Solutions pour le calcul scientifique intense en mémoire et CPU

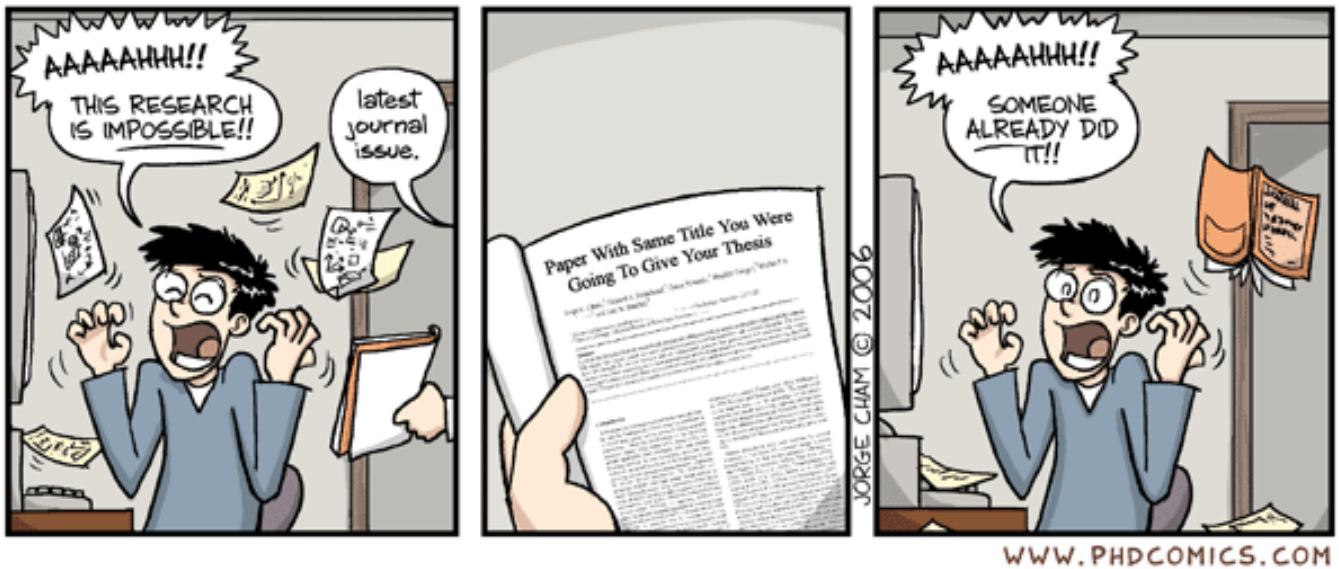
Rayan Chikhi

PyCONFR 2011

```
%CPU    %MEM     TIME+  COMMAND
101.3   39.7   1899:28 python
99.6    39.7   1899:28 python
100.6   39.7   1899:28 python
...
```

QUI SUIS-JE ?

- Doctorant à l'IRISA
- Recherche en bio-informatique (analyse de séquences ADN)



Mon expérience avec Python :

- Implémentation d'algorithmes de reconstitution de séquences à base de graphes (10k LOC)
- Enseignement d'introduction à Python + encadrement de projet (ENS Rennes, 2011)

Python et l'ADN

- Calculs non-numériques : rien pour nous dans SciPy.
- BioPython
  - Manipulation de séquences ADN (pas trop, ni trop grandes) : comparaison, recherche dans des bases.
- Chaque bio-informaticien a son répertoire de scripts (statistiques, parsers)
- Talk [1] de C. Titus Brown à la PyCon 2011.

