


BioBITs

Euromicro PDP 2010 - Pisa Italy - 17th Feb 2010

# Efficient Smith-Waterman on multi-core with FastFlow

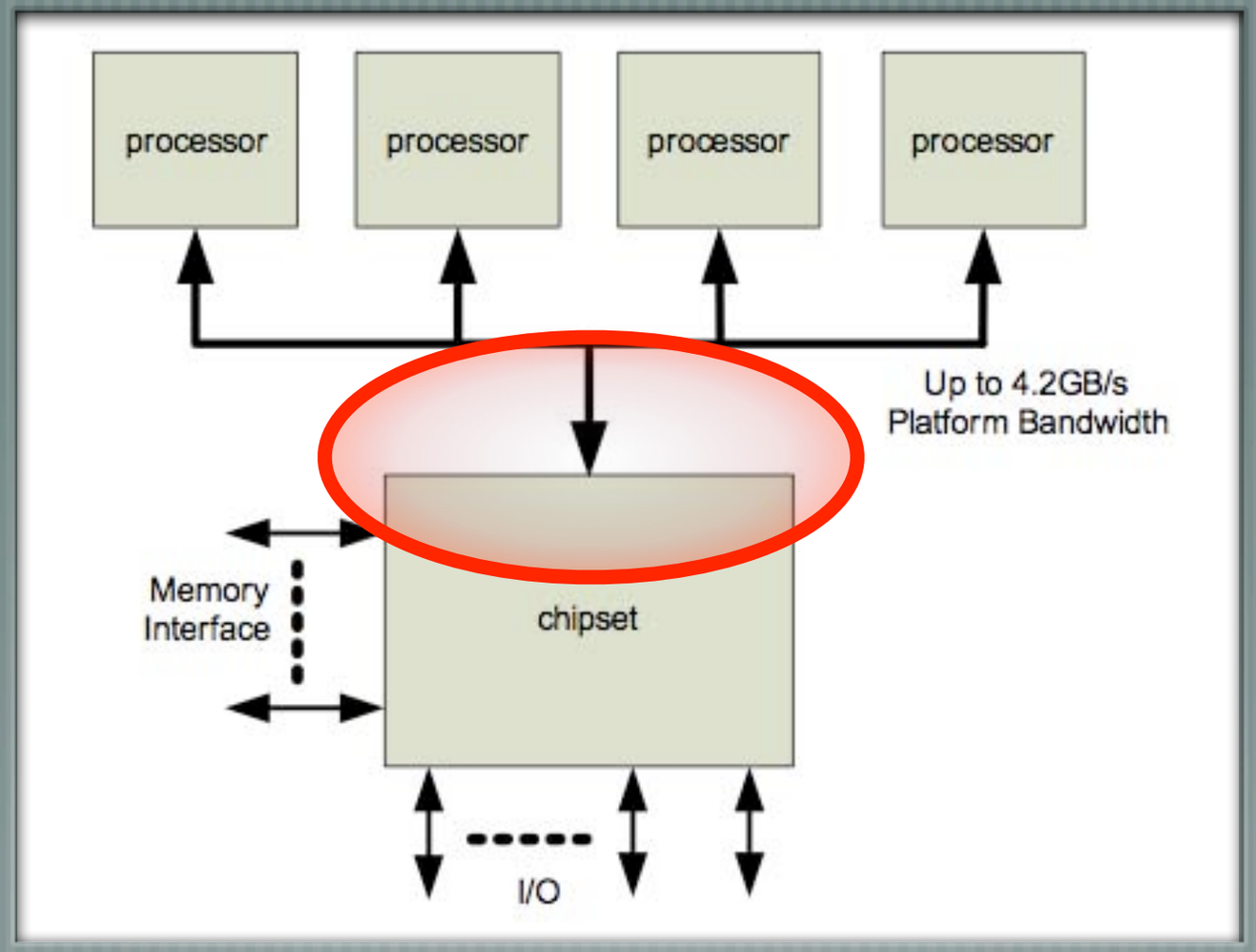
Marco Aldinucci
*Computer Science Dept. - University of Torino - Italy*

