High Performance Computing Experiments in Enumerative and Algebraic Combinatorics

Florent Hivert*
Laboratoire de Recherche en Informatique (UMR CNRS 8623), Univ. Paris Sud 11, Paris, France
Bureau 33, Bâtiment 650, Université Paris Sud 11
ORSAY CEDEX, FRANCE 91405
Florent.Hivert@lri.fr

CCS CONCEPTS

• **Mathematics of computing → Mathematical software performance**; **Combinatorics**; **Permutations and combinations**; • **Theory of computation → Parallel algorithms**;

KEYWORDS

Multithreaded programming; work-stealing; computer algebra; combinatorics;

ACM Reference format:
Florent Hivert. 2017. High Performance Computing Experiments in Enumerative and Algebraic Combinatorics. In *Proceedings of PASCO 2017, Kaiserslautern, Germany, July 23-24, 2017,* 4 pages.
DOI: 10.1145/3115936.3115938

The goal of this abstract is to report on some parallel and high performance computations in combinatorics, each involving large datasets generated recursively: we start by presenting a small framework implemented in Sagemath [12] allowing performance of map/reduce like computations on such recursively defined sets. In the second part, we describe a methodology used to achieve large speedups in several enumeration problems involving similar map/reduced computations. We illustrate this methodology on the challenging problem of counting the number of numerical semigroups [5], and present briefly another problem about enumerating integer vectors upto the action of a permutation group [2]. We believe that these techniques are fairly general for those kinds of algorithms.

1 MAP-REDUCE IN COMBINATORICS

In this first part, we present a small framework implemented in Sagemath [12] allowing performance map/reduce like computations on large recursively defined sets. Map-Reduce is a classical programming model for distributed computations where one maps a function on a large data set and uses a reduce function to summarize all the produced information. It has a large range of intensive applications in combinatorics:

- Compute the cardinality;
- More generally, compute any kind of generating series;
- Test a conjecture: i.e. find an element of S satisfying a specific property, or check that all of them do;
- Count/list the elements of S having this property.

Use cases in combinatorics often have two specificities: First of all, due to combinatorial explosion, sets often don't fit in the computer's memory or disks and are enumerated on the fly. Then, many problems are flat, leading to embarassingly parallel computations which are easy to parallelize. However, a second very common use case is to have data sets that are described by recursion tree which may be heavily unbalanced (see Section 2.2 for an example).

The framework [4] we developed works on the following input: A **recursively enumerated set** given by:

- the `roots` of the recursion
- the `children` function computing
- the `postprocessing` function that can also filter intermediate nodes

Then, a **Map/Reduce problem** is given by:

- the `mapped` function
- the `reduce_init` function
- the `reduce` function

Here is an example where we count binary sequence of length 15:

```
sage: S = RecursivelyEnumeratedSet( [[]],
....:  lambda l: [l+[0], l+[1]] if len(l) <= 15 else [],
....:  post_process = lambda x : x if len(x) == 15 else None,
....:   structure='forest', enumeration='depth')
sage: sage: S.map_reduce(
....:   map_function = lambda x: 1,
....:   reduce_function = lambda x,y: x+y,
....:   reduce_init = 0 )
32768
```

This framework uses a multi-process implementation of a work-stealing algorithm [1], and scales relatively well, as shown below in a typical computation:

# processors	1	2	4	8
Time (s)	250	161	103	87

Though it doesn't really qualify as HPC, it allowed to efficiently parallelize a dozen of experiments ranging from Coxeter group and representation theory of monoids to the combinatorial study of the C3 linearization algorithm used to compute the method resolution order (MRO) in scripting language such as Python and Perl [13].

*Supported by Horizon 2020 European Research Infrastructure project grant #676541 (OpenDreamKit)

2 OPTIMIZING COMBINATORICS

In this second part, we describe a methodology used to achieve large speedups in several enumeration problems. Indeed, in many combinatorial structures (permutations, partitions, monomials, young tableaux), the data can be encoded as a small sequence of small integers that can often be handled efficiently by a creative use of vector instructions. Through the challenging example of numerical monoids, I will then report on how Cilk++ allows for an extremely fast parallelization of the enumeration. Indeed, we have been able to enumerate sets with more that 2.10^{15} elements on a single multicore machine.

