Multi-level parallelism for high performance combinatorics

Florent Hivert

LRI / Université Paris Sud 11 / CNRS

SPLS / June 2018

Goal

Present some experiments, experience return, and challenges around parallel (algebraic) combinatorics computations.

What I learned:

Following the these optimization steps

- Micro data-structures optimization
- Work stealing parallelization
- Careful memory management

we can achieve surprisingly (at least for me) large speedups.

Some classical algebraic/combinatorics objects

Multivariate polynomials:

$$x_1^3 x_4 x_6 + 5\, x_2^3 x_5^4 x_8^2 - 12\, x_4^8$$

Number of monomials v variables, degree d :

$$M(v,d) = \binom{v+d-1}{v}$$

$$M(5,5) = 126, \qquad M(5,10) = 252$$

$$M(10,10) = 92378, \qquad M(10,20) = 2\cdot 10^7$$

$$M(16,16) = 300\,540\,195, \qquad M(16,32) = 1.5\cdot 10^{12}$$

Some classical algebraic/combinatorics objects

(Fully) Symmetric polynomials:

$$\begin{aligned} m_{(2,1)} =& x_0^2x_1 + x_0x_1^2 + x_0^2x_2 + x_1^2x_2 + x_0x_2^2 + x_1x_2^2 + x_0^2x_3 \\ &+ x_1^2x_3 + x_2^2x_3 + x_0x_3^2 + x_1x_3^2 + x_2x_3^2 \\ m_{(2,2,1)} =& x_0^2x_1^2x_2 + x_0^2x_1x_2^2 + x_0x_1^2x_2^2 + x_0^2x_1^2x_3 + x_0^2x_2^2x_3 + x_1^2x_2^2x_3 \\ &+ x_0^2x_1x_3^2 + x_0x_1^2x_3^2 + x_0^2x_2x_3^2 + x_1^2x_2x_3^2 + x_0x_2^2x_3^2 + x_1x_2^2x_3^2 \end{aligned}$$

Index: **integer partitions:**

$$(5), (4,1), (3,2), (3,1,1), (2,2,1), (2,1,1,1), (1,1,1,1,1)$$

n	1	2	4	8	10	16	20	50	100	256
$p(n)$	1	2	5	22	42	231	627	204226	$2 \cdot 10^8$	$3.7 \cdot 10^{14}$

Group algebra

Linear combination of **permutations:**

$$[1,2,3,4,5] + 2\,[1,2,3,5,4] + 3\,[1,2,4,3,5] + [5,1,2,3,4]$$

Product: composition of permutations.

The number of permutation grows very fast:

$$16! = 1\,307\,674\,368\,000 = 1.3\,10^{12}$$

Nested higher order directional derivative

Directional derivative, first and higher order:

$$\nabla_{\Xi_1} A \qquad\qquad \nabla^3_{(\Xi_1,\Xi_2,\Xi_3)} A = \nabla^3_{\Xi_1\otimes\Xi_2\otimes\Xi_3} A$$

Chain rule for directional derivative

$$\nabla_\xi \nabla^k_{\Xi_1\otimes\cdots\otimes\Xi_k} A = \nabla^{k+1}_{\xi\otimes\Xi_1\otimes\cdots\otimes\Xi_k} A + \sum_{j=1}^{k} \nabla^k_{\Xi_1\otimes\cdots\otimes\nabla_\xi\Xi_j\otimes\cdots\otimes\Xi_k} A$$

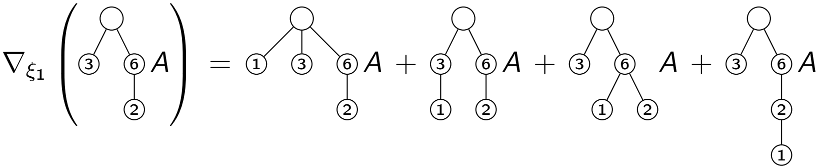
Nested higher order directional derivative

Directional derivative, first and higher order:

$$\nabla_{\Xi_1} A \qquad\qquad \nabla^3_{(\Xi_1,\Xi_2,\Xi_3)} A = \nabla^3_{\Xi_1\otimes\Xi_2\otimes\Xi_3} A$$

