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

Outline

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

- This course focus 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*
  - **S**ignal-to-**N**oise **R**atio
- Trade-off
  - image quality
  - compression rates
  - computational complexity

- Video encoding with wavelets
  - Image parts with low energy $\rightarrow$ should use few bits
  - Try to avoid adding noise
  - Coupled to *lossless* encoding
- Time dimension
  - wavelet analysis: 3D

    or
  - 2D wavelet transform + motion estimation

- PDEs
  - help describing physical phenomena, help modeling
  - hard to solve analytically $\longrightarrow$ 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
    $\longrightarrow$ increase local resolution
  - smooth areas
    $\longrightarrow$ decimate/remove grid points
  - accuracy control, numerical schemes, ... (mathematics)
  - efficient sparse data structure, algorithms, parallelization (computer science)

- 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

    $\longrightarrow$ algorithms choice, sparsity management
  - compact representation in memory
  - multiresolution scheme: have access to one peculiar level

irfm cea cadarache

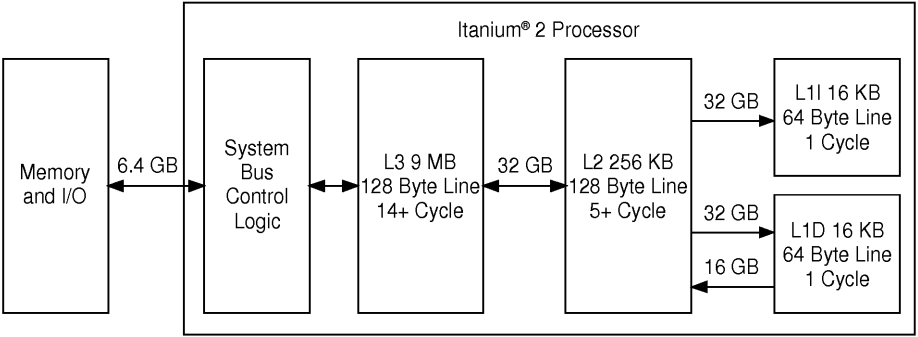
- Architectural trends:
  $\longrightarrow$ cost of accessing main memory
- Wide gap
  $\longrightarrow$ available perf. and achieved perf. of software
  $\longrightarrow$ estimation of the gap
- What should we do ?
  - reorganize data struct. to improve cache locality
  - *clustering*, compression of data in memory
  - *cache-conscious* data structures
- Extensively compare against *dense* applications
  - optimization and high-performance computations

- Programmer's problem
  1. design/choose the right algorithm
  2. proove correctness of the algorihm
  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?

Outline

- 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 $\longrightarrow$ 5 MFLOPS
  - achieves small fraction of the peak processor performance
- This major problem is known as
  memory wall or processor-memory bottleneck



Reference: Intel Itanium 2 Processor Reference Manual for Software Development and Optimization

irfm cea cadarache

- When the processor needs to read/write main memory
  $\rightarrow$ checks whether a copy of that data lies in the cache
  $\rightarrow$ if not, copy a *block* of data into the cache
  $\rightarrow$ 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
    $\rightarrow$ *Cache-aware* algorithm
  - be aware of the constraint of main memory bandwidth
  - dense calculations often easier to optimize that 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
    $\rightarrow$ guaranty that data frequently accessed are in a cache
  - data alignment can impact cache performance

- Reminder
  - a C variable has a unique *memory address*
  - pointer: a variable that stores a *memory address*
  - accessing a variable's value by a pointer is called indirection

- *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 ☹

irfm cea cadarache

- 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* $\longrightarrow$ 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 $O(1) < O(N)$
   - insert/delete time increased $O(N) > O(1)$
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 $\longrightarrow$ 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)

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

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*
  - 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 ∞

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

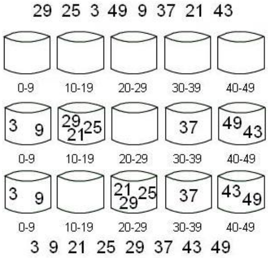
$$\forall n \geq n_0, \quad 0 \leq f(n) \leq c \, . \, g(n)$$

  - $\Theta(g(n))$: 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: $n^2 + n = O(n^2)$ 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 $O(n)$.

- 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

- 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
    $\longrightarrow$ C language
  - column-major order: stored column-wise in memory
    $\longrightarrow$ Fortran

```
|----+----+----+----|
|  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 with row major order

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

(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)

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

Figure: Basic algorithm

Code stored in matmul directory

Multidimensional array

Matrix-multiply example (2)

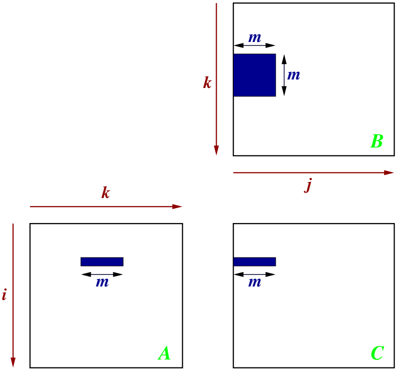
```
void mat_mul_dgemm(double*A, double*tB,
                   double*tC, int N) {
  double alpha = 1.;
  double beta  = 0.;
#ifdef MKL
  char *notransp  = "N";
  char *transpos  = "T"; /* transpose */
  dgemm(transpos, notransp, &N, &N, &N,
        &alpha, A, &N, tB, &N, &beta,
        tC, &N);
#else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

Figure: BLAS library call

```
  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



- Worst case (m the common size of blocking factors)
  - $MT_{basic}(N) = 2\,N^3 + N^2$
  - $MT_{blocked}(N) = 2\,N^3/m + N^2$

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

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

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* 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

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

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

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

- 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

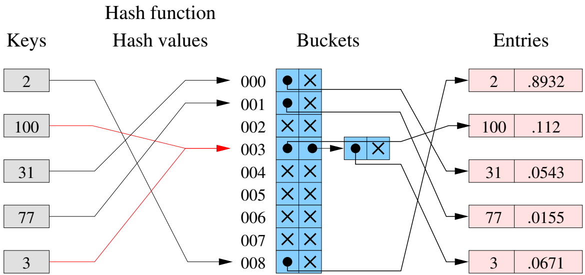


Figure: A chained hash table with five pairs (key-value) and one collision located at bucket (id 3).

Hash table

Some code