1. Handling ridiculous amounts of data with probabilistic data structures

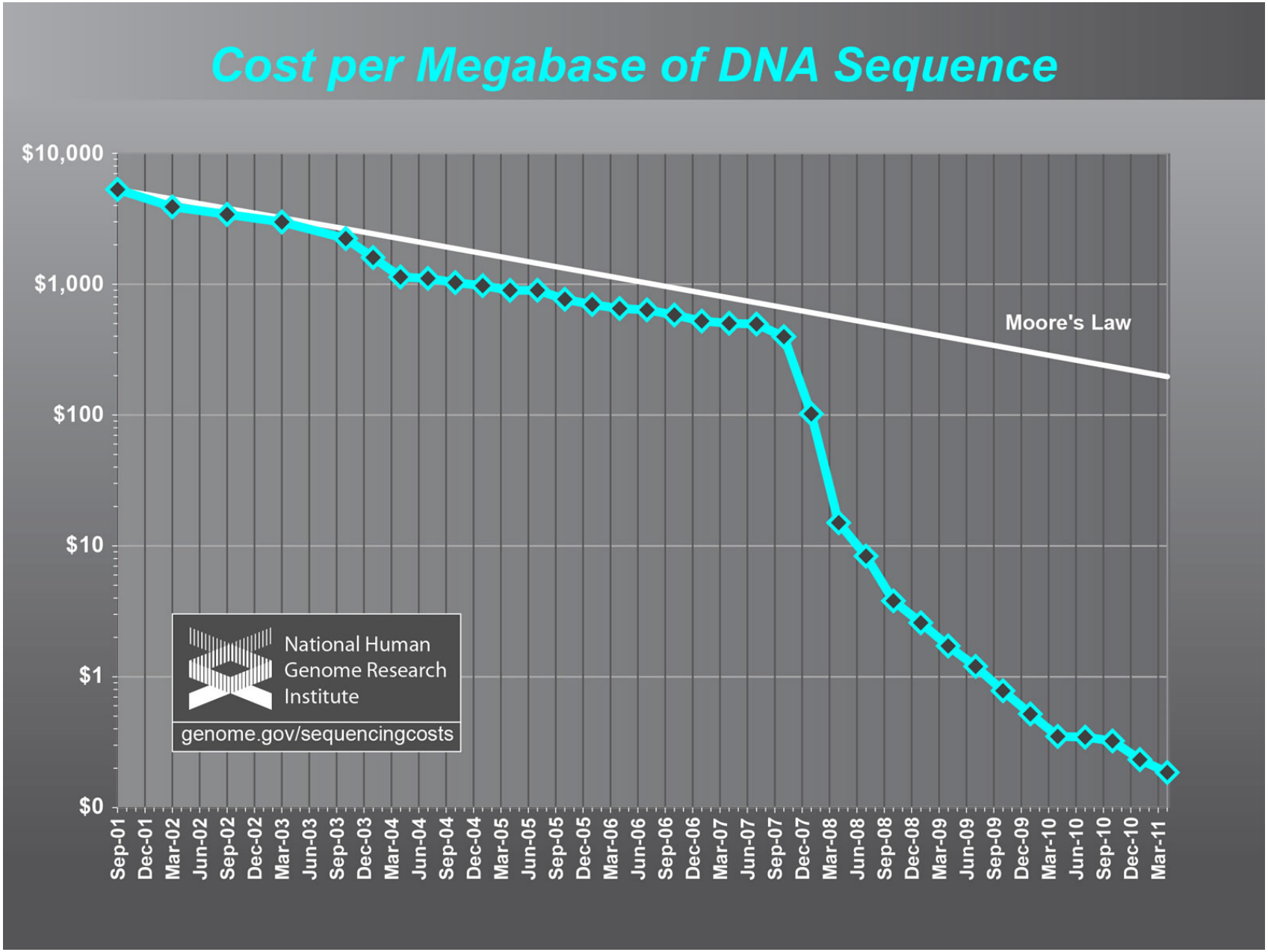
Python et l'ADN

- Calculs non-numériques : rien pour nous dans SciPy.
- BioPython
  - Manipulation de séquences ADN (pas trop, ni trop grandes) : comparaison, recherche dans des bases.
- Chaque bio-informaticien a son répertoire de scripts (statistiques, parsers)
- Talk [1] de C. Titus Brown à la PyCon 2011.
- Incroyable : C/C++ regagnent du terrain

1. Handling ridiculous amounts of data with probabilistic data structures

QUE S'EST-IL PASSÉ ?