Chain rule for directional derivative

$$\nabla_\xi \nabla^k_{\Xi_1\otimes\cdots\otimes\Xi_k} A = \nabla^{k+1}_{\xi\otimes\Xi_1\otimes\cdots\otimes\Xi_k} A + \sum_{j=1}^{k} \nabla^k_{\Xi_1\otimes\cdots\otimes\nabla_\xi\Xi_j\otimes\cdots\otimes\Xi_k} A$$



Algebraic combinatorics: Summary

Note

- *Dealing with (formal) linear combinations of*
- *objects whose set cardinality grows exponentially fast;*

Corollary

- *sparse Linear algebra;*
- *small objects are usually sufficient !*

Algebraic combinatorics: Summary

Note

- *Dealing with (formal) linear combinations of*
- *objects whose set cardinality grows exponentially fast;*

Corollary

- *sparse Linear algebra;*
- *small objects are usually sufficient !*

Small combinatorial objects (i.e. monomials)

Very often, small combinatorial objects can be encoded into small sequences of small integers !

- Permutations:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 6 & 9 & 4 & 8 & 2 & 7 & 3 & 5 \end{pmatrix} = [1, 6, 9, 4, 8, 2, 7, 3, 6]$$

- Integer partitions: $10 = 5 + 2 + 2 + 1 = 4 + 3 + 1 + 1 + 1$
- Set partitions: $\{\{1, 4, 8\}, \{2, 3\}, \{5, 6, 7\}\}$
- Young tableaux: $\begin{array}{|c|c|c|c|c|} \hline 5 \\ \cline{1-1} 2 & 6 & 9 \\ \cline{1-3} 1 & 3 & 4 & 7 & 8 \\ \hline \end{array}$
- Dyck (well bracketed) word: 1101101001100011010

Integer Vector Instruction

Register: `epi8`,`epu8`: 128 bits = 16 bytes

Even more: AVX, AVX2, AVX512

- Arithmetic/logic operations: `and, or, add, sub, min, max, abs, cmp`
- Bit finding, scanning: `popcount`, `bfsd`

But more crucial for me:

- Array manipulation: `blend, broadcast, shuffle`
- String comparision: `cmpistr` (lex, find).

Very efficient manipulations !

Integer Vector Instruction

Register: `epi8`,`epu8`: 128 bits = 16 bytes

Even more: AVX, AVX2, AVX512

- Arithmetic/logic operations: `and`, `or`, `add`, `sub`, `min`, `max`, `abs`, `cmp`
- Bit finding, scanning: `popcount`, `bfsd`

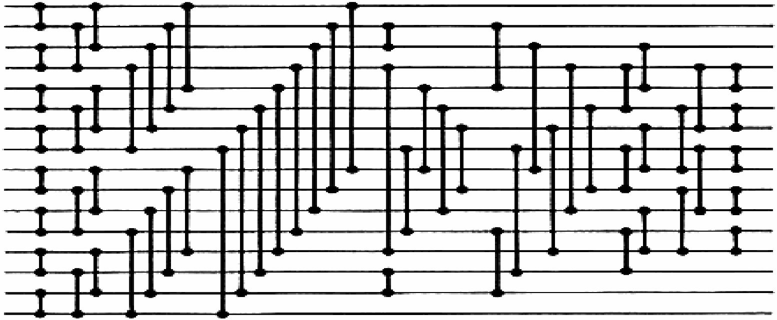
But more crucial for me:

- Array manipulation: `blend`, `broadcast`, `shuffle`
- String comparision: `cmpistr` (lex, find).

Very efficient manipulations !