```
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);
    max = (fabs(*pval)>max?fabs(*pval):max);
  }
}
```

irfm cea cadarache

- 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 occured
    - reduce collisions
    - overhead: copy of the table

- 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

- 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)
    $\rightarrow$ space complexity & algo. complexity $O(S)$
  - one based on simple static array
    $\rightarrow$ entry of the array: (real value) or (NOTAVALUE constant)
    $\rightarrow$ space complexity & algo. complexity $O(N)$

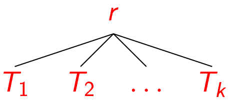
- Run-time for three traversals $N = 8.10^7$
  1. Iterate over the S records of the hash table, using a `glib` iterator (complexity in time $O(S)$).
  2. Traverse the whole dense array and select only defined values (complexity in time $O(N)$).
  3. Traverse the whole dense array and make a lookup to the hash table at each position (complexity in time $O(N)$).

- Timings on Nehalem 8-core node
  - run-time 2 $\approx$ run-time 1

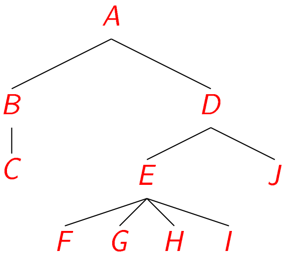
    iterator of the hash-table $\approx (\alpha = 15) \times$ array access time
  - run-time 3 $\approx 20 \times$ run-time 2

    lookup to the hash-table $\approx 20 \times$ array access time

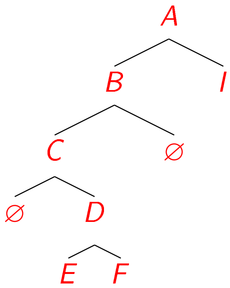
- Constitution
  - set of *nodes* connected with edges
    $\rightarrow$ graph with no cycle
  - *internal node*: has a set of children nodes
  - *leaf node*: no child, one parent node
  - *root*: common ancestor node
- Notation
  - *siblings*: children of one node
  - *degree* of a node: number of children
  - leaves have *height* 0
  - root has *depth* 0
- Root r and its sub-trees $T_1, T_2, \ldots, T_k$



Example & binary tree (1)



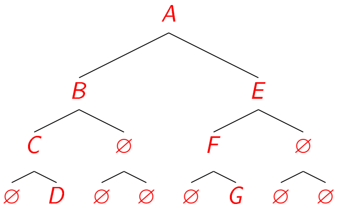
(a) General tree



(b) Binary tree

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 implicitly stored as an array (*Ahnentafel* list)
- a node is stored at index k,
  left child at $2k+1$,
  right child at $2k+2$
- *eliminate* memory *overhead* due to pointers




```
|---|---+---|---+---+---+---|---+---+---+---+---+---+---+---|
| A | B | E | C |   | F |   |   | D |   |   |   | G |   |   |
|---|---+---|---+---+---+---|---+---+---+---+---+---+---+---|
```

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

```
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

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

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

- Benefits: Fortran, contiguous, logical links, copy cost ...

- 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.)

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

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) ?

- 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)

```
|-------------+---------------------------------------|
|  Inst. Id.  |             Pipeline Stage            |
|-------------+----+----+----+----+----+----+----+----|
|      1      | FI | DI | EI | WB |    |    |    |    |
|-------------+----+----+----+----+----+----+----+----|
|      2      |    | FI | DI | EI | WB |    |    |    |
|-------------+----+----+----+----+----+----+----+----|
|      3      |    |    | FI | DI | EI | WB |    |    |
|-------------+----+----+----+----+----+----+----+----|
|      4      |    |    |    | FI | DI | EI | WB |    |
|-------------+----+----+----+----+----+----+----+----|
|      5      |    |    |    |    | FI | DI | EI | WB |
|-------------+----+----+----+----+----+----+----+----|
| Clock cycle |  1 |  2 |  3 |  4 |  5 |  6 |  7 | 8  |
|-------------+----+----+----+----+----+----+----+----|
```

  - Nehalem → 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
    $\rightarrow$ reduced number of indirection levels
  - compression: implicit binary tree
  - pointer elimination: significance map

- 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 → 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) → bit stream
  3. *Entropy coding* (*lossless* compression of the bit stream)

Wavelet tree in image processing

EZW - *Bit-plane* encoder

- 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 the m entries
  - $\ldots$
  - Last, encode least significant bit $n-1$ of the entries
- Example on a grayscale (8-bit) image (© CAES CNRS)

Original Clythia

Most Significant Bit

Bit (6)

Bit (4)

Bit (2)

Bit (7)

Bit (5)

Bit (3)

Least Significant Bit

- 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

```
SubList ← ∅; /* Subordinate List                                  */
for j ← 7 to 0 by -1 do
    /* Update Significance Map during Significance Pass         */
    SigMap ← coeff. of W such as w_{x,y} < 2^j and (x, y) ∉ SubList;
    Output in B: SigMap with a tree encoder;
    /* Refinement Pass                                          */
    SubList ← SigMap ∪ SubList;
    Output in B: bit-plane j for all w_{x,y} ∈ 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).

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

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

- 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 dictionnary: list of *(coeff., bit sequence)*
  2. loop on input coeff., output their bit sequence
- Examples of entropy coders
  - Huffman, Arithmetic, Shannon–Fano

Outline

- Sampled input function

$$c^n = \{c^n_k \mid 0 \leq k < 2^n\}\,.$$

c^4_0	c^4_1	c^4_2	c^4_3	c^4_4	c^4_5	c^4_6	c^4_7	c^4_8	c^4_9	c^4_{10}	c^4_{11}	c^4_{12}	c^4_{13}	c^4_{14}	c^4_{15}

Input signal $(N = 2^4)$

- One Haar representation: coeff. c^0_0 and $(d^j_{k\in[0,2^j-1]})_{j=0,n-1}$ defined recursively by difference and average operators

$$d^{n-1}_k = c^n_{2k+1} - c^n_{2k}\,,$$
$$c^{n-1}_k = c^n_{2k} + \frac{d^n_{2k+1}}{2}\,.$$

c^0_0	d^0_0	d^1_0	d^1_1	d^2_0	d^2_1	d^2_2	d^2_3	d^3_0	d^3_1	d^3_2	d^3_3	d^3_4	d^3_5	d^3_6	d^3_7

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

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

Mallat 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

d_k^j has the vector index $(1+2.k)\,2^{j_{max}-j}$ in the *in-place* version

Haar

Forward Wavelet transform

