftarrow IF$; $e[1:4] \leftarrow 2^{j0+nblev} - 1$; $s_d \leftarrow IF$;
      ii. $t_d \leftarrow IC$;
      b. for $i_0 \leftarrow s_0$ to $e_0$ by $t_0$ do in parallel
         i. for $i_1 \leftarrow s_1$ to $e_1$ by $t_1$ do in parallel
            a. for $i_2 \leftarrow s_2$ to $e_2$ by $t_2$ do in parallel
               i. for $i_3 \leftarrow s_3$ to $e_3$ by $t_3$ do in parallel
                  a. if $w_{i_0}^{\text{t+e}} \in D^{\text{t+e}}$ then
                     i. $m[1:4] \leftarrow i[1:4]$; $\tau \leftarrow 0$;
                     ii. for $z \leftarrow h_1$ to $l_2$ do
                          a. $m_d \leftarrow i_d + z \cdot IC - IF$;
                          b. $\tau \leftarrow \tau - h(z) \cdot w_{m_d}^{\text{t+e}}$
                          c. $w_{i_0}^{\text{t+e}} \leftarrow w_{i_0}^{\text{t+e}} + \tau$
      c. $iF \leftarrow 2iF$; $iC \leftarrow 2iC$;
Main problems

- Find a 4D sparse structure to store wavelet coefficients:
  - That preserves sparsity
    (even if sparsity develops in only 1D)
  - With efficient traversal of large adaptive structure
  - Leading to spatial and temporal locality
    (to use cache memory)
  - Adaptive algorithms involve: complex and large memory access patterns → random accesses must be quick

- Evaluate different load balancing strategies
Test case (Beam): Alternate Gradient Focusing

- 40 mA, 1 MeV potassium beam in alternating gradient lattice, $\Delta t = 0.000464s$
- Generates sparsity in 4D data structure
- Raises problem of managing work distribution
Adaptive data structure

- Analogy with binary tree, quad-tree used to partition 1D, 2D → a hexadeca-tree stores the 4D wavelet decomposition
- In one node: wavelet coeff., links towards direct descendants
- Reduce memory usage by pruning the tree.
  Example on quad-tree:

  ![Diagram of a quad-tree](image)

  Memory representation

  ▶ Wavelet coefficient
  ▼ Indirection pointer
  ▶ Non-smooth area
  ▶ Smooth area

  With such trees → many indirections to go to finest level.
  Solution: big nodes that encapsulate two levels of the hexadeca-tree
Parallelization of substeps in Obiwan 4D

- **Choice:** OpenMP, Shared Memory programming
- **Constraint:**
  - writing in the 4D data: avoid concurrent accesses
- **Grain of computation:**
  - one coarse point and its descendants

<table>
<thead>
<tr>
<th>Substep</th>
<th>Number of // loops</th>
<th>computation grain</th>
</tr>
</thead>
<tbody>
<tr>
<td>F_Prediction</td>
<td>2</td>
<td>2D slice of coarse pts and descendants</td>
</tr>
<tr>
<td>F_Maketree</td>
<td>3</td>
<td>1D slice of coarse pts and descendants</td>
</tr>
<tr>
<td>B_Advection</td>
<td>3</td>
<td>1D slice of coarse pts and descendants</td>
</tr>
<tr>
<td>Wavelet transform</td>
<td>3</td>
<td>1D slice of coarse pts and descendants</td>
</tr>
<tr>
<td>Field computation</td>
<td>3 (partly seq.)</td>
<td>1D slice of coarse pts and descendants</td>
</tr>
<tr>
<td>Diagnostics</td>
<td>almost sequential</td>
<td>1D slice of coarse pts and descendants</td>
</tr>
</tbody>
</table>

**Table:** Loops parallelization in each substep
Performance issue
Scalability AGF test case (Static LB)

- **Static load balancing strategy**
  (mapping coarse points on processors)

