High Performance Combinatorics

Florent Hivert

Mél : Florent.Hivert@lri.fr
Adresse universelle : http://www.lri.fr/~hivert

Objectifs

Montrer deux technologies qui permettent de rendre les calculs beaucoup plus efficaces :

- Instructions vectorielles entières pour les petits tableaux de nombres
- Parcours d'arbre récursif parallèle avec `Cilk++`

Objectifs

Montrer deux technologies qui permettent de rendre les calculs beaucoup plus efficaces :

- Instructions vectorielles entières pour les petits tableaux de nombres
- Parcours d'arbre récursif parallèle avec `Cilk++`

Objectifs

Montrer deux technologies qui permettent de rendre les calculs beaucoup plus efficaces :

- Instructions vectorielles entières pour les petits tableaux de nombres
- Parcours d'arbre récursif parallèle avec `Cilk++`

Avertissement ! ! !

Même si l'on obtient des gains très importants (x50, x500...), rien de tout cela n'est de la **Bonne Informatique**...

- Technologie pas mure, bugs dans les compilateurs...
- Pas portable, y compris d'une version à l'autre du compilateur...
- Tentatives en cours de normaliser les syntaxes...

Avertissement ! ! !

Même si l'on obtient des gains très importants (x50, x500...), rien de tout cela n'est de la **Bonne Informatique**...

- Technologie pas mure, bugs dans les compilateurs...
- Pas portable, y compris d'une version à l'autre du compilateur...
- Tentatives en cours de normaliser les syntaxes...

Avertissement ! ! !

Même si l'on obtient des gains très importants (x50, x500...), rien de tout cela n'est de la **Bonne Informatique**...

- Technologie pas mure, bugs dans les compilateurs...
- Pas portable, y compris d'une version à l'autre du compilateur...
- Tentatives en cours de normaliser les syntaxes...

Que retenir ?

Retenir

Les principes de parallélisation

- *Vectorisation automatique ou manuelle*
- *Parallélisation récursive*

La syntaxe n'est **PAS** stable.

Instruction vectorielles entières

Registre : `epi8,epu8` : 128 bits = 16 octets

Encore plus : AVX, AVX2, AVX512

- Opérations Arithmetiques/logiques : `and, or, add, sub, min, max, abs, cmp`
- Recherche de bits : `popcount`, `bfsd`

Encore plus intéressant pour nous :

- Manipulation de tableau : `blend, broadcast, shuffle`
- Comparaison de chaînes de caractères : `cmpistr` (lex, find).

Manipulations très efficaces !

Instruction vectorielles entières

Registre : `epi8,epu8` : 128 bits = 16 octets

Encore plus : AVX, AVX2, AVX512

- Opérations Arithmetiques/logiques : `and, or, add, sub, min, max, abs, cmp`
- Recherche de bits : `popcount`, `bfsd`

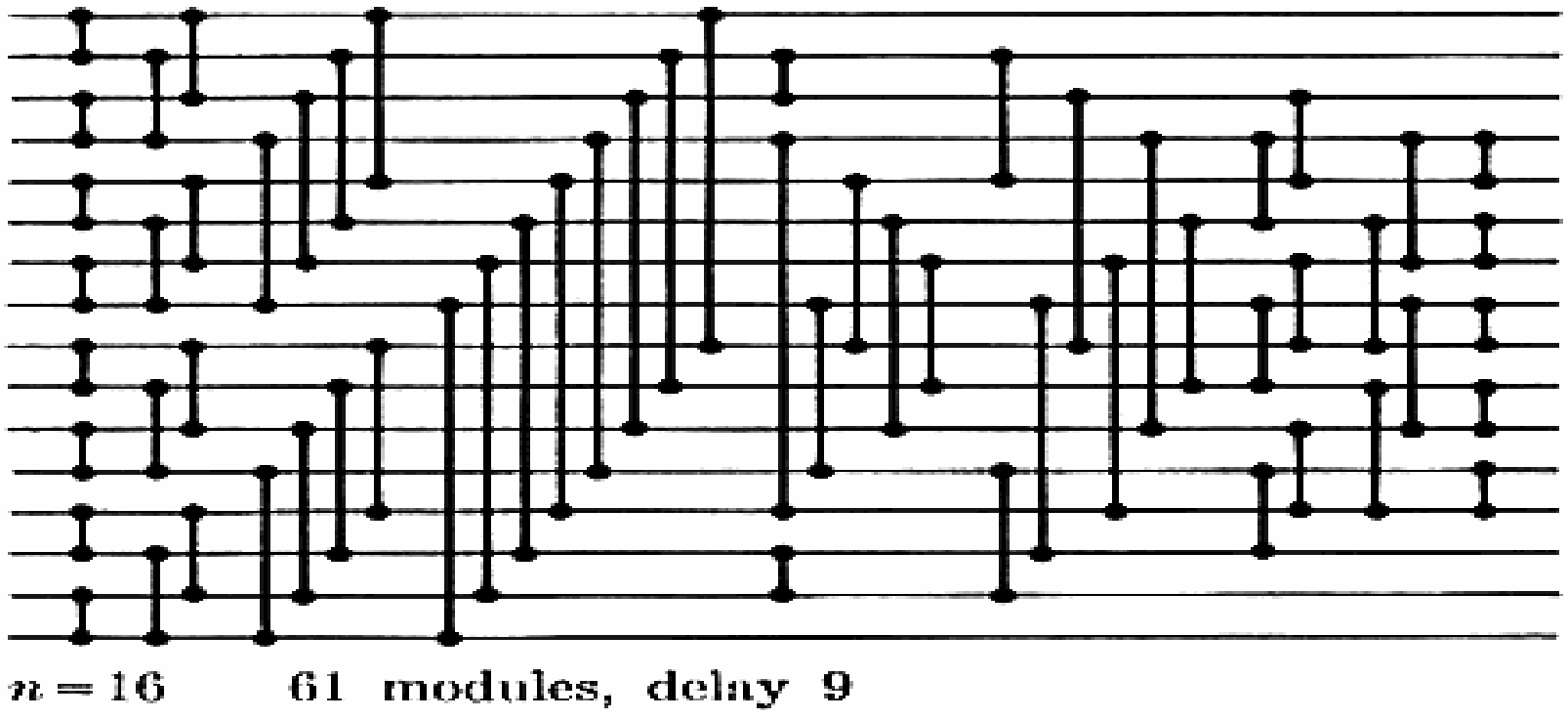
Encore plus intéressant pour nous :