```
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];
    }
  }
}
```

(a) In-place storage

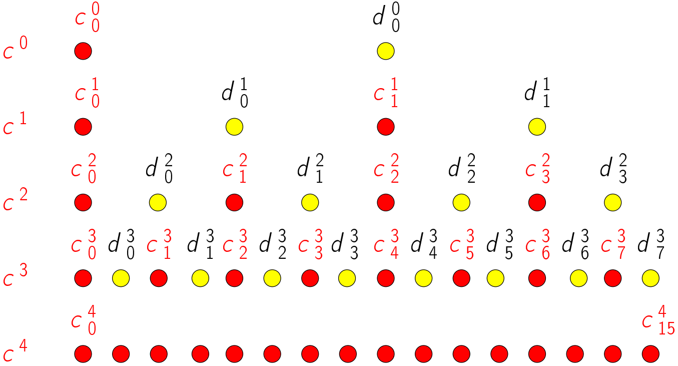
```
void haar_fdwt(double* vec, double *ts, int N) {
  register int half, k;
  register int idetail, iaverage;
  /* loop from finest level to coarsest level */
  for (half = N/2; half >= 1; half /= 2) {
    /* Copy input 'vec' to tempory 'ts' */
    for (k = 0; k < 2*half; k++)
      ts[k] = vec[k];

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

(b) Mallat representation

At level j, we have $itf = 2^{nblevel-j}$, $itc = 2\,itf$, $half = 2^j$,

2^j details are computed at each level $j \in [0, nblevel-1]$.



Dyadic grid and localization of c_*^* in red and d_*^* in yellow

Haar

Inverse wavelet transform

```
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) In-place storage

```
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
         store and at 'iaverage' is the average
         will be store */
      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 tempory 'ts' to input 'vec' */
    for (i=0; i < 2*half; i++) vec[i] = ts[i];
  }
}
```

(d) Mallat representation

Code stored in haar_mallat, haar_inplace directories

Haar

Forward/Inverse wavelet transform

```
void haar_predict(double* vec, int itc,
                  int N, int dir) {
  int i, itf = itc / 2;
  for (i = 0; i < N; i += itc)
    vec[i+itf] -= dir * vec[i];
```

```
void haar_ftransform(double *ts, int N) {
  int itc;
  for (itc = 2; itc <= N; itc *= 2) {
    haar_predict( ts, itc, N, 1 );
    haar_update( ts, itc, N, 1 );
  }
}
```

```
void haar_itransform(double *ts, int N) {
  int itc;
  for (itc = N; itc >= 2; itc /= 2) {
    haar_update( ts, itc, N, -1);
    haar_predict( ts, itc, N, -1);
  }
}
```

(e) In-place storage

```
void haar_ftransform(double *ts,
                     double *tmpvec, int N) {
  int size;
  for (size = N; size > 1; size /= 2) {
    haar_split(ts, tmpvec, size);// rearrange ts
    haar_predict(ts, size, 1);
    haar_update(ts, size, 1);
  }
}
```

```
void haar_itransform(double *ts,
                     double *tmpvec, int N) {
  int size;
  for (size = 2; size <= N; size *= 2) {
    haar_update(ts, size, -1);
    haar_predict(ts, size, -1);
    haar_merge(ts, tmpvec, size);// rearrange ts
  }
}
```

(f) Mallat representation

Exercise: Much 'simpler' DWT are shown here compared to previous slides.

But are these functions *cache-aware* compared to previous ones.

Evaluate $MT(N)$ for the different forward transforms.

Haar

Thresholding (Mallat representation)

```c
/* 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);
}
```

Outline

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 ← THR(DWT(In));
for n = 1 to n ≤ nb_steps do
    Translate signal W_wold and compute a new adaptive grid G_new;
    Build well structured tree-grid G_new ← TREE(G_new);
    Compute signal F_new on grid G_new (interpolating on F_old);
    Adaptive wavelet transform W_new ← ADWT(F_new);
    Thresholding W_new ← THR(W_new);
    Swap W_new and W_old;
    Swap F_new and F_old;
```

Traveling signal

Macros & accessors

```
#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;
```

Traveling signal

Code samples (1)

```
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);
}
```

Traveling signal

Code samples (2)

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

typedef struct shashtable hashtable;
```

```
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);
  }
}
```

```
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"\
             "i %d itc %d\n",(int)i,(int)itc);
      exit(2);
    }
    if (detailVal != NOTAVALUE) {
      _GETPOINTER(sdata,idetail,pdetail);
      *pdetail -= dir * predictVal;
    }}}
```

Traveling signal

Build well structured wavelet tree

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

int inline wav_itf(int input, int nblevel) {
  int j = 0x1<<nblevel, 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,nblevel);
      if ((myitf == itf) && (valdet != NOTAVALUE)) {
#else
    for (i = itf; i < N; i+= itc) {
      _GETF(snew,i,valdet);
      if (valdet != NOTAVALUE) {
#endif
        icoa = i-itf;
        _GETF(snew,icoa,val);
        if (val == NOTAVALUE) {
        _SETF_NEW(snew,icoa,0.);
        }
      }
    }
  }
```