Example: Sorting network

Knuth AoCP3 Fig. 51 p. 229:



$n = 16$ 61 modules, delay 9

```
// Sorting network Knuth AoCP3 Fig. 51 p 229.
static const array<Perm16, 9> rounds = {{
       { 1, 0, 3, 2, 5, 4, 7, 6, 9, 8,11,10,13,12,15,14},
       { 2, 3, 0, 1, 6, 7, 4, 5,10,11, 8, 9,14,15,12,13},
       ...
    }};

perm sort(perm a) {
  for (perm round : rounds) {
    perm minab, maxab, mask;
    perm b = _mm_shuffle_epi8(a, round);
    mask = _mm_cmplt_epi8(round, permid);
    minab = _mm_min_epi8(a, b);
    maxab = _mm_max_epi8(a, b);
    a = _mm_blendv_epi8(minab, maxab, mask);
  }
  return a;
}
```

```
// Sorting network Knuth AoCP3 Fig. 51 p 229.
static const array<Perm16, 9> rounds = {{
      { 1, 0, 3, 2, 5, 4, 7, 6, 9, 8,11,10,13,12,15,14},
      { 2, 3, 0, 1, 6, 7, 4, 5,10,11, 8, 9,14,15,12,13},
      ...
   }};

perm sort(perm a) {
  for (perm round : rounds) {
    perm minab, maxab, mask;
    perm b = _mm_shuffle_epi8(a, round);
    mask = _mm_cmplt_epi8(round, permid);
    minab = _mm_min_epi8(a, b);
    maxab = _mm_max_epi8(a, b);
    a = _mm_blendv_epi8(minab, maxab, mask);
  }
  return a;
}
```

Compared to `std::sort`**, speedup = 22.3**

Disjoint-set (Union-Find) of data-structure

SetPartition of $\{1, 2 \dots, 9\}$:

$$\begin{aligned} P &= \{\{6\}, \{1,5\}, \{7,2,3,8\}, \{9,4\}\} \\ &= \{\{1,5\}, \{2,3,7,8\}, \{4,9\}, \{6\}\} \end{aligned}$$

Note

Union-Find data structure: Choose a canonical representative for each classes (e.g. the smallest element).

- `Find` *the canonical representative of some element*
- `Union` *combines two parts*

$$\mathrm{Union}(P, 5, 3) = \{\{1,2,3,5,7,8\}, \{4,9\}, \{6\}\}$$

Disjoint-set (Union-Find) of two set-partitions

$$P = \{\{1, 5\}, \{2, 3, 7, 8\}, \{4, 9\}, \{6\}\}$$
$$Q = \{\{1\}, \{3\}, \{2, 4\}, \{5, 6\}, \{7, 8\}, \{9\}\}$$

Then

$$P \cup Q = \{\{1, 5, 6\}, \{2, 3, 4, 7, 8, 9\}\}$$

Disjoint-set (Union-Find) of two set-partitions

- Store a partition P as a function Can_P:

$$\begin{array}{c|ccccccccc} i & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \mathrm{Can}_P & 1 & 2 & 2 & 4 & 1 & 6 & 2 & 2 & 4 \end{array}$$

Lemma

$$\mathrm{Can}_{P\cup Q} = (\mathrm{Can}_P \circ \mathrm{Can}_Q)^{\circ n/2}$$

```
setpart16 union(setpart16 p, setpart16 p) {
  setpart16 res = _mm_shuffle_epi8(p, q);
  res = _mm_shuffle_epi8(res, res);
  res = _mm_shuffle_epi8(res, res);
  return = _mm_shuffle_epi8(res, res);
}
```

Some more examples and speedup

Operation	Speedup
Sorting a list of bytes	21.3
Number of cycles of a permutation	41.5
Cycle type of a permutation	8.94
Number of inversions of a permutation	9.39
Inverting a permutation	2.02

Problems:

- missing primitive (eg: inverting a permutation)
- AVX2 and AVX512 deals in parallel on 2 or 4 registers of size 128 bits. Shuffle instruction doesn't cross 128 bits barriers.
- no support for the compiler
- need to rethink all the algorithms !

Some more examples and speedup

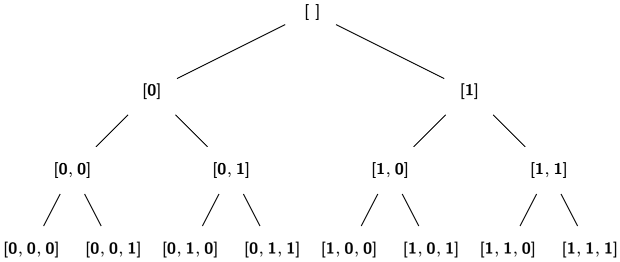
Operation	Speedup
Sorting a list of bytes	21.3
Number of cycles of a permutation	41.5
Cycle type of a permutation	8.94
Number of inversions of a permutation	9.39
Inverting a permutation	2.02

Problems:

- missing primitive (eg: inverting a permutation)
- AVX2 and AVX512 deals in parallel on 2 or 4 registers of size 128 bits. Shuffle instruction doesn't cross 128 bits barriers.
- no support for the compiler
- need to rethink all the algorithms !