QUE S'EST-IL PASSÉ ?



CONCRÈTEMENT : ASSEMBLAGE ADN

espèce dont on
ne connait pas
le génome

→ Séquenceur ADN →

ACGTGA
TTGGAA
TGATTG

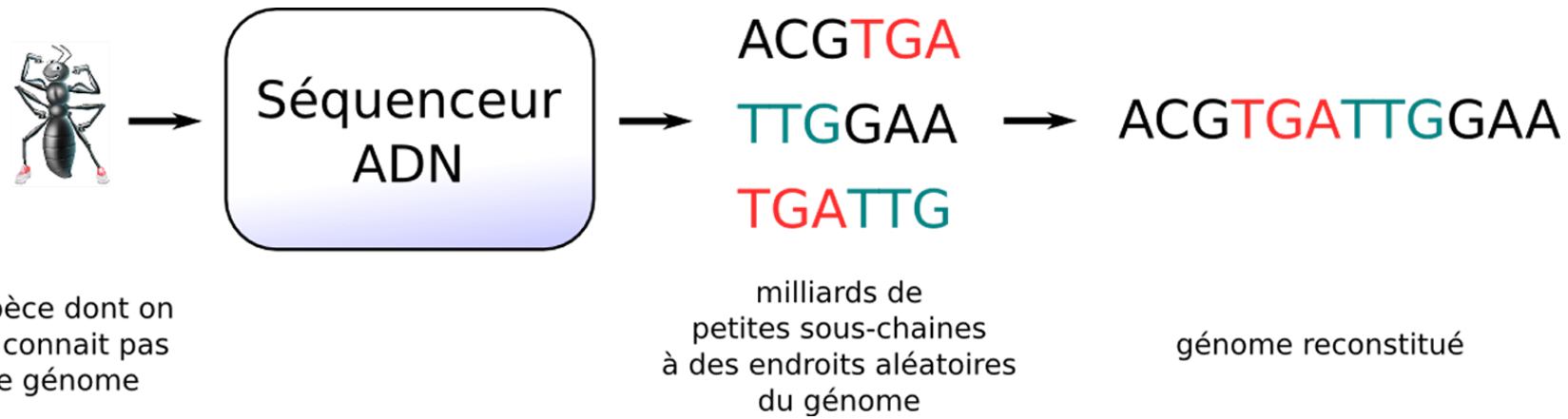
milliards de
petites sous-chaines
à des endroits aléatoires
du génome