Massimo Torquati
*Computer Science Dept. - University of Pisa - Italy*

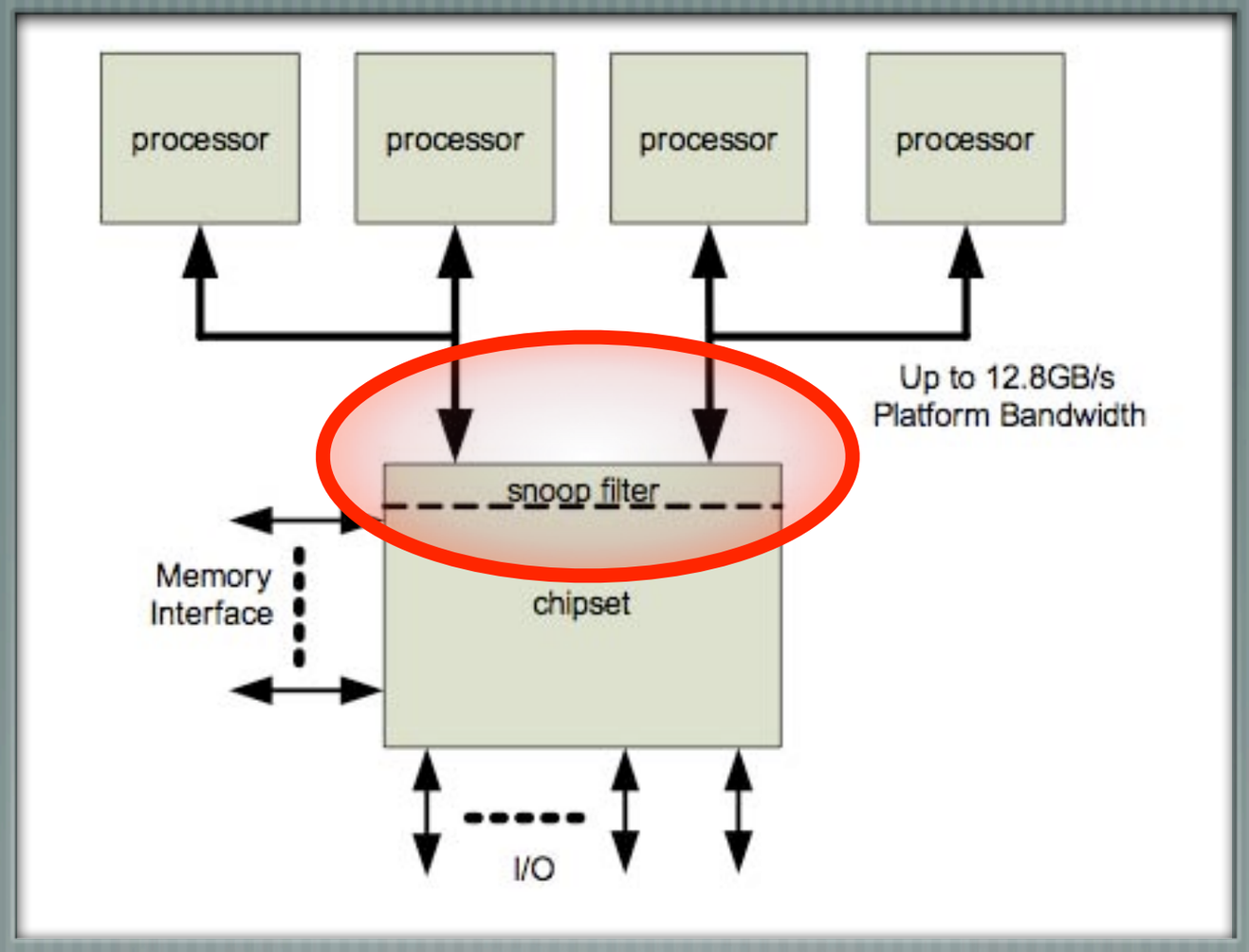
Massimiliano Meneghin
*IBM Technology Campus, Dublin Software Lab, Ireland*

# Outline

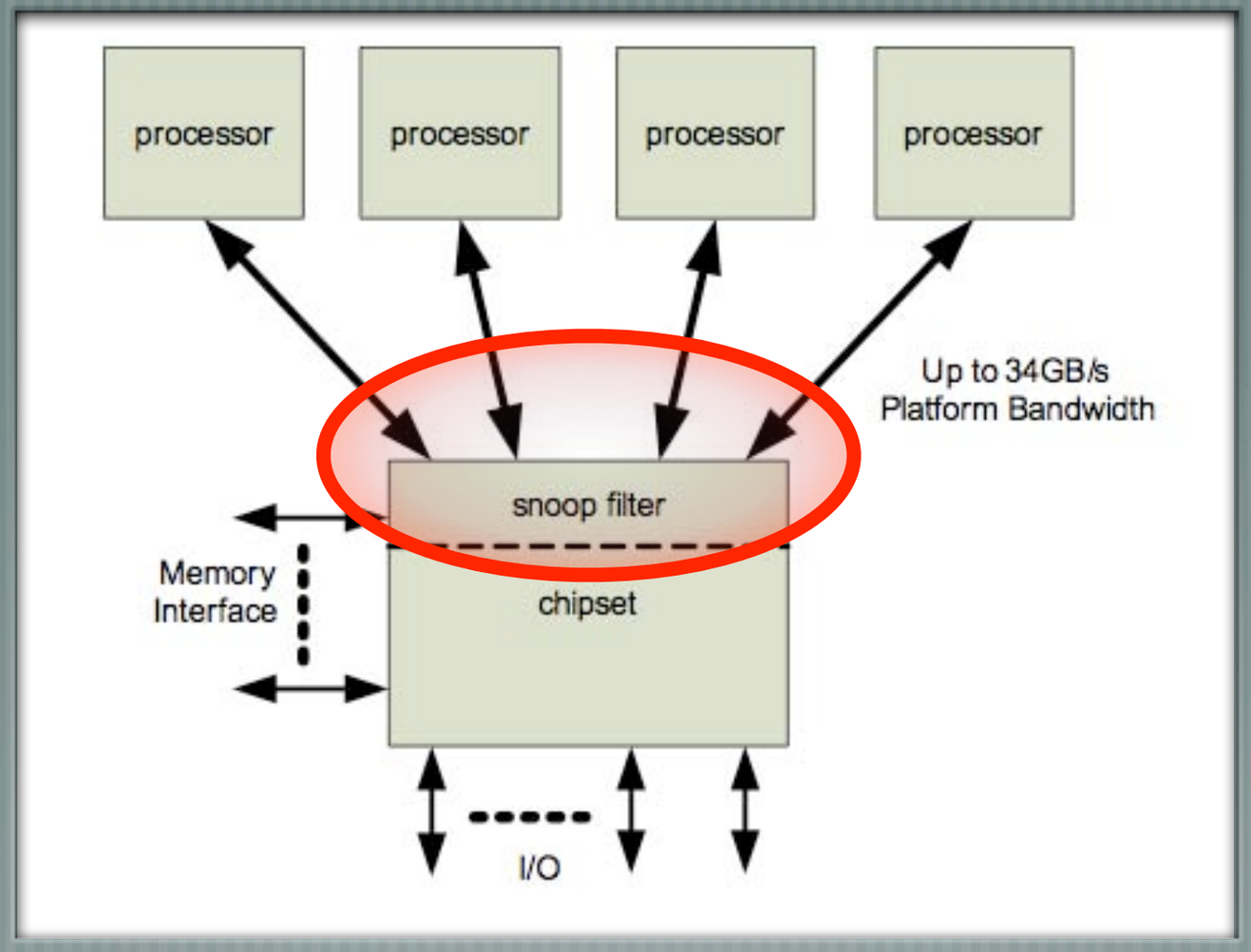
- Motivations
- FastFlow
  - Performance
- Smith-Waterman benchmark
  - Performance
- Conclusions & Commercial