- Manipulation de tableau : `blend, broadcast, shuffle`
- Comparaison de chaînes de caractères : `cmpistr` (lex, find).

Manipulations très efficaces !

Exemple : les réseaux de tris

Knuth AoCP3 Fig. 51 p. 229 :



```
// Sorting network Knuth AoCP3 Fig. 51 p 229.
static const array<perm, 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;
}
```

Exemple 2 : les cycles dans les permutations

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

$O(n)$ algorithme utilisant $O(n)$ memoire.

```
uint8_t nb_cycles_ref(Perm16 p) {
  Vect16 v {};
  int i, j, c = 0;
  for (i = 0; i < 16; i++) {
    if (v[i] == 0) {
      for (j=i; v[j] == 0; j = p[j]) v[j] = 1;
      c++;
    }
  }
  return c;
}
```

Exemple 2 : les cycles dans les permutations

$O(\log_2(n))$ algorithme utilisant $O(n)$ memoire.

Idée : propager le minimum le long des cycles :

i	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\sigma(i)$	d	6	b	f	5	2	c	4	9	1	7	0	8	a	e	3
$m := \min(i, \sigma)$	0	1	2	3	4	2	6	4	8	1	7	0	8	a	e	3
$\sigma^2(i)$	a	c	0	3	2	b	8	5	1	6	4	d	9	7	e	f
$s := \sigma^2(\min)$	7	8	0	3	2	0	8	2	1	6	4	a	1	4	e	3
$m := \min(m, s)$	0	1	0	3	2	0	6	2	1	1	4	0	1	4	e	3
$\sigma^4(i)$	4	9	a	3	0	d	1	b	c	8	2	7	6	5	e	f
$s := \sigma^4(\min)$	2	1	4	3	0	4	1	0	1	1	0	2	6	0	e	3
$m := \min(m, s)$	0	1	0	3	0	0	1	0	1	1	0	0	1	0	e	3
$\sigma^8(i)$	0	8	2	3	4	5	9	7	6	c	a	b	1	d	e	f
$s := \sigma^8(\min)$	0	1	0	3	0	0	1	0	1	1	0	0	1	0	e	3
$m := \min(m, s)$	0	1	0	3	0	0	1	0	1	1	0	0	1	0	e	3
$m == i$	1	1	.	1	.	.	.	.	.	.	.	.	.	.	1	.

Parallelisation avec `Cilk++`

Retenir

- *Programmation parallèle multifils en mémoire partagé.*
- *Extension du `C` et `C++`*
- *Ne pas écrire le parallélisme, mais l'exposer quand il est possible.*
- *Modèle Fork-Join :*

`cilk_spawn` `cilk_sync`

Parallelisation avec `Cilk++`

Retenir

- *Programmation parallèle multifils en mémoire partagé.*
- *Extension du `C` et `C++`*
- *Ne pas écrire le parallélisme, mais l'exposer quand il est possible.*
- *Modèle Fork-Join :*

`cilk_spawn` `cilk_sync`

Parallelisation avec `Cilk++`

Retenir

- *Programmation parallèle multifils en mémoire partagé.*
- *Extension du `C` et `C++`*
- *Ne pas écrire le parallélisme, mais l'exposer quand il est possible.*
- *Modèle Fork-Join :*

`cilk_spawn` `cilk_sync`

Parallelisation avec `Cilk++`

Retenir

- *Programmation parallèle multifils en mémoire partagé.*
- *Extension du `C` et `C++`*
- *Ne pas écrire le parallélisme, mais l'exposer quand il est possible.*
- *Modèle Fork-Join :*

`cilk_spawn` `cilk_sync`

Fibonacci

```
int fib(int n) {
  if (n < 2) {
    return n;
  }
  else {
    int x, y;
    x = cilk_spawn fib(n - 1);
    y = cilk_spawn fib(n - 2);
    sync;
    return x + y;
  }
}
```