Traveling signal

Dynamic remeshing in time: advection+refinement

```
  /* scan all levels */
  for (itf = 1, itc = 2, ilevel = 0; itc <= N;
       itf *=2, itc*=2, ilevel++) {
    atf = itf/2; if (atf < 1) atf = 1;
    /* higher bits mask corrresponding to atf */
    wav_getmask(atf, &maskatf);
    /* traversal at level 'itc' */
#ifdef _HASH
    pstart = pend;
    pend   = swav->ends[ilevel];
    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);
#endif
      if (val != NOTAVALUE) {
        /* advect grid point */
        intdisp    = (i + floordisp + N)%N;
        /* get a 'divide atf' grid point */
        intdisp   &= maskatf;
        /* SAME AS: intdisp /= atf; intdisp *= atf; */
        /* refinement procedure, insert
           patchsize points for each initial point */
        beginpatch = N+intdisp-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

OBIWAN : Vlasov solver using a Wavelet based Adaptive Mesh Refinement

Matthieu Haefele, Guillaume Latu
Michael Gutnic, Eric SonnenDrücker

CALVI project

Cadre, modélisation

Vlasov equation

$$\frac{\partial f}{\partial t} + \vec{v}.\nabla_x f + (E + \vec{v} \times B).\nabla_v f = 0$$

- $f(\vec{x}, \vec{v}, t)$: particle distribution function at time t in phase space, $(\vec{x}, \vec{v}) \in \Re^d \times \Re^d$ 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

$$\frac{\partial f}{\partial t} + v.\frac{\partial f}{\partial x} + (E_{self}(x,t) + E_{app}(x,t)).\frac{\partial f}{\partial v} = 0$$

- Non linear PDE

Cadre, modélisation

Splitting de l'équation

- Pour décrire une évolution en temps,
  on **peut** résoudre l'équation en **deux** temps
  - Splitting en V (*x* constant)

$$\frac{\partial f}{\partial t} + E_{applied\ +self}(x,t)\frac{\partial f}{\partial v} = 0$$

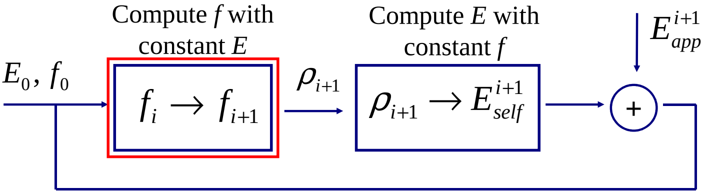
  - Splitting en X (*v* constant)

$$\frac{\partial f}{\partial t} + v\frac{\partial f}{\partial x} = 0$$

- On **peut** aussi résoudre l'équation **sans** splitting

Cadre, modélisation

Global algorithm



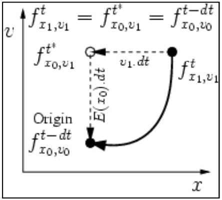
- Electrical field solved with Poisson equation

$$\frac{dE}{dx} = \rho(x,t) = \int f(x,v,t)dv$$

Cadre, modélisation

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

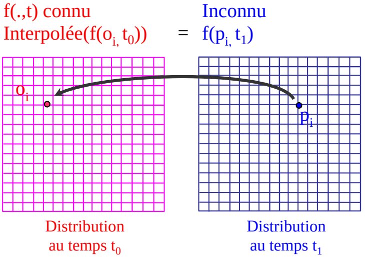
- Two 1D advections (splitting method)

- Two major kind of numerical schemes
  - **P**article-**I**n-**C**ell 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



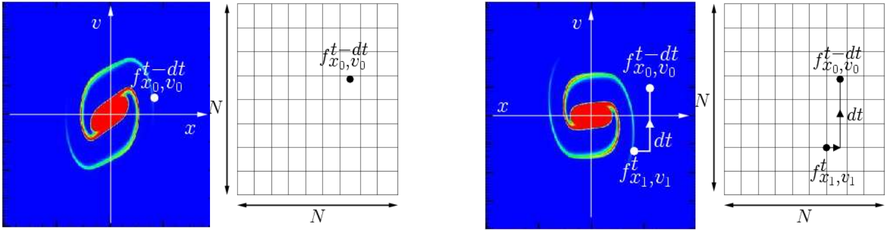
  - Version sans splitting -> interpolation 2D

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.

Semi-Lagrangian method



Time *t-dt* Time *t*

- Appearance of fine structures ➡ fine grids needed

Idea: use a multiresolution analysis to adapt the numerical method

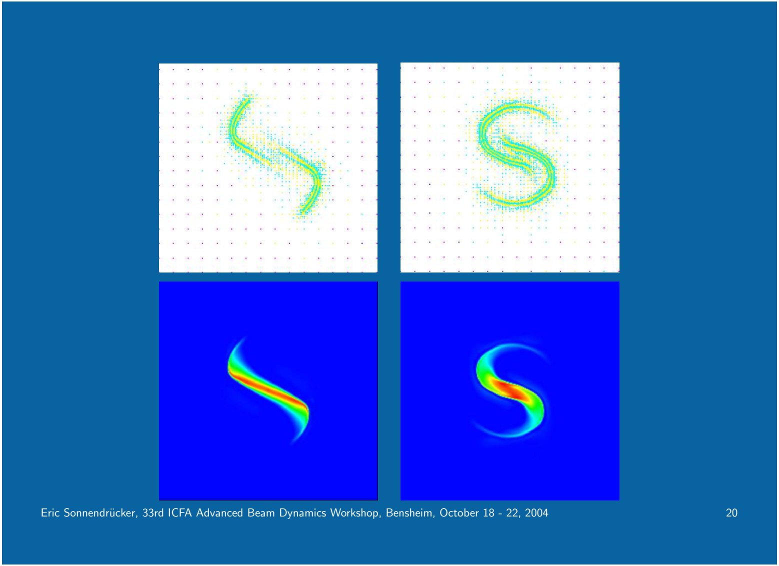
irfm
cea
cadarache

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

Eric Sonnendrücker, 33rd ICFA Advanced Beam Dynamics Workshop, Bensheim, October 18 - 22, 2004 19

Application 1: Vlasov 2D



Application 1: Vlasov 2D

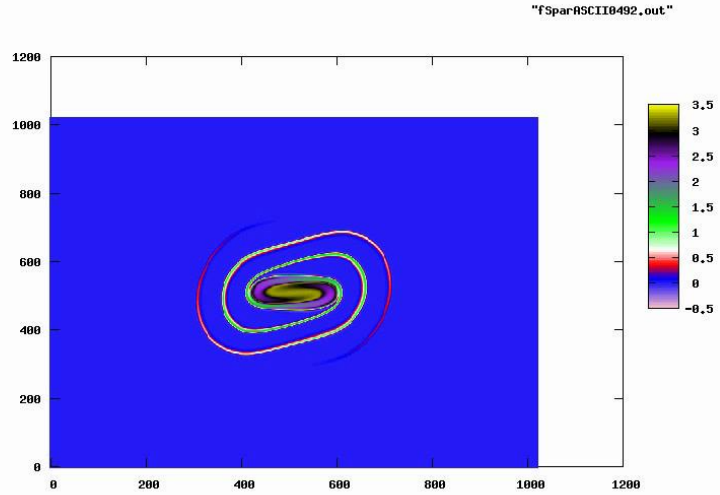


Schéma adaptatif

MultiResolution Analysis (MRA)

- For a grid G_j
  - $c_k^j = f(x_k^j)$
  - Projection operator: $c_k^j = c_{2k}^{j+1}$
  - Prediction operator: $c_{2k+1}^{j+1} = P_{2N+1}(x_{2k+1}^{j+1})$
    with P the Lagrange polynomial
  - The detail is defined as $\boxed{d_k^j = c_{2k+1}^{j+1} - P(x_{2k+1}^{j+1})}$
- We have defined a MRA!

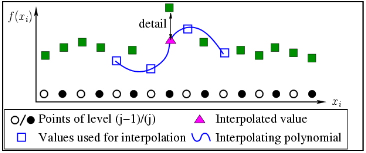
$$f(x) = \sum_{k=0}^{N} c_k^0 \varphi_k^0(x) + \sum_{j=1}^{nblevel} \sum_{k=0}^{N} d_k^j \psi_k^j(x)$$

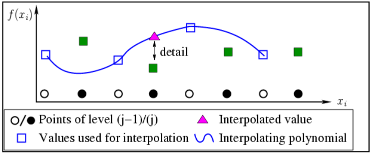
d is small $(d < \varepsilon)$ where f is regular