# [< 2004] Shared Font-Side Bus (Centralized Snooping)



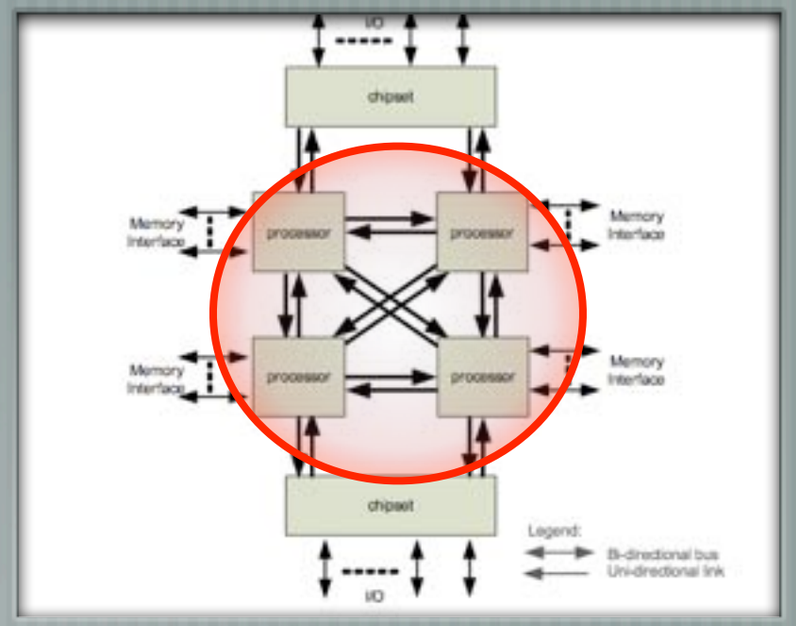

# [2005] Dual Independent Buses (Centralized Snooping)




# [2007] Dedicated High-Speed Interconnects (Centralized Snooping)

# This and next generation Multi-cores

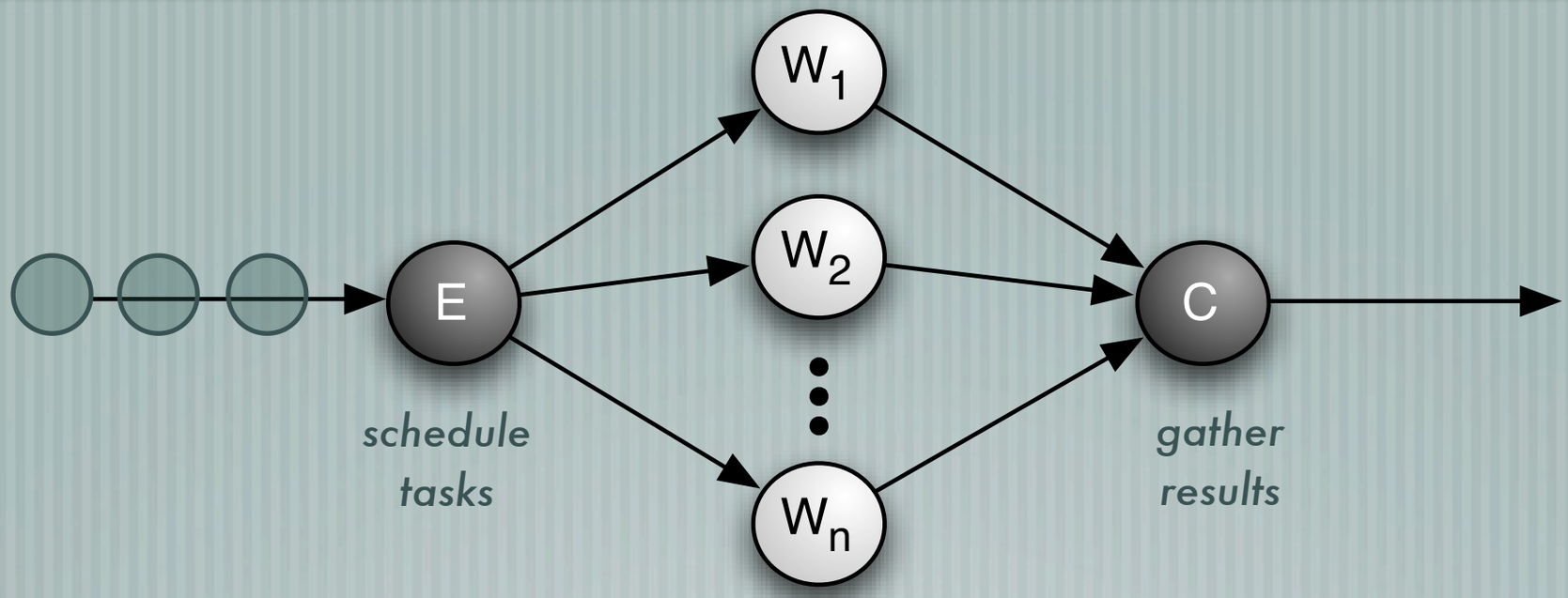
- Are programmed at "concurrent assembler" level
  - Complex, not portable, not efficient
- Exploit cache coherence
  - Memory fences are expensive
  - Will worsen with core count
  - Atomic ops do not solve the problem (still fences)
- Fine-grained parallelism is off-limits
  - I/O bound problems, High-throughput, Streaming, Irregular DP problems
  - Automatic and assisted parallelisation