The methodology takes the following steps:

- Vectorization (MMX, SSE, AVX instructions sets) and careful memory alignment;
- Shared memory multi-core computing using Cilk++ for low level enumerating tree branching;
- Partially derecursived algorithm using a stack;
- Careful memory management: avoiding all dynamic allocation during the computation, avoiding all unnecessary copies (often needed to rewrite the containers);

2.1 Combinatorial structures and vector instructions

In many combinatorial structures (permutations, partitions, monomials, young tableaux), the data can be encoded as a small sequence of small integers that can often efficiently be handled thanks to vector instructions. For example, on the current x86 machines, small permutations ($N \leq 16$) are very well handled. Indeed thanks to machine instructions such as PSHUFB (Packed Shuffle Bytes), applying a permutation on a vector only takes a few cycles. Here are some examples of operation with their typical speedups:

Operation	Speedup
Inverting a permutation	1.28
Sorting a list of bytes	21.3
Number of cycles of a permutation	41.5
Number of inversions of a permutation	9.39
Cycle type of a permutation	8.94

As a more concrete example, here is how to sort an array of 16 bytes:

```
// 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},
        [...]
    }};

Vect16 sort(Vect16 a) {
  for (Perm16 round : rounds) {
   Vect16 minab, maxab, blend, mask, b = a.permuted(round);
    mask = _mm_cmplt_epi8(round, Perm16::one);
    minab = _mm_min_epi8(a, b);
    maxab = _mm_max_epi8(a, b);
    a = _mm_blendv_epi8(minab, maxab, mask);
  }
  return a;
}
```

Unfortunately, this requires rethinking all the algorithms, and there is nearly no support by the compiler.

2.2 Numerical semigroups

We present now an application which is particularly challenging. The goal is to enumerate or test a conjecture on so-called *numerical semigroups*. This part is joint work with Jean Fromentin [5].

Definition 1. *A* numerical semigroup S *is a subset of* $\mathbb{N}$ *containing* 0*, closed under addition and of finite complement in* $\mathbb{N}$*.*

For example the set

$$S_E = \{0, 3, 6, 7, 9, 10\} \cup \{x \in \mathbb{N}, x \geq 12\} \tag{1}$$

is a numerical semigroup. We need a little terminology:

Definition 2. *Let S be a numerical semigroup. We define*

- $g(S) = \mathrm{card}(\mathbb{N} \setminus S)$*, the* genus *of S;*
- $f(S) = \max(\mathbb{Z} \setminus S)$*, the* Frobenius *of S;*
- $c(S) = f(S) + 1$*, the* conductor of S.

For example the genus of S_E is 6, the cardinality of $\{1, 2, 4, 5, 8, 11\}$, it is of Frobenius number 11 and of conductor 12.

For a given positive integer g, the number of numerical semigroups of genus g is finite and is denoted by n_g. In J.A. Sloane's *on-line encyclopedia of integer sequences* [10] we find the values of n_g for $g \leq 52$. These values were obtained by M. Bras-Amorós ([3] for more details).

To enumerate the semigroups, we need to organize them as a recursively enumerated set, that it to build a tree whose nodes at depth g are exactly the semigroups of genus g. We now explain the construction such a tree. Let S be a numerical semigroup. The set $S' = S \cup \{f(S)\}$ is also a numerical semigroup and its genus is $g(S) - 1$. As each integer greater than $f(S)$ is included in S' we have $c(S') \leq f(S)$. Therefore every semigroup S of genus g can be obtained from a semigroup S' of genus $g - 1$ by removing an element of S' greater than or equal to $c(S')$.

Definition 3. *A non-zero element x of a numerical semigroup S is said to be* irreducible *if it cannot be expressed as a sum of two non-zero elements of S. We denote by* $\mathrm{Irr}(S)$ *the set of all irreducible elements of S.*

Note that, the set $\mathrm{Irr}(S)$ is the minimal generating set of S relative to the inclusion ordering. Therefore to identify a numerical semigroup S, we only need to know its set $\mathrm{Irr}(S)$. We write such a semigroup by $\langle \mathrm{Irr}(S) \rangle$. For example, returning to the semigroup S_E, we find that $\mathrm{Irr}(S_E) = \{3, 7\}$, we therefore write $S_E = \langle 3, 7 \rangle$.