Schéma adaptatif

MultiResolution Analysis (MRA)

- Wavelet transform algorithm





- Need a well formed tree of wavelet coefficients

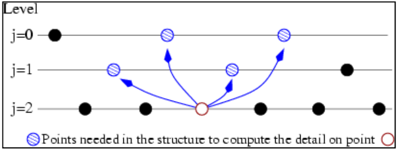
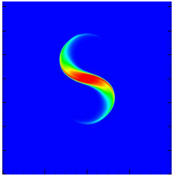


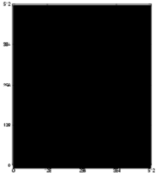
Schéma adaptatif

MultiResolution Analysis (MRA)

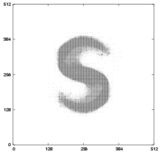
- Error control



Dense representation



F_{dense}



F_{adapt}

Wavelet representation

$$\boxed{\left\| F_{dense} - F_{adapt} \right\|_{L_1, L_2, L_\infty} < \varepsilon}$$

Schéma adaptatif

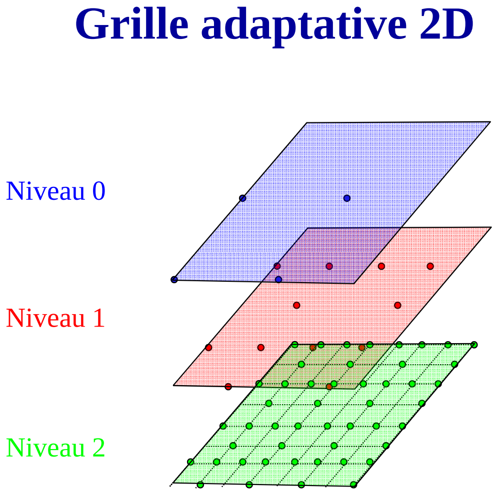
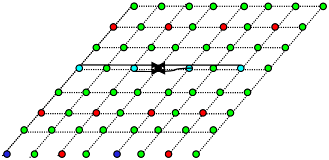


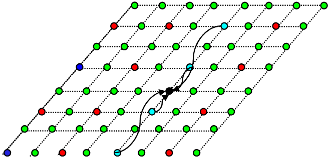
Schéma adaptatif

Schéma 2D

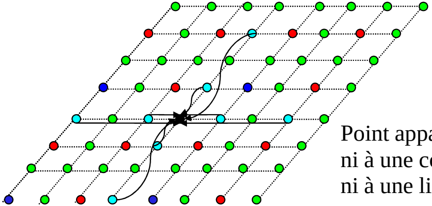
produit tensoriel de transformations 1D



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 $\#F$)

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

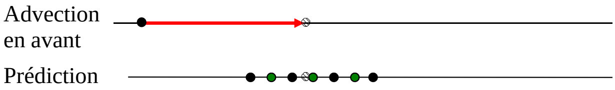
    (raffinement du maillage)

    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*



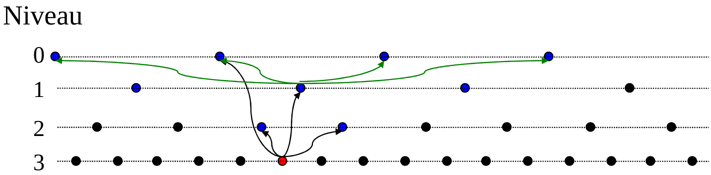
Points ajoutés au même niveau que le point de départ (●)

Points ajoutés au niveau plus fin pour raffiner le maillage(●)

Algorithme adaptatif

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

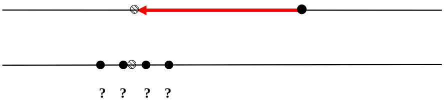
Algorithme adaptatif

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

Advection en arrière

Interpolation



- 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
    Reconstruire *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** $j=max$**-**1
  - Interpoler 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

Algorithme 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

Etude algorithmique

Complexités algorithmiques (1)

1) Prédiction et 2) Raffinement

$D_t \to G$

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

3) Construction d'une structure arborescente

$G \to G$

- 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 $\#G$

Etude algorithmique

Complexités algorithmiques (2)

Advection arrière

Point advecté (nœud de *G*)

Interpolation avec points sur la grille fine

- **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)

$(G,D_t)\rightarrow F_{t+dt}$

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 * (taille du filtre: 4 ou +)

$(G,F_{t+dt}) \rightarrow D_{t+dt}$

- **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
    → aspect séquentiel qui nécessitera des synchronisations des algorithmes parallèles

Optimisation

Optimisations mises en œuvre

- α) Parties coûteuses $\rightarrow$ 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 $\rightarrow$ [4-calcul de la densité]

    7 % du temps $\rightarrow$ calcul du champ

    normal : complexités linéaires en (taille grille fine)
  - grille fine de grande taille (2048x2048), nombreux détails dans D_t

    45 % du temps $\rightarrow$ [4-calcul de la densité]

    4 % du temps $\rightarrow$ calcul du champ

    51 % du temps $\rightarrow$ 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
    $\rightarrow$ utilisation de macros
  - favoriser les lectures/écritures de suites d'octets contigus en mémoire (localité spatiale) $\rightarrow$ 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

irfm cea cadarache

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

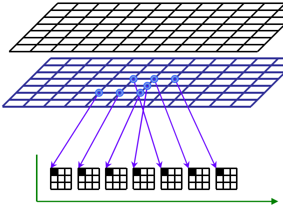


Tableau de valeurs
(niveaux 0 à L)

Tableau de pointeurs

Tableaux de valeurs (cellules)
(niveaux L+1 à *nblevel*)

- Avantages et inconvénient :
  - 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 $\rightarrow$ refonte,