[2009] i7 QuickPath
(MESI-F Directory Coherence)

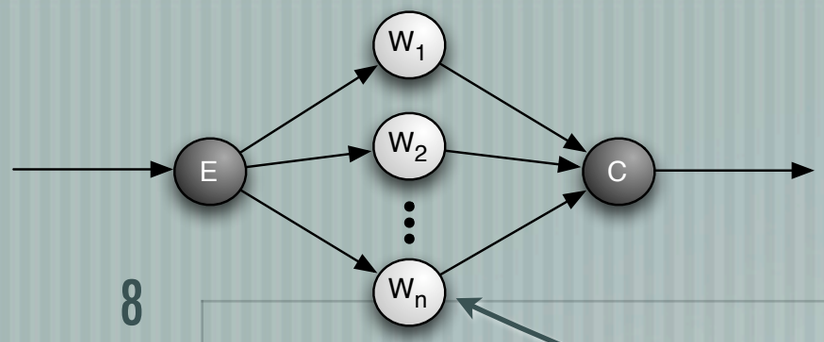
ERCIM CoreGRID BioBITs

# E.g. farm (a.k.a. master-workers)




- Common paradigm
  - Model foreach loop and Divide&Conquer
  - Exploit it as a high-order language construct
    - Why should we re-code it from scratch each application?
  - A C++ template factory exploiting highly optimised implementation

# E.g. farm with POSIX lock/unlock







ERCIM CoreGRID BioBITs

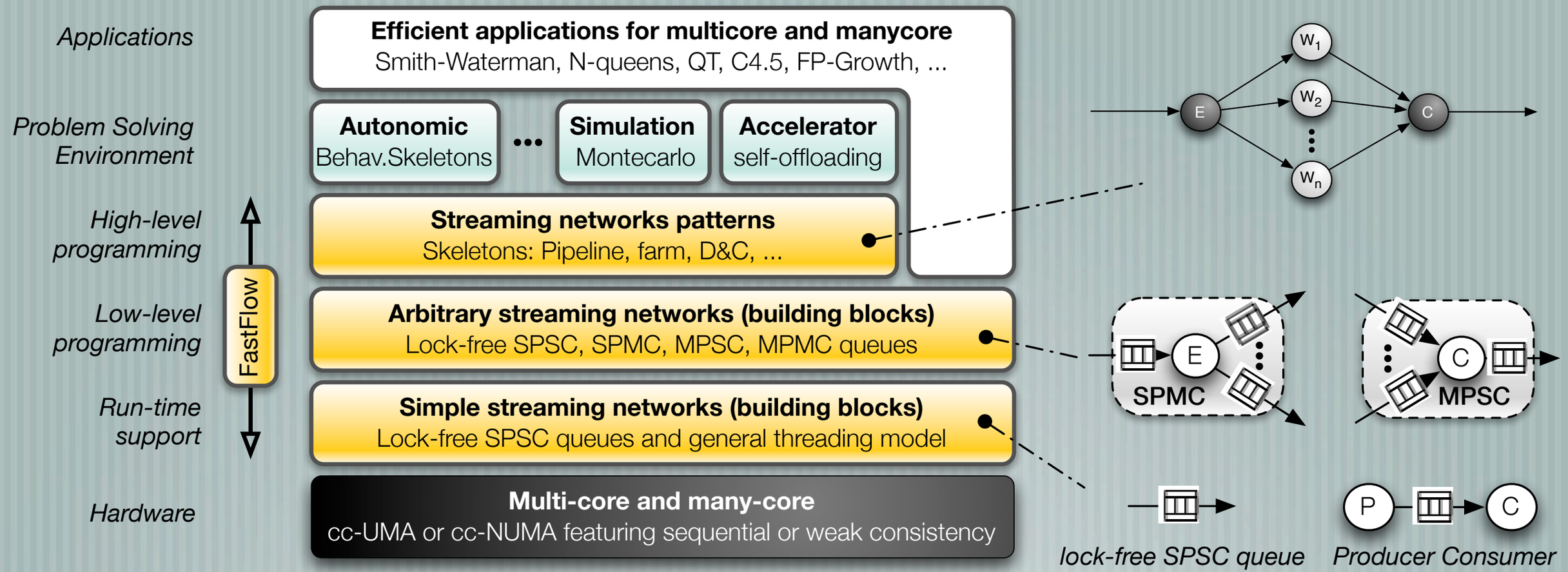
# Lock-free and CAS-free (fence-free)

- Single-Producer-Single-Consumer FIFO queues
  - Lamport et al. 1983 Trans. PLS (Sequential consistency only - **passive**)
  - Giacomoni et al. 2008 PPoPP (Relaxed consistencies (e.g. TSO) - **passive**)
- Multiple-Producers-Multiple-Consumers FIFO queues
  - with CAS (at least one) - passive ... a plethora
  - without CAS - passive ➠ **Cannot be done**
  - without CAS - active ➠ **FastFlow**