Examples of recursively enumerated sets

Binary words: generation tree



Now that we know how to deals with each small objects,

How to generate them ?

Generation trees !

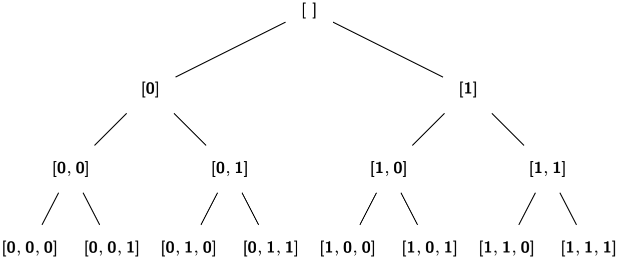
Now that we know how to deals with each small objects,

How to generate them ?

Generation trees !

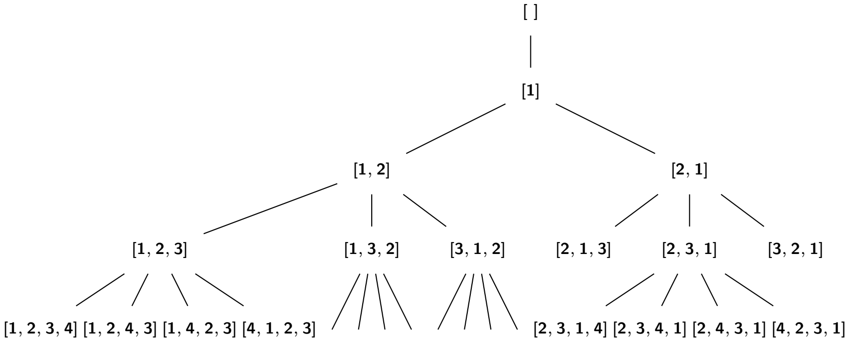
Examples of recursively enumerated sets

Binary words: generation tree



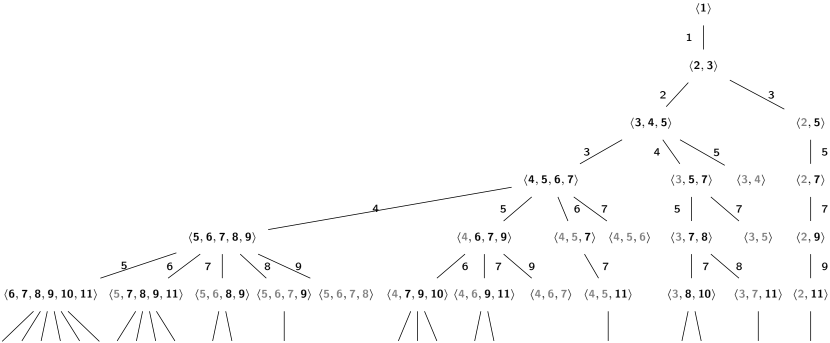
Examples of recursively enumerated sets (RESets)

Permutations: generation tree



Examples of recursively enumerated sets

The tree of numerical semigroups



Frobenius Coin Problem

Problem

What is the largest amount that cannot be obtained using only coins of specified denominations ?

Example: coins of 5 and 7:

- $12 = 5 + 7$
- $24 = 2 * 5 + 2 * 7$
- $25 = 5 * 5$
- $26 = 5 + 3 * 7$
- $27 = 4 * 5 + 7$
- $28 = 4 * 7$
- $29 = 24 + 5 = 3 * 5 + 2 * 7$
- ...

But you cannot make 23 . . .