- Modification de certains algo. pour version //
  - [4-calcul de densité] est réécrit : algo. par bloc
    $\rightarrow$ surcoût dû à des calculs redondants
    $\rightarrow$ 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

Optimisation

Optimisation β

- Performances sur un pas de temps (janvier 2005) : le temps de chaque partie a été réduite, 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

Optimisation

Conclusion Optimisations α, β

- Amélioration notable sur un pas de temps
  grille fine de grande taille, beaucoup de détails
  obiwan 13s $\rightarrow$ 2.8s obitwo

Schéma adaptatif

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 $(E_{t-dt}, D_{t-dt} \rightarrow G_t)$
- Make a well formed tree $(G_t \rightarrow G_t)$

1B Compute the distrib. function F_t

- Inverse Wavelet Transform $(D_{t-dt}, G_t \rightarrow F_{t-dt})$
- Backward advection on A_t $(E_{t-dt}, F_{t-dt}, G_t \rightarrow F_t)$

1C Adaptive Wavelet Transform and compress $(F_t \rightarrow D_t)$

2 Splitting in x-direction (idem)

3 Compute electrical field $(D_t \rightarrow E_t)$

Algorithmes et structure de données

Algorithmic complexities

- *Hypothesis*: At time step t, each adaptive struct. holds $\sim S$ coefficients, with $S < N^2$
- Algorithmic complexities (reads, writes, operations)

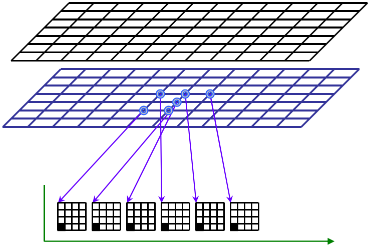
Steps	#reads	#operations	#writes
Build the adaptive grid (1A or 2A)	$O(S)$	$O(S)$	$O(S)$
Compute distrib. function (1B or 2B)	$O(S)$	$O(S)$	$O(S)$
Compute wavelet transform (1C or 2C)	$O(S)$	$O(S)$	$O(S)$
Compute electrical field (3)	$O(N+S)$	$O(N+S)$	$O(N)$

- Each part of a time step should be done in parallel

Algorithmes et structure de données

Data structures

- Constraints : a) Traversal of large adaptive structure
  b) 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*)

Algorithmes et structure de données

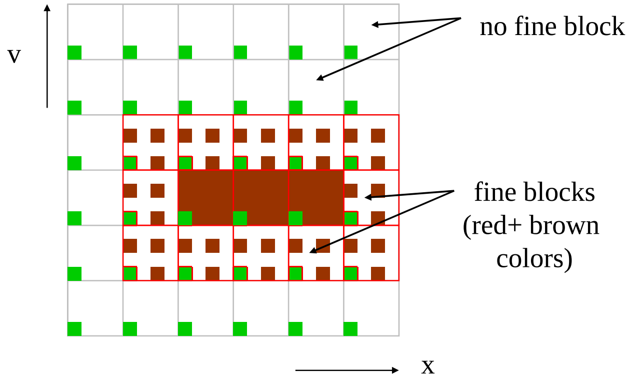
Data structures

- Management of sparsity → not optimal
- Low cost to read one element in sparse structure
  → 1 memory access for levels *0* → *L*
  → 2 memory accesses for levels *L+1* → *nblevel*
- Spatial and temporal locality improved
  significant reduction of exec. time (vs. hash tables)

Parallélisation

Data partitioning

- Example of sparse data to distribute:



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

Parallélisation

Target paradigm & machine

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

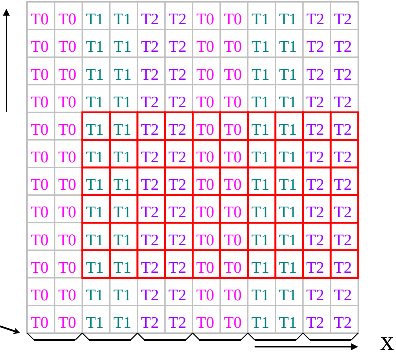
- OpenMP chosen (one thread per processor)

- 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

B=6 subdomains
mapped
onto threads



3 threads:
T0
T1
T2

Parallélisation

Optimization of subdomain size

- Subdomains lead to different computation costs
  $\rightarrow$ implies load imbalance
  $\rightarrow$ try to increase the number B of subdomains

- Most of our algorithms benefit from spatial locality
  $\rightarrow$ numerous subdomains increase
  the number of distant memory accesses
  $\rightarrow$ memory bandwidth limitation
  $\rightarrow$ 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

irfm cea cadarache

Parallélisation

Performance analysis

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

	Number of procs.	1	16	32	64
	Steps	time (sec.)	speedup	speedup	speedup
Splitting V	1A Prediction	0.7659	14.6	24	45
Splitting V	1A Well formed tree	1.4023	15.4	30	49
Splitting V	1B Compute distrib. function	5.1605	14.8	28	54
Splitting V	1C Wavelet transform	0.6782	17.3	33	62
Splitting X	2A Prediction	1.1447	14.6	25	47
Splitting X	2A Well formed tree	2.4335	15.3	28	52
Splitting X	2B Compute distrib. function	4.9924	13.3	23	33