→ ACGTGATTGGAA

génome reconstitué

2. ALLPATHS, SOAPdenovo, Velvet, etc..

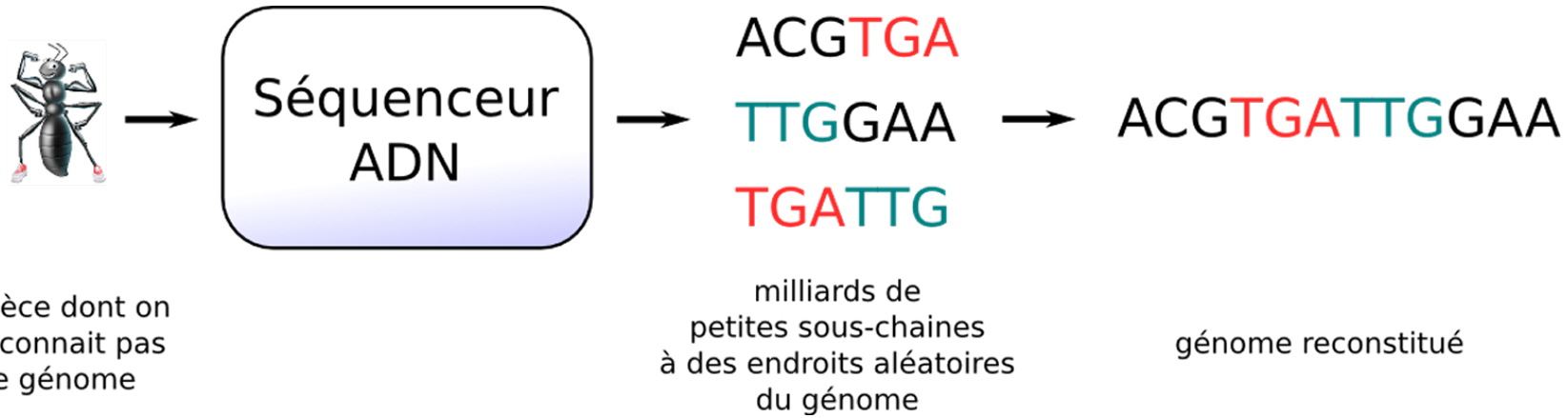
CONCRÈTEMENT : ASSEMBLAGE ADN



- Reconstituer un génome humain :
  - $>$ 200 GB séquences ADN en entrée.
  - génome : séquence de 3 GB en sortie
  - Algorithme heuristique linéaire en temps et mémoire (heureusement).
  - Non trivialement parallélisable, lourde occupation en RAM.

2. ALLPATHS, SOAPdenovo, Velvet, etc..

CONCRÈTEMENT : ASSEMBLAGE ADN



- Reconstituer un génome humain :
  - $>$ 200 GB séquences ADN en entrée.
  - génome : séquence de 3 GB en sortie
  - Algorithme heuristique linéaire en temps et mémoire (heureusement).
  - Non trivialement parallélisable, lourde occupation en RAM.
  - Beaucoup d'implémentations [2], généralement codées en C ou C++.
  - Typiquement besoin de 100 GB de RAM et qq semaines CPU.

2. ALLPATHS, SOAPdenovo, Velvet, etc..

POURQUOI PENSER À UTILISER PYTHON ?

Has anyone really been far even as decided to use even go want to do use Python

Nouvelle approche d'assemblage [3] :

- Méthode pour assembler à l'aide d'un grand nombre de sous-graphes ($<$ 10000 noeuds).
- Plus facilement parallélisable

Enjeux majeurs de l'implémentation :

1. Indexer 200 GB de données dans une table de hachage.

3. R.C., D. Lavenier, Localized genome assembly from reads to scaffolds : practical traversal of the paired string graph, WABI 11

POURQUOI PENSER À UTILISER PYTHON ?

Has anyone really been far even as decided to use even go want to do use Python

Nouvelle approche d'assemblage [3] :

- Méthode pour assembler à l'aide d'un grand nombre de sous-graphes ($<$ 10000 noeuds).
- Plus facilement parallélisable

Enjeux majeurs de l'implémentation :

1. Indexer 200 GB de données dans une table de hachage.
   - *Petit* `dict()` *deviendra grand.*
2. Les calculs suivants vont prendre des CPU-mois.

3. R.C., D. Lavenier, Localized genome assembly from reads to scaffolds : practical traversal of the paired string graph, WABI 11

POURQUOI PENSER À UTILISER PYTHON ?