Numerical semigroup of a given genus

Problem

- *Given an integer g (called the* ***genus****).*
- *Compute the number of minimal set of denominations such that they are exactly g non-obtainable amount (called* ***holes****).*

Example $g = 3$:

$$
\begin{array}{ccc}
\{4,5,6,7\} & \mapsto & \{-,-,-,4,5,6,7,8,9,10,11,\dots\} \\
\{3,5,7\} & \mapsto & \{-,-,3,-,5,6,7,8,9,10,11,\dots\} \\
\{3,4\} & \mapsto & \{-,-,3,4,-,6,7,8,9,10,11,\dots\} \\
\{2,7\} & \mapsto & \{-,2,-,4,-,6,7,8,9,10,11,\dots\}
\end{array}
$$

Problem to parallelize: A very unbalanced tree

Depth	Number of Semigroups
30	5 646 773
45	8 888 486 816

- The first node has 42% of the descendants;
- The second one node has 7.5% of the descendants;
- The 10 first node have 73% of the descendants;
- The 100 first node have 93% of the descendants;
- The 1000 first node have 99.4% of the descendants;
- Only 27 321 nodes have descendants at depth 45;
- Only 5 487 nodes have more than 10^3 descendants;
- Only 257 nodes have more than 10^6 descendants;

Cilk parallelization

- Vectorization (`MMX`, `SSE` instructions sets) and careful memory alignment;
- Aggressive loop unrolling: the main loop is unrolled by hand using some kind of Duff's device;
- Shared memory multi-core computing using `Cilk++` for low level enumerating tree branching;
- Partially derecursived algorithm using a stack;
- Avoiding all dynamic allocation during the computation: everything is computed "in place";
- Avoiding all unnecessary copy: Indirection in the stack.

Cilk parallelization

- Vectorization (`MMX`, `SSE` instructions sets) and careful memory alignment;
- Aggressive loop unrolling: the main loop is unrolled by hand using some kind of Duff's device;
- Shared memory multi-core computing using `Cilk++` for low level enumerating tree branching;

- Partially derecursived algorithm using a stack;
- Avoiding all dynamic allocation during the computation: everything is computed "in place";
- Avoiding all unnecessary copy: Indirection in the stack.

The results!

29 days 6 hours, on 32 Haswell core 2.3GHz.

- $2.59 \cdot 10^{15}$ monoids
- $6.22 \cdot 10^{17}$ bytes
- $1.02 \cdot 10^{9}$ monoids$/s$
- $2.46 \cdot 10^{11}$ bytes$/s$

g	n_g	g	n_g	g	n_g
0	1	24	282 828	48	38 260 496 374
1	1	25	467 224	49	62 200 036 752
2	2	26	770 832	50	101 090 300 128
3	4	27	1 270 267	51	164 253 200 784
4	7	28	2 091 030	52	266 815 155 103
5	12	29	3 437 839	53	433 317 458 741
6	23	30	5 646 773	54	703 569 992 121
7	39	31	9 266 788	55	1 142 140 736 859
8	67	32	15 195 070	56	1 853 737 832 107
9	118	33	24 896 206	57	3 008 140 981 820
10	204	34	40 761 087	58	4 880 606 790 010
11	343	35	66 687 201	59	7 917 344 087 695
12	592	36	109 032 500	60	12 841 603 251 351
13	1 001	37	178 158 289	61	20 825 558 002 053
14	1 693	38	290 939 807	62	33 768 763 536 686
15	2 857	39	474 851 445	63	54 749 244 915 730
16	4 806	40	774 614 284	64	88 754 191 073 328
17	8 045	41	1 262 992 840	65	143 863 484 925 550
18	13 467	42	2 058 356 522	66	233 166 577 125 714
19	22 464	43	3 353 191 846	67	377 866 907 506 273
20	37 396	44	5 460 401 576	68	612 309 308 257 800
21	62 194	45	8 888 486 816	69	992 121 118 414 851
22	103 246	46	14 463 633 648	70	1 607 394 814 170 158
23	170 963	47	23 527 845 502	Σ	4 198 294 061 955 752

Some conclusions

- Need to have good libraries for small object level optimizations.
- Cilk is very efficient to handle the balancing, but

- GCC / ICC support for Cilk dropped in next release
- Only shared memory, need for distributed (`YewPar` Blair Archibald)

- Memory management is very important; reusable code ?