<table>
<thead>
<tr>
<th>Nb. procs.</th>
<th>1</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parts in one time step</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F_Prediction</td>
<td>13.670 (1.0)</td>
<td>3.706 (3.7)</td>
<td>2.467 (5.5)</td>
</tr>
<tr>
<td>F_Maketree</td>
<td>19.351 (1.0)</td>
<td>5.100 (3.8)</td>
<td>2.253 (8.6)</td>
</tr>
<tr>
<td>B_Advection</td>
<td>200.508 (1.0)</td>
<td>52.211 (3.8)</td>
<td>20.866 (9.6)</td>
</tr>
<tr>
<td>Wavelet transform</td>
<td>31.887 (1.0)</td>
<td>7.982 (4.0)</td>
<td>3.384 (9.4)</td>
</tr>
<tr>
<td>Field computation</td>
<td>1.464 (1.0)</td>
<td>0.372 (3.9)</td>
<td>0.099 (14.8)</td>
</tr>
<tr>
<td>Diagnostics</td>
<td>2.086 (1.0)</td>
<td>0.921 (2.3)</td>
<td>0.695 (3.0)</td>
</tr>
<tr>
<td>Complete Iteration</td>
<td>269.0 (1.0)</td>
<td>70.3 (3.8)</td>
<td>29.8 (9.0)</td>
</tr>
</tbody>
</table>

**Table:** Computation time and **speedup** (indicated between parentheses) averaged on 3 iterations - 128⁴ test case, IBM Power5 16-way node
Performance issue
Scalability AGF test case (Dynamic LB)

- Dynamic load balancing strategy
  (mapping coarse points on processors)

<table>
<thead>
<tr>
<th>Nb. procs.</th>
<th>1</th>
<th>4</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parts in one time step</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F_Prediction</td>
<td>13.687 (1.0)</td>
<td>4.016 (3.4)</td>
<td>2.446 (5.6)</td>
</tr>
<tr>
<td>F_Makeetree</td>
<td>19.395 (1.0)</td>
<td>4.918 (3.9)</td>
<td>1.904 (10.2)</td>
</tr>
<tr>
<td>B_Advection</td>
<td>201.101 (1.0)</td>
<td>50.120 (4.0)</td>
<td>13.773 (14.6)</td>
</tr>
<tr>
<td>Wavelet transform</td>
<td>31.797 (1.0)</td>
<td>7.841 (4.1)</td>
<td>2.342 (13.6)</td>
</tr>
<tr>
<td>Field computation</td>
<td>1.464 (1.0)</td>
<td>0.374 (3.9)</td>
<td>0.099 (14.9)</td>
</tr>
<tr>
<td>Diagnostics</td>
<td>2.088 (1.0)</td>
<td>0.923 (2.3)</td>
<td>0.688 (3.0)</td>
</tr>
<tr>
<td>Complete Iteration</td>
<td>269.5 (1.0)</td>
<td>68.2 (4.0)</td>
<td>21.3 (12.7)</td>
</tr>
</tbody>
</table>

Table: Computation time and speedup (indicated between parentheses) averaged on 3 iterations - 128⁴ test case, IBM Power5 16-way node
Performance issue
Memory Scalability
Obiwan 4D code versus Loss 4D

- Test case AGF (threshold = 1e-4)
- Compare results against a non adaptive code: Loss 4D
- Zoom on one time step on one SMP node
  - 16 processors, 27 GB

<table>
<thead>
<tr>
<th>Domain size</th>
<th>128⁴</th>
<th>256⁴</th>
<th>512⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Memory (Dense code)</td>
<td>2 GB</td>
<td>32 GB</td>
<td>512 GB</td>
</tr>
<tr>
<td>Total Memory (Adaptive code)</td>
<td>0.45 GB</td>
<td>2.32 GB</td>
<td>14.1 GB</td>
</tr>
<tr>
<td>Time (Dense code)</td>
<td>35.3 s</td>
<td>770 s</td>
<td>-</td>
</tr>
<tr>
<td>Time (Adaptive code)</td>
<td>21.3 s</td>
<td>107 s</td>
<td>808 s</td>
</tr>
</tbody>
</table>

**Table**: Scalability in test case size
Conclusions

- Save memory and computation time on sparse test cases.
- Obiwan 4D code can perform realistic simulations.
- Postprocessing tool:
  - Out-of-core visualization of 2D slices from 4D data.
- Perspective 1: 2D advections?
- Perspective 2: MPI version?