Has anyone really been far even as decided to use even go want to do use Python

Nouvelle approche d'assemblage [3] :

- Méthode pour assembler à l'aide d'un grand nombre de sous-graphes ($<$ 10000 noeuds).
- Plus facilement parallélisable

Enjeux majeurs de l'implémentation :

1. Indexer 200 GB de données dans une table de hachage.
   - *Petit* `dict()` *deviendra grand.*
2. Les calculs suivants vont prendre des CPU-mois.
   - Accélérer CPython
   - Paralléliser

3. R.C., D. Lavenier, Localized genome assembly from reads to scaffolds : practical traversal of the paired string graph, WABI 11

Ingrédients / Plan de la présentation :

1. *Étendre Python avec des tables de hachage efficaces en extension C.*
2. *Librairies de graphes avec bindings tout prêts.*
3. *Calcul parallèle.*
4. *Calcul distribué.*

Ingrédients / Plan de la présentation :

1. *Étendre Python avec des tables de hachage efficaces en extension C.*
2. *Librairies de graphes avec bindings tout prêts.*
3. *Calcul parallèle. (1 seule machine, mémoire partagée)*
4. *Calcul distribué. (cluster avec plusieurs noeuds)*
5. *Accéleration avec quelques modules JIT.*

Étendre Python avec du C/C++

Méthodes testées :

1. CPython C API
   - code.py :

```
import mon_extension
mon_extension.f("test")
```

   - mon_extension.c :

```
#include <Python.h>
#include "traitement.h"
static PyObject *
mon_extension_f(PyObject *self, PyObject *args)
{
    const char *texte, *retour;
    if (!PyArg_ParseTuple(args, "s", &texte))
        return NULL;
    retour = traitement(texte);
    return Py_BuildValue("s", retour);
}
```

Étendre Python avec du C/C++

2. SWIG
   - code.py :

```
import mon_extension
mon_extension.f("test")
```

   - mon_extension.i :

```
%module mon_extension
%{
#define SWIG_FILE_WITH_INIT
#include "traitement.h"
%}

char* f(char* texte);
```

ÉTENDRE PYTHON AVEC DU C/C++

Autres méthodes :

1. Pyrex (et Cython)
   - langage hybride C/Python, permet aussi une interface vers le C.

```
def f(int j):
    cdef int k
    k = 0
    while k < j:
        k += 1
```

   - Ni du Python, ni du C : lock-in complet.
2. `boost::python`
   - Binding pour la librairies C++ Boost.
   - Plus maintenu depuis 2005..

RXP DES TABLES DE HASH C/C++

Tables testées :

1. `tr1::unordered_map` (C++)
2. `google_sparsehash` (C++)
3. `libchash` (C)
4. `Judy` (C)
5. `khash` (C)
6. `uthash` (C)

Meilleure taille mémoire : `google_sparse` (mais `resize()` problématique)
Meilleur compromis temps/mémoire : `unordered_map`
Meilleure table C : `libchash`

RXP DES TABLES DE HASH C/C++

Tables testées :

1. `tr1::unordered_map` (C++)
2. `google_sparsehash` (C++)
3. `libchash` (C)
4. `Judy` (C)
5. `khash` (C)
6. `uthash` (C)

Meilleure taille mémoire : `google_sparse` (mais `resize()` problématique)
Meilleur compromis temps/mémoire : `unordered_map`
Meilleure table C : `libchash`

Référencer toutes les *k*-sous-chaines d'un génome (*E. coli*) :

naive `dict()`	291 MB
custom `unordered_map`	99 MB

RXP DES TABLES DE HASH C/C++

Tables testées :

1. `tr1::unordered_map` (C++)
2. `google_sparsehash` (C++)
3. `libchash` (C)
4. `Judy` (C)
5. `khash` (C)
6. `uthash` (C)

Meilleure taille mémoire : `google_sparse` (mais `resize()` problématique)
Meilleur compromis temps/mémoire : `unordered_map`
Meilleure table C : `libchash`

Référencer toutes les *k*-sous-chaines d'un génome (*E. coli*) :

naive `dict()`	291 MB
custom `unordered_map`	99 MB
custom `sdarray`	42 MB

RXP DES LIBRAIRIES DE GRAPHES

1. python-graph [4] (Python pur)

```
from pygraph.classes.digraph import digraph
gr = digraph()
gr.add_node("n1")
gr.add_node("n2")
gr.add_edge( ("n1","n2") )
```