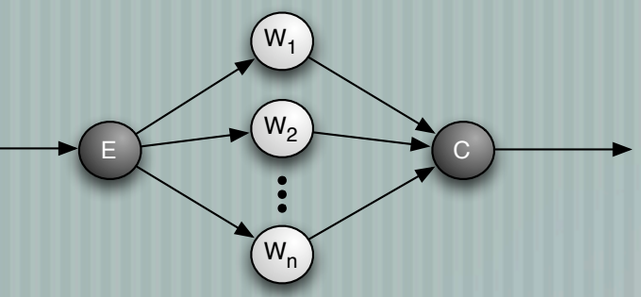
# FastFlow: A step forward

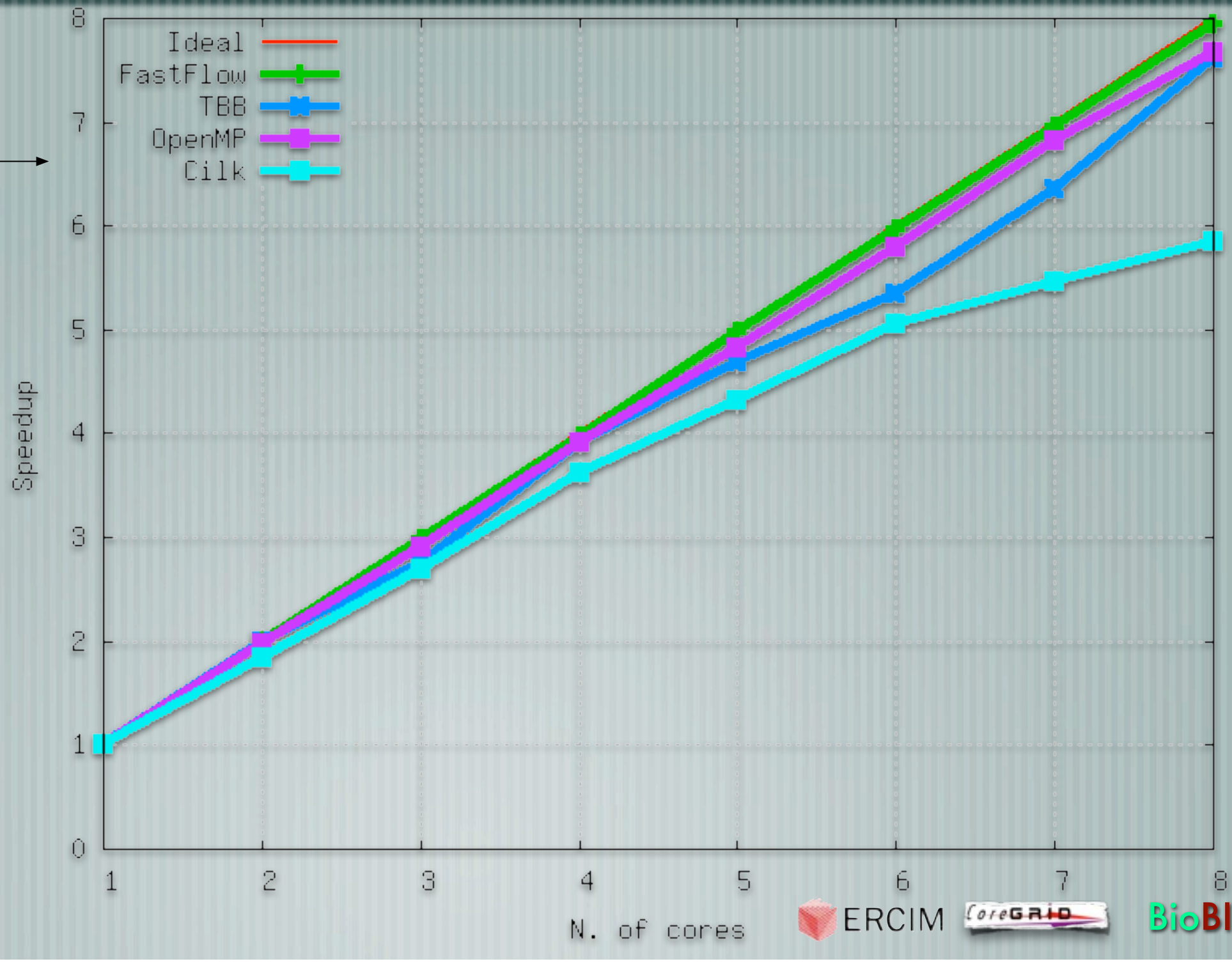



## High-level programming

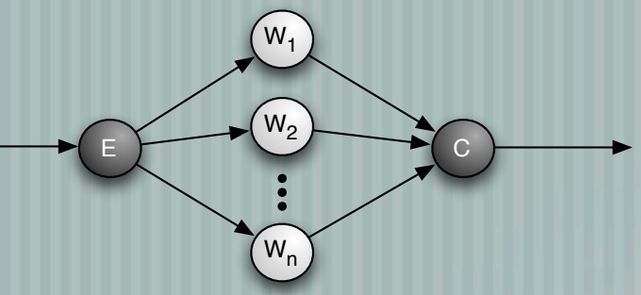
- Implemented on top of lock-free/fence-free non-blocking synchronizations
- C++ STL-like implementation