Splitting X	2C Wavelet transform	0.8598	14.3	23	34
	3 Compute Field	0.2523	15.5	24	45

Parallélisation

Performance analysis

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

Number of procs.	**1**	**16**	**32**	**64**
Machines	time (sec.)	speedup	speedup	speedup
SGI Origin 3800	15539	14,2	25,1	42,2
IBM SP3 NH2	19644	14,6	-	-

- Main problems:
  - Load balancing
  - Output writing time (on hard drive)

irfm
cea
cadarache

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 x 2^K]
[précision 10^-5 en adaptatif]

K	10	11	12	13	14
Taille maillage (MO)	8	32	128	512	2048
LOSS2D (sec.)	0.11	0.44	2.70	24.20	138.60
Obiwan2D (sec.)	0.33	0.83	2.46	3.70	8.90

- Avantage indéniable à l'adaptatif lorsque K est élevé.

Parametric Instability (2)

time $= 86.778\omega_p^{-1}$

time $= 104.130\omega_p^{-1}$

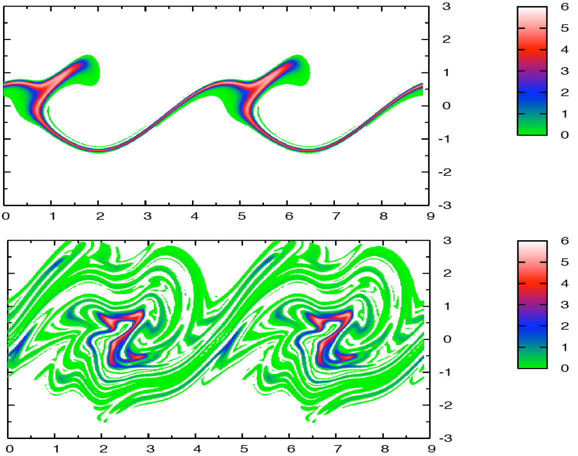
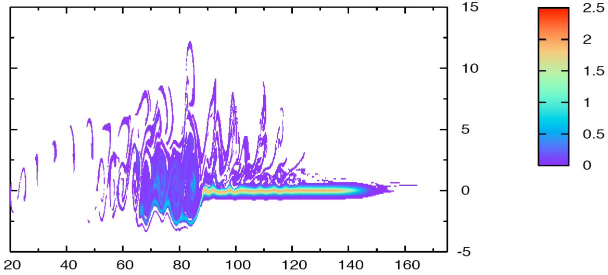


Figure: Evolution of the distribution function in the phase space (x, p_x) during the saturation phase for the parametric instability.

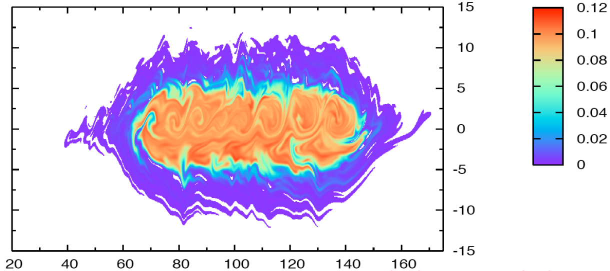
Vertical: p_x-axis, Horizontal: x-axis

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 $= 1230.4\omega_p^{-1}$

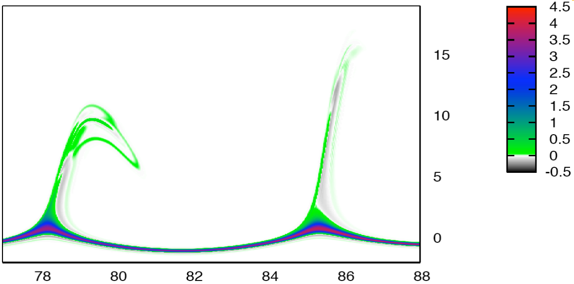


$P_{osc} = 1.25,\ T_e = 100kev,\ n_0/n_c = 1.20,$ Mesh: $2^{8+3}(x) \times 2^{6+3}(p_x)$

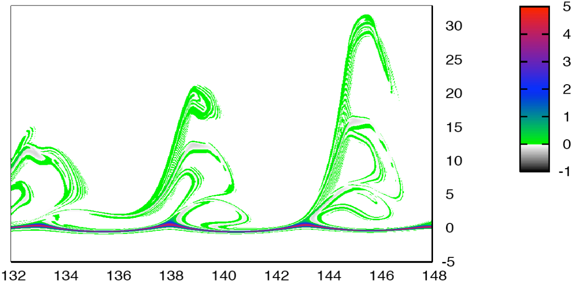
Application 1: Vlasov 2D (with N. Besse)

Laser wake

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



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



$P_{osc} = \sqrt{3/2},\ T_e = 3kev,\ n_0/n_c = 0.1$, Mesh: $2^{10+3} \times 2^{8+3}$

Conclusion

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

- In the sequel we shall consider only the collision-less Vlasov-Maxwell equations

$$\begin{aligned}\partial_t f + \mathbf{v} \cdot \nabla_x f + \mathcal{F} \cdot \nabla_{\mathbf{v}} f &= 0 \\ \mathcal{F} &= \mathbf{E} + \mathbf{v} \times \mathbf{B}\end{aligned}$$

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

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

- 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
  $\longrightarrow$ 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

irfm cea cadarache

- The Vlasov equation (2D space, 2D velocity, d=4) is split. We solve it, using elementary 1D advection equations:

$$
\begin{aligned}
\partial_t f + v_x \partial_x f &= 0 & (\hat{x} \ \text{operator})\\
\partial_t f + v_y \partial_y f &= 0 & (\hat{y} \ \text{operator})\\
\partial_t f + \dot{v_x} \partial_{v_x} f &= 0 & (\hat{v_x} \ \text{operator})\\
\partial_t f + \dot{v_y} \partial_{v_y} f &= 0 & (\hat{v_y} \ \text{operator})
\end{aligned}
$$

- One time step of simulation is composed of:
  - successive splittings in each dimension (advection steps)
  - field solving using the source term $\rho = \int f \, d\mathbf{v}$
- Main cost of the application: interpolations

irfm cea cadarache

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

$$f(z) \;=\; \sum_{\boldsymbol{k}} c_{\boldsymbol{k}}^{j_0} \varphi_{\boldsymbol{k}}^{j_0} + \sum_{j=coarse\ \ level}^{fine\ \ level} \sum_{\boldsymbol{k}} d_{\boldsymbol{k}}^{j} \psi_{\boldsymbol{k}}^{j}$$

- Coefficients $c_{\boldsymbol{k}}^{j_0} = f(z_{\boldsymbol{k}}^{j_0})$
- Detail coefficients
  $d_{\boldsymbol{k}}^{j} = f(\boldsymbol{z}_{\boldsymbol{k}}^{j}) - \text{Lagrange interp. in } \boldsymbol{z}_{\boldsymbol{k}}^{j} \text{ at level } j-1$
  Details d are small if f is locally smooth
- Only grid points where f varies most are kept, others are eliminated
  - Usually:
    Nb. of c and d coeff. $\ll$ Nb. of points in uniform grid N^4

irfm cea cadarache