2. igraph [5] (C pur avec bindings)

```
import igraph
gr=igraph.Graph(0,directed=True)
gr.add_vertices(2)
gr.vs[0]["sequence"]="n1"
gr.vs[1]["sequence"]="n2"
gr.add_edge( (0, 1) )
```

4. http://code.google.com/p/python-graph/
5. http://igraph.sourceforge.net/
6. http://projects.skewed.de/graph-tool/
7. http://networkx.lanl.gov/

RXP DES LIBRAIRIES DE GRAPHES

1. python-graph [4] (Python pur)

```
from pygraph.classes.digraph import digraph
gr = digraph()
gr.add_node("n1")
gr.add_node("n2")
gr.add_edge( ("n1","n2") )
```

2. igraph [5] (C pur avec bindings)

```
import igraph
gr=igraph.Graph(0,directed=True)
gr.add_vertices(2)
gr.vs[0]["sequence"]="n1"
gr.vs[1]["sequence"]="n2"
gr.add_edge( (0, 1) )
```

python-graph : algorithmes de parcours lents
igraph : appels SWIG lents

Non testées : graph-tool [6] (dep : `boost`, `expat`, `Sci/NumPy` et `CGAL`) et Networkx [7] (Python pur).

4. `http://code.google.com/p/python-graph/`
5. `http://igraph.sourceforge.net/`
6. `http://projects.skewed.de/graph-tool/`
7. `http://networkx.lanl.gov/`

CALCUL PARALLÈLE

Quizz : calculs parallèles sans I/O, quel module choisir ?

1. module `threading`
2. module `multiprocessing`

8. `http://wiki.python.org/moin/ParallelProcessing`

CALCUL PARALLÈLE

Quizz : calculs parallèles sans I/O, quel module choisir ?

1. module `threading`
2. module `multiprocessing`

Réponse : `multiprocessing` évite le Global Interpreter Lock.

Les alternatives[8] sont basées sur `fork()` :

- Parallel Python
  - documentation minimaliste
- Forkmap : un `map()` parallèle
- Je passe sur les packages datant de 2003.

8. `http://wiki.python.org/moin/ParallelProcessing`

CALCUL DISTRIBUÉ

Python officiel [9] propose beaucoup de solutions, peu sont maintenues, une lection :

1. RPyC
   - serveur.py

```
import rpyc
class MyService(rpyc.Service):
  def exposed_get_answer(self):
    return 42
from rpyc.utils.server import ThreadedServer
ThreadedServer(MyService, port = 18861).start()
```

   - client.py

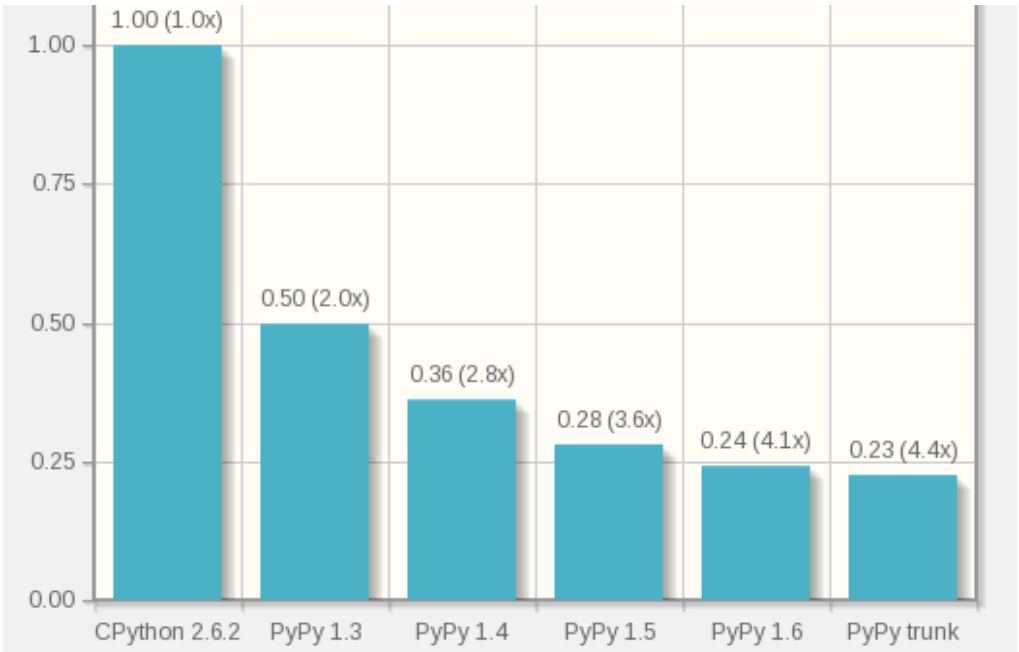
```
import rpyc
c = rpyc.connect("localhost", 18861)
c.root.get_answer()
```