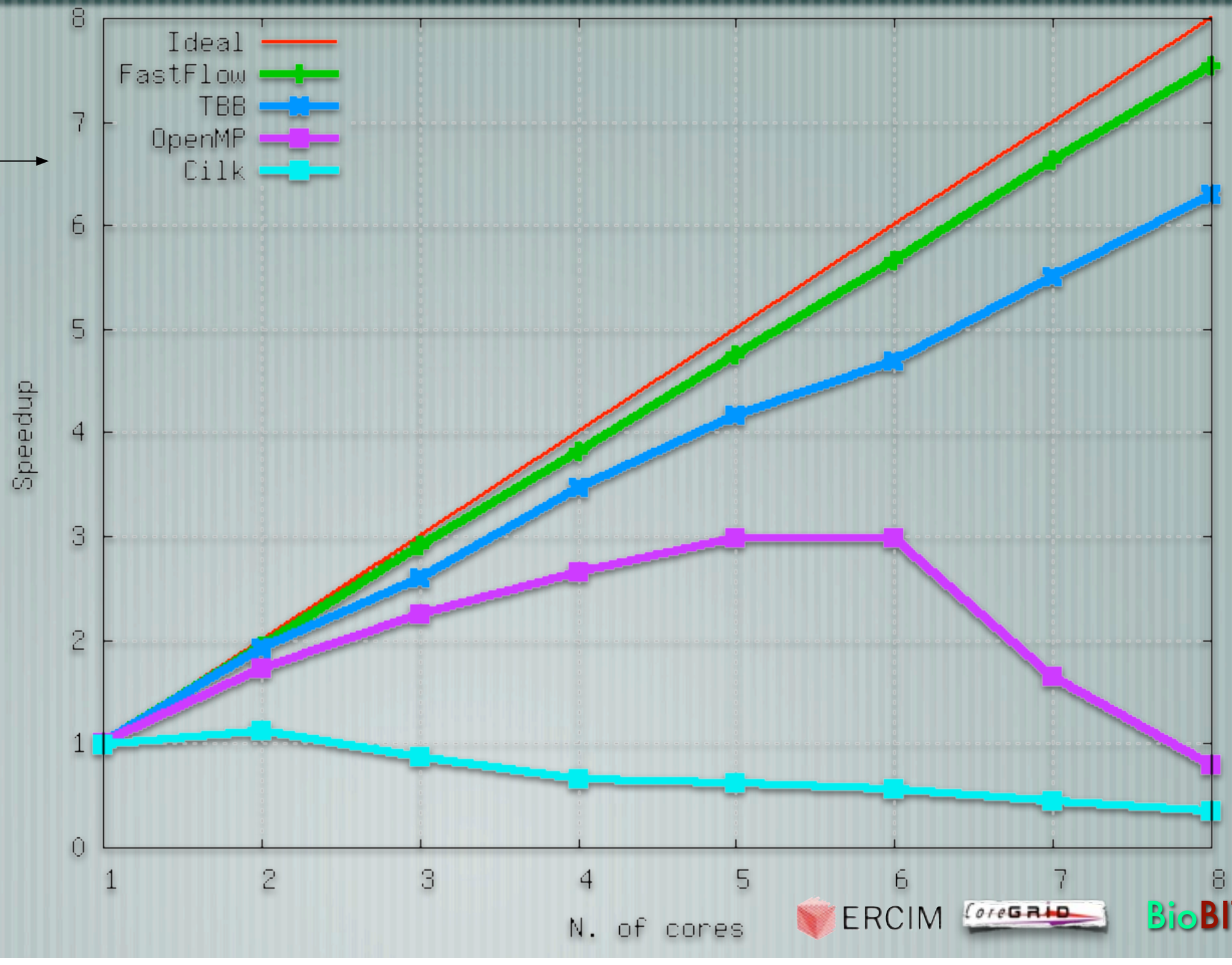
# Coarse grain (50 µS workload)