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

Input : $F^{t+\epsilon}$ (distribution function known on the adaptive grid)
Output: $D^{t+\epsilon}$ (wavelet coefficients, details)

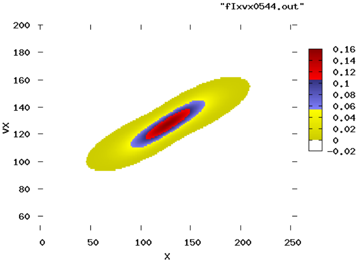
$D^{t+\epsilon} \leftarrow F^{t+\epsilon}$; iF $\leftarrow$ 1; iC $\leftarrow$ 2;
for $j \leftarrow nblev$ **to** *1* **by** *-1* **do**
  for $d \leftarrow 1$ **to** *4* **do**
    $s_{[1:4]} \leftarrow 0$; $t_{[1:4]} \leftarrow iF$; $e_{[1:4]} \leftarrow 2^{j0+nblev} - 1$; $s_d \leftarrow iF$;
    $t_d \leftarrow iC$;
    for $i_0 \leftarrow s_0$ **to** e_0 **by** t_0 **do in parallel**
      for $i_1 \leftarrow s_1$ **to** e_1 **by** t_1 **do in parallel**
        for $i_2 \leftarrow s_2$ **to** e_2 **by** t_2 **do in parallel**
          for $i_3 \leftarrow s_3$ **to** e_3 **by** t_3 **do in parallel**
            if $w_{i_*}^{t+\epsilon} \in D^{t+\epsilon}$ **then**
              $m_{[1:4]} \leftarrow i_{[1:4]}$; $\tau \leftarrow 0$;
              for $z \leftarrow l_1$ **to** l_2 **do**
                $m_d \leftarrow i_d + z\, iC - iF$;
                $\tau \leftarrow \tau - h(z)\, w_{m_*}^{t+\epsilon}$;
              $w_{i_*}^{t+\epsilon} \leftarrow w_{i_*}^{t+\epsilon} + \tau$
  iF $\leftarrow$ 2 iF; iC $\leftarrow$ 2 iC;

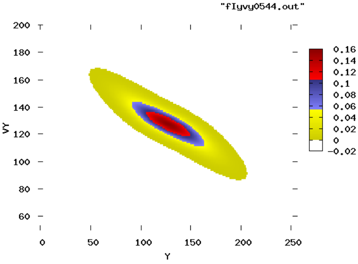
Main problems

- 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 $\longrightarrow$ random accesses must be quick
- Evaluate different load balancing strategies

Test case (Beam): Alternate Gradient Focusing

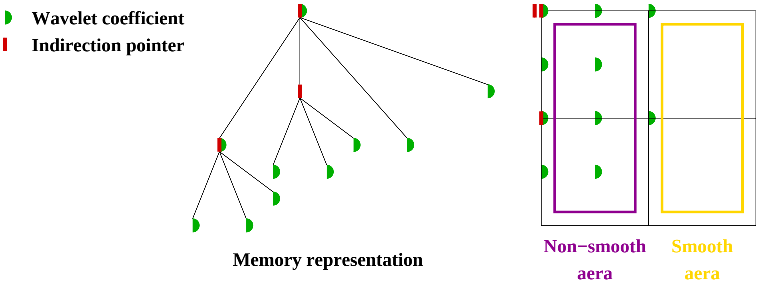
- 40 mA, 1 MeV potassium beam in alternating gradient lattice, $\Delta t = 0.000464s$
- Generates sparsity in 4D data structure
- Raises problem of managing work distribution





Adaptive data structure

- Analogy with binary tree, quad-tree used to partition 1D, 2D
  $\rightarrow$ 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:



- With such trees $\rightarrow$ 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

Substep	Number of // loops	computation grain
F_Prediction	**2**	2D slice of coarse pts and descendants
F_Maketree	3	1D slice of coarse pts and descendants
B_Advection	3	1D slice of coarse pts and descendants
Wavelet transfom	3	1D slice of coarse pts and descendants
Field computation	3 (partly seq.)	1D slice of coarse pts and descendants
Diagnostics	almost sequential	

Table: Loops parallelization in each substep

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

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

Table: Computation time and speedup (indicated between parentheses) averaged on 3 iterations - 128^4 test case, IBM Power5 16-way node

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

Nb. procs.	**1**	**4**	**16**
Parts in one time step			
F_Prediction	13.687 (1.0)	4.016 (3.4)	2.446 (5.6)
F_Maketree	19.395 (1.0)	4.918 (3.9)	1.904 (10.2)
B_Advection	201.101 (1.0)	50.120 (4.0)	13.773 (14.6)
Wavelet transfom	31.797 (1.0)	7.841 (4.1)	2.342 (13.6)
Field computation	1.464 (1.0)	0.374 (3.9)	0.099 (14.9)
Diagnostics	2.088 (1.0)	0.923 (2.3)	0.688 (3.0)
Complete Iteration	269.5 (1.0)	68.2 (4.0)	21.3 (12.7)

Table: Computation time and speedup (indicated between parentheses) averaged on 3 iterations - 128^4 test case, IBM Power5 16-way node

irfm cea cadarache

- 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

Domain size	128^4	256^4	512^4
Total Memory (Dense code)	2 GB	32 GB	512 GB
Total Memory (Adaptive code)	0.45 GB	2.32 GB	14.1 GB
Time (Dense code)	35.3 s	770 s	-
Time (Adaptive code)	21.3 s	107 s	808 s

Table: Scalability in test case size

- 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 ?