Proposition 1 (Proposition 7.28 of [9]). *Let S be a numerical semigroup and x an element of S. The set $S^x := S \setminus \{x\}$ is a numerical semigroup if and only if x is irreducible in S.*

Proposition 1 implies that every semigroup S of genus g can be obtained from a semigroup S' by removing a generator x of S that is greater than or equal to $c(S)$.

We construct the tree of numerical semigroups, denoted by T as follows: The root of the tree is the unique semigroup of genus 0, i.e. , $\langle 1 \rangle$ that is equal to $\mathbb{N}$. If S is a semigroup in the tree, the children of S are exactly the semigroups S^x where x belongs to

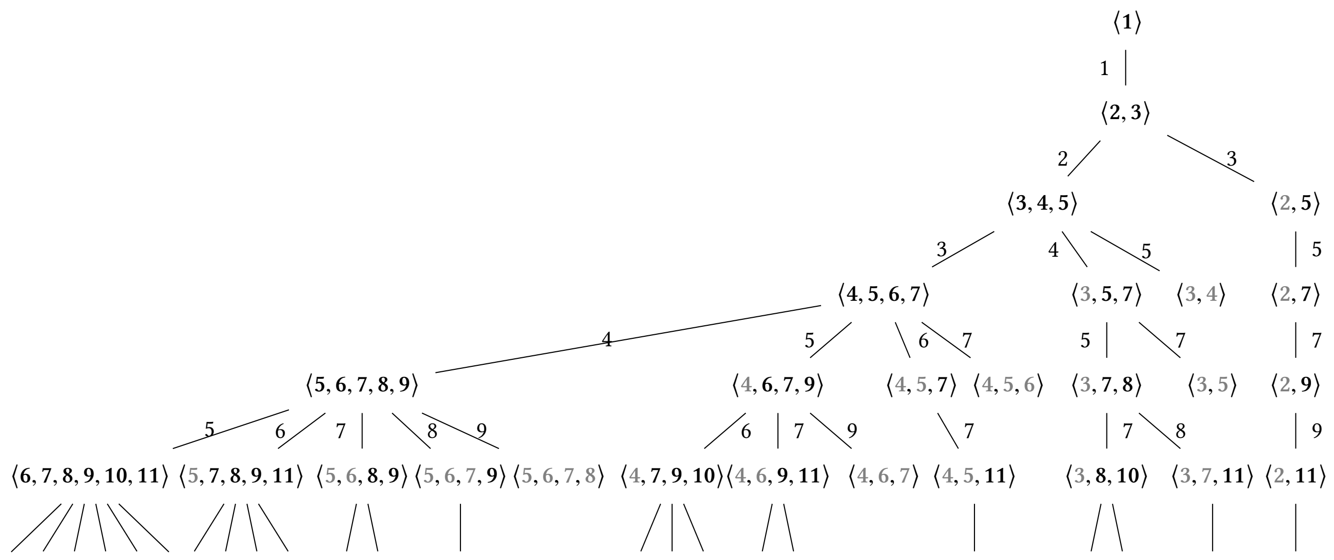

Figure 1: The first five layers of the tree T of numerical semigroups. A generator of a semigroup is it in gray if is not greater than $c(S)$. An edge between a semigroup S and its son S' is labelled by x if S' is obtained from S by removing x.

$\mathrm{Irr}(S) \cap [c(S), +\infty]$. By convention, when depicting the tree, the numerical semigroup S^x is in the left of S^y if x is smaller than y. With this construction, a semigroup S has depth g in T if and only if its genus is g, see Figure 1.

In [5] we describe a data structure for storing a numerical semigroup which fits particularly well the architecture of modern computers allowing very large optimizations. Thanks to these optimization computing a children in the tree from its father takes a time which is comparable to the time needed to simply copy it.