2. Pour MPI : `pypar`
3. Pyro4 a l'air prometteur aussi.

9. `http://wiki.python.org/moin/ParallelProcessing`

RXP JIT

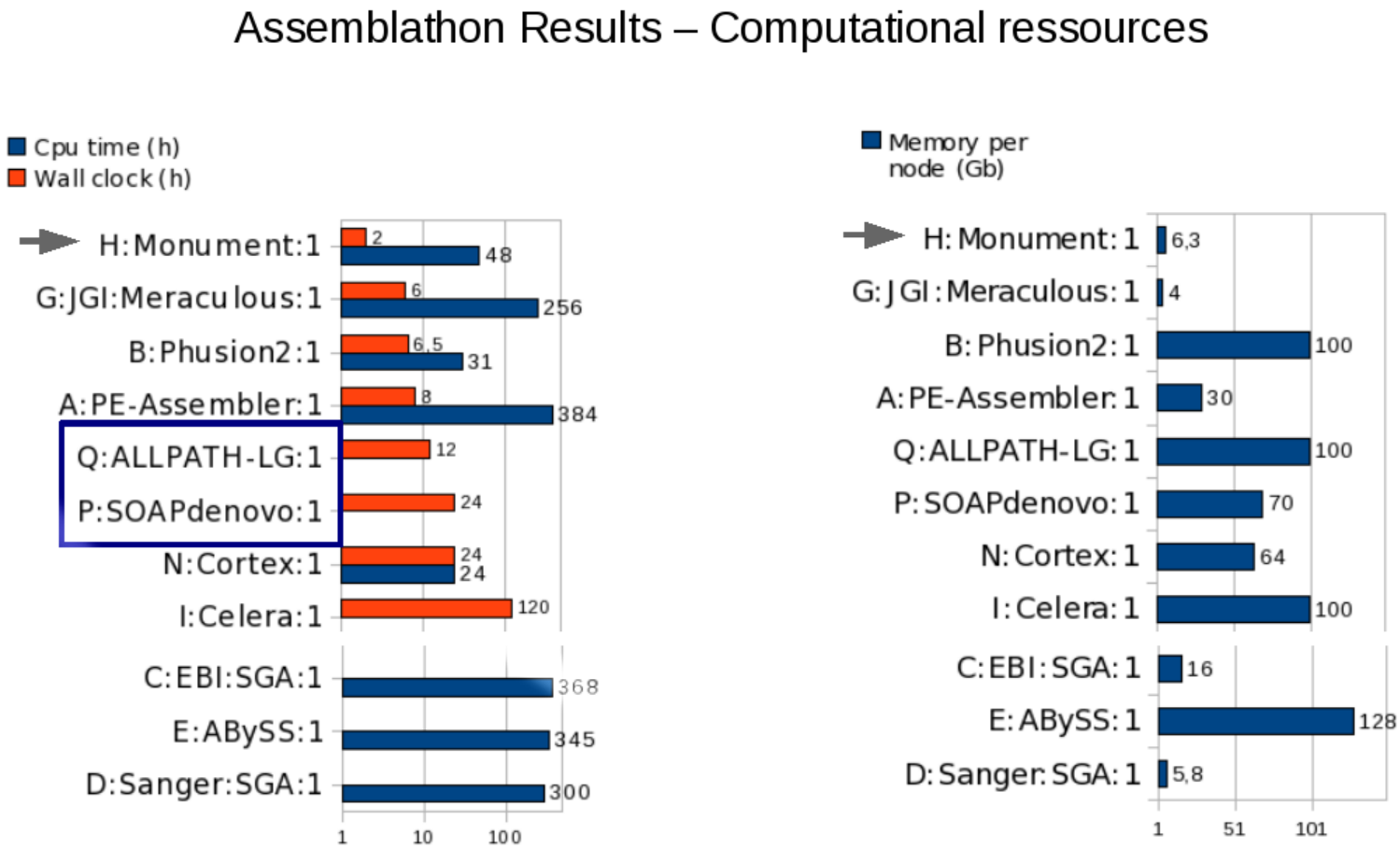
1. Module `psyco` : très efficace à mettre en place, mais uniquement 32bits.
   ```
   import psyco
   psyco.full()
   ```
2. PyPy : implémentation alternative de Python 2.7