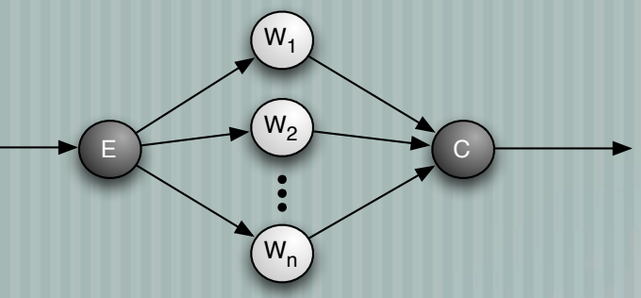



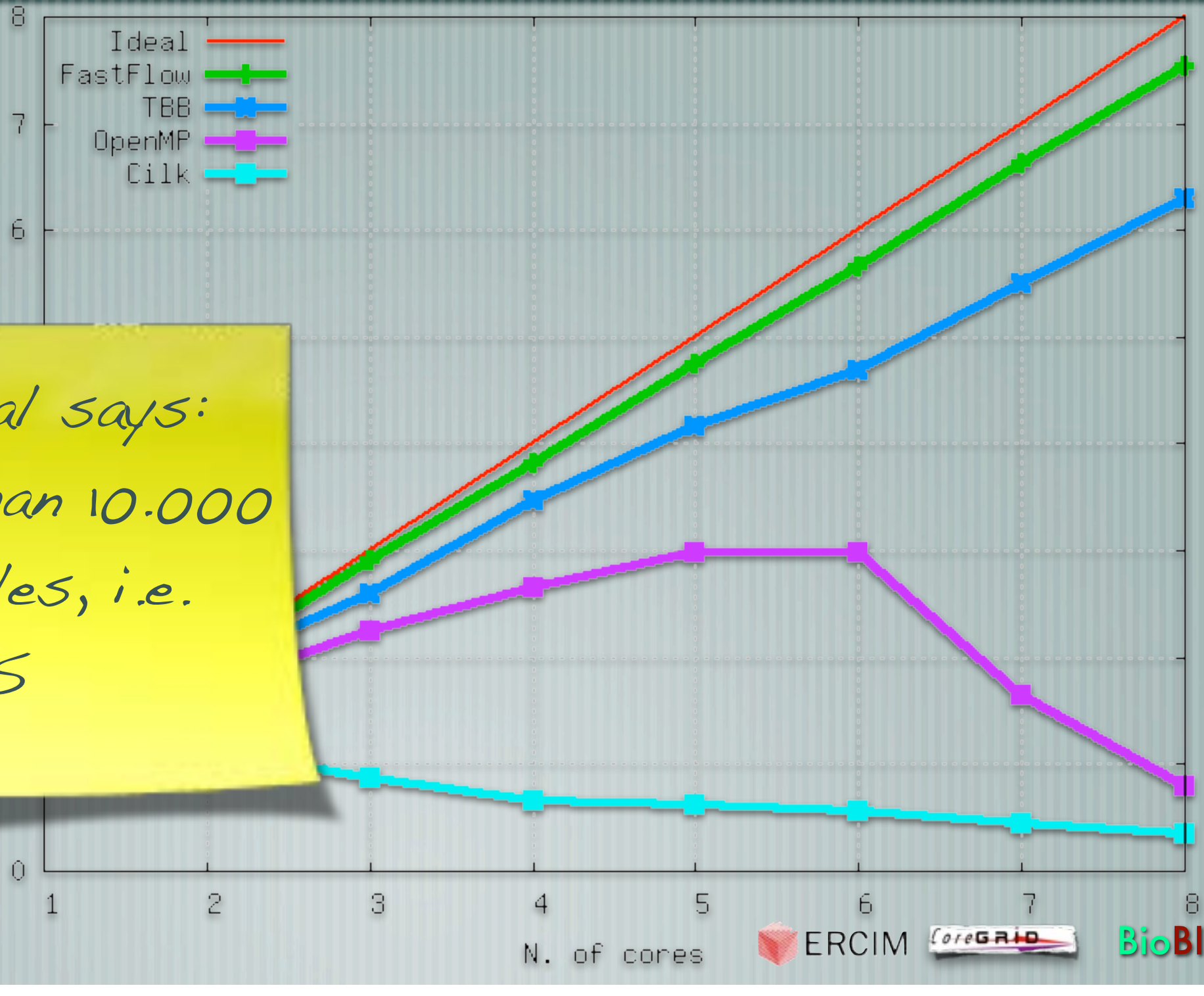
# Medium grain (5 µS workload)



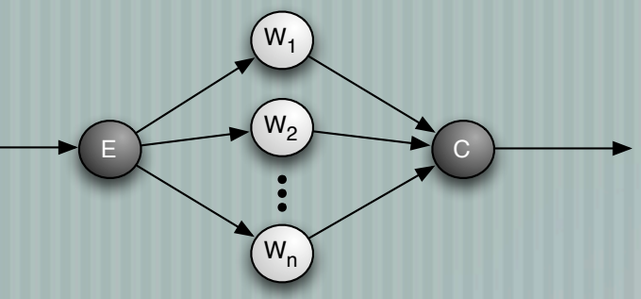
# Medium grain (5 µS workload)