We think that exploring this tree is quite challenging as a parallel problem. Indeed, though non trivial, the computation of the children of a node is very fast and the tree is extremely unbalanced. This can be seen on Figure 1 or on the following experiments: We compare nodes at depth 30 anb 45: The number of nodes at depth 30 and 45 are 5 646 773 and 8 888 486 816. If we sort decreassingly the number of descendants at depth 45 of the nodes at depth 30, then

- 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;

Fortunately, the exploration of the tree is easily parallelized on a multicore machine using Cilk++. The idea here is that different branches of the tree can be explored in parallel by different cores of the computer. The tricky part is to ensure that all cores are busy, giving a new branch when a core is done with a former one. The Cilk++ [11] technology is particularly well suited for those kinds of problems. For our computation, we used the free version which is integrated in the latest version of the GNU C compiler [8].

Cilk is a general-purpose language designed for multithreaded parallel computing. The C++ incarnation is called Cilk++. The biggest principle behind the design of the Cilk language is that the programmer should be responsible for *exposing* the parallelism, identifying elements that can safely be executed in parallel; the runtime environment decide during execution how to actually divide the work between cores. The parallel features of Cilk++ are used mainly through the cilk_spawn keyword: used on a procedure call, it indicates that the call can safely operate in parallel with the remaining code of the current function. Note that the scheduler is not obliged to run this procedure in parallel; the keyword merely alerts the scheduler that it can do so.

We then write the following code for semigroup exploration:

```
void explore(const Semigroup &S) {
  unsigned long int nbr = 0;
  if (S.g < MAX_GENUS - STACK_BOUND) {
    //iterate along the children of S
    auto it = generator_iter<CHILDREN>(S);
    while (it.move_next()) {
      auto child = remove_generator(S, it.get_gen()).
      cilk_spawn explore(child);
      nbr++;
    }
    cilk_results[S.g] += nbr;
  }
  else explore_stack(S, cilk_results.get_array());
}
```

In the previous code, the function explore_stack performs a similar computation but iteratively (oposed as recursively) with the help of a stack.

To give some figure of the performance we managed to achieve, we performed a full exploration of the tree up to depth 70 on a 32 Haswell core at 2.3 Ghz. The number of monoid at depth 70 is 1607394814170158. It tooks $2.528 \cdot 10^6$ s (29 days and 6 hours) exploring $2590899247785594 = 2.59 \cdot 10^{15}$ monoids at a rate of $1.02 \cdot 10^9$ monoids per second. Each monoid is stored in 240 bytes. Storing all the computed monoids would take $6.22 \cdot 10^{17}$ bytes of data, which means that we generated $2.46 \cdot 10^{11}$ bytes of data per second.

2.3 N. Borie algorithm for integer vector modulo permutation groups

We briefly report on another successful optimization using the same methodology. We optimized an algorithm due to N. Borie for enumerating integer vector modulo permutation groups [2]. The problem is the following: we are given a subgroup G of the symmetric group S_n. It acts by permutation of coordinates on the vectors in $\mathbb{N}^n$. The problem is to generate one vector in each orbit. Note that there are infinitely many such vectors; in practice one usually wants to enumerate the vectors with a given sum or content.

N. Borie designed a tree structure on those vectors which allows to enumerate them recursively. At the level of each node, a relatively complicated computation is done involving partial lexicographic comparison and a hash table to avoid some duplication. The goal was to optimize the particular case of small groups where $n \leq 16$. The development went along the following steps:

- permutation, vectors and lexicographic comparison using vector instructions;
- recursive enumeration using `Cilk++`
- used thread local strorage for the hash table at the level of each node
- designed a handmade hash table to avoid dynamic allocation and adapted to the specific use-case

This last step is due to a very specific use case for the hash table: we needed it to store a dynamic set where we only add elements and never remove one, and we clear the hash table very often. Profiling showed that the hash table may grow up to thousand of elements but, on the average, is only cleared when containing 2.5 elements ! We decided therefore to use a closed bounded hash table together with a linked list of used buckets to be able to clear the table quickly.