   - Pas de support de l'API C jusqu'à récemment (alpha).

Le parallélisme a encore de beaux jours devant lui.

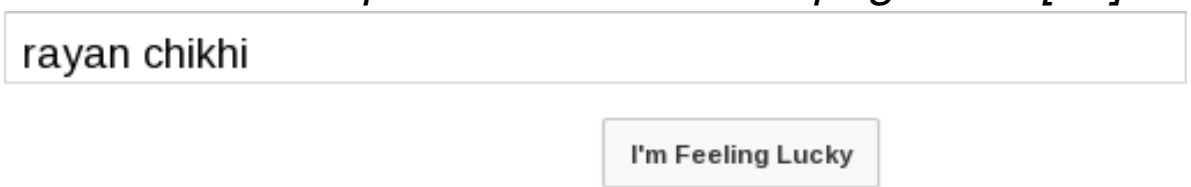
RÉSULTATS


Assemblathon Results – Computational ressources
Cpu time (h)
Wall clock (h)
H:Monument:1
G:JGI:Meraculous:1
B:Phusion2:1
A:PE-Assembler:1
Q:ALLPATH-LG:1
P:SOAPdenovo:1
N:Cortex:1
I:Celera:1
C:EBI:SGA:1
E:ABySS:1
D:Sanger:SGA:1
2
48
6
256
6,5
31
8
384
12
24
24
24
120
368
345
300
1
10
100
Memory per node (Gb)
H:Monument:1
G:JGI:Meraculous:1
B:Phusion2:1
A:PE-Assembler:1
Q:ALLPATH-LG:1
P:SOAPdenovo:1
N:Cortex:1
I:Celera:1
C:EBI:SGA:1
E:ABySS:1
D:Sanger:SGA:1
6,3
4
100
30
100
70
64
100
16
128
5,8
1
51
101

CONCLUSION

1. Python peut faire du traitement intensif en temps/mémoire
2. Les extensions C/C++ (e.g. SWIG) permettent de mieux gérer l'occupation mémoire.
3. Parallélisme (e.g. `multiprocessing`, `RPyC`) indispensable.

Retrouvez cette présentation sur ma page web [ou] le site PyCONFR.



ACKNOWLEDGEMENTS

- BioGenouest Platform and the Symbiose team @ IRISA

RÉSULTATS

Assemblathon Results – Scaffolds/Contigs NG50

(1 result per assembler, best scaffold NG50)

N50 = Scaffold/contig length at which you have covered 50% of total ***assembly*** length
NG50 = -- -- -- ***genome*** --