Altogether, we compared our optimized version with an already optimized non-parallel compiled version using the Python compiler Cython. Computing the 375810 integer vectors of sum 25 for the largest transitive subgroup of S_{16} took 9min 23s on a single core with Sage's code, whereas our code is able to do it in 0.503s on 8 cores for a speedup of 1112 times. Finally, the code (not yet released) is downloadable at [7].

3 CONCLUSION

As a conclusion, we'd like to comment on the main technology used here, namely `Cilk++`. It is very efficient at balancing our work on a shared memory machine. The following table show timings where `C++` is a reference serial implementation and the other column shows the number of `Cilk++` threads:

Threads	C++	1	2	4	8	12
Time (s)	3588	3709	1865	932.4	486.8	325.7
Speedup Cilk	1.03	1.	1.99	3.97	7.61	11.39

However for the GCC implementation, we feel that it is not completely mature. We indeed found a core bug [6], we describe here briefly: in C/C++, when a parameter is passed to a function by value, the calling function is responsible to the construction (including the allocation) of the parameter. It is responsible to their destruction too. However, with `Cilk++`, it is possible to have the calling function stolen and therefore executed concurrently on another thread, *before* the called function returns. The problem was that the parameter was destroyed too early in this case. Here is a small code sample to reproduce the problem:

```
void walk(std::vector<int> v, unsigned size) {
  if (v.size() < size)
    for (int i=0; i<8; i++) {
      std::vector<int> vnew(v); vnew.push_back(i);
     // The vnew parameter below is destroyed too early
      cilk_spawn walk(vnew, size);
    }
}
```

The bug was corrected quickly but we learned along the way that GCC is considering deprecating and stopping support for the `Cilk++` features. We feel that this is a big loss for our kinds of computation.

Altogether, work stealing is very efficient to parallelize those kinds of computation in a shared memory machine, but to go further, we badly need an efficient distributed work-stealing framework.

REFERENCES

[1] R. D. Blumofe and C. E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. *J. ACM* 46, 5 (1999), 720–748.

[2] N. Borie. 2013. Generation modulo the action of a permutation group. *Proceeding of The 25th International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC '13)* (2013), 767–778.

[3] M. Bras-Amorós. 2008. Fibonacci-like behavior of the number of numerical semigroups of a given genus. *Semigroup Forum* 76, 2 (2008), 379–384. DOI: https://doi.org/10.1007/s00233-007-9014-8

[4] J.-B. Priez F. Hivert and N. Cohen. 2016. *Parallel computations using RecursivelyEnumeratedSet and Map-Reduce.* The Sage Development Team. http://doc.sagemath.org/html/en/reference/parallel/sage/parallel/map_reduce.html

[5] Jean Fromentin and Florent Hivert. 2016. Exploring the tree of numerical semigroups. *Math. Comput.* 85, 301 (2016), 2553–2568. DOI: https://doi.org/10.1090/mcom/3075

[6] GNU. 2016. *Random segfault using local vectors in Cilk function.* GCC Bugzilla. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=80038

[7] Florent Hivert. 2016. *Integer vectors modulo the action of a permutation group.* https://github.com/hivert/IVMPG

[8] B.V. Iyer, R. Geva, and P. Halpern. 2012. Cilk™ Plus in GCC. In *GNU Tools Cauldron.* http://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=Cilkplus_GCC.pdf

[9] J. C. Rosales and P. A. García-Sánchez. 2009. *Numerical semigroups.* Developments in Mathematics, Vol. 20. Springer, New York. x+181 pages. DOI: https://doi.org/10.1007/978-1-4419-0160-6

[10] N. J. A. Sloane. 2014. The On-Line Encyclopedia of Integer Sequences. http://oeis.org/. (2014).

[11] Software.intel.com. 2013. Intel® Cilk™ Homepage. https://www.cilkplus.org/. (2013).

[12] W. A. Stein and others. 2016. *Sage Mathematics Software (Version 7.6).* The Sage Development Team. http://www.sagemath.org

[13] Nicolas M. Thiéry. 2016. *The C3 algorithm, under control of a total order.* The Sage Development Team. http://doc.sagemath.org/html/en/reference/misc/sage/misc/c3_controlled.html