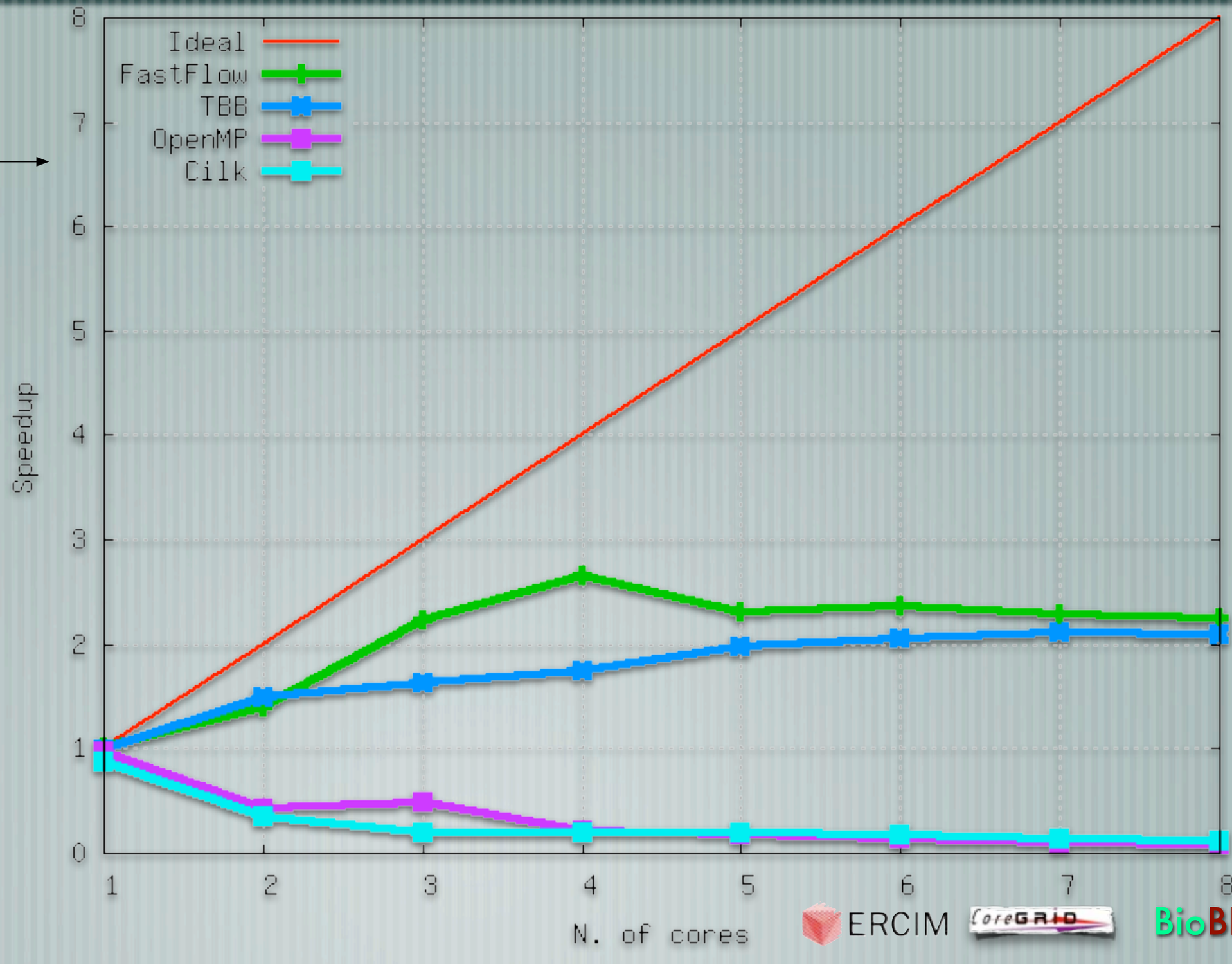


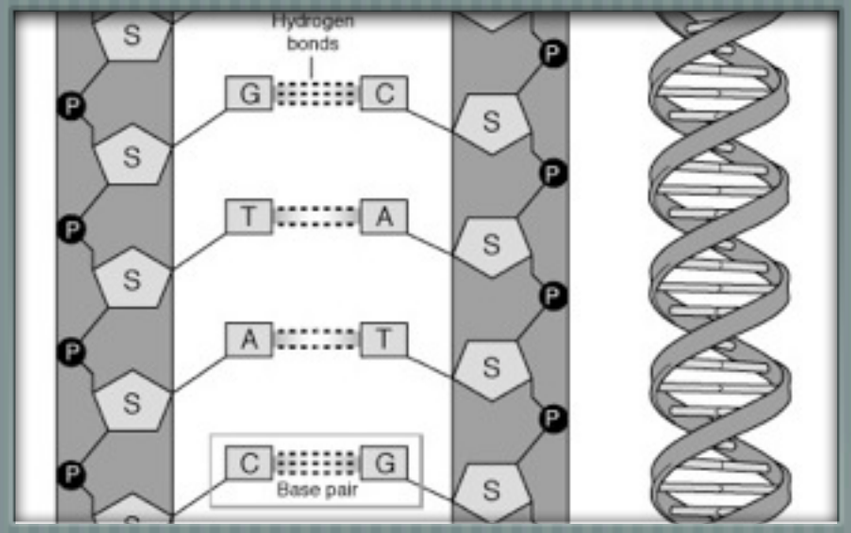



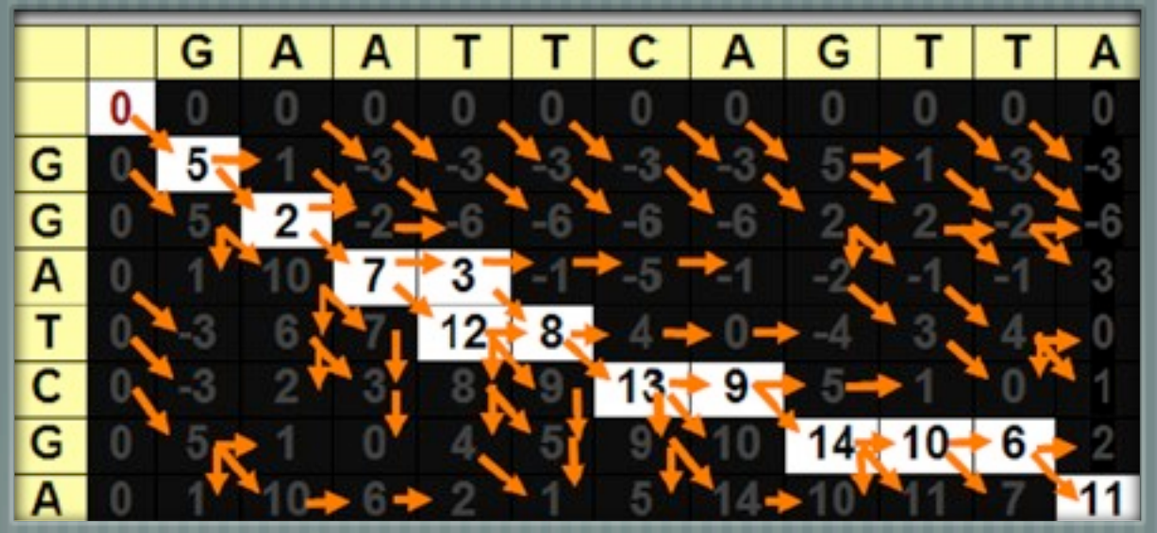

TBB manual says:
no less than 10.000
clock cycles, i.e.
about 5µS

# Fine grain (0.5 µS workload)






| | | G | A | A | T | T | C | A | G | T | T | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 1 | 0 | 0 |
| G | 0 | 5 | 2 | 0 | 0 | 0 | 0 | 0 | 5 | 2 | 0 | 0 |
| A | 0 | 1 | 10 | 7 | 3 | 0 | 0 | 5 | 1 | 2 | 0 | 5 |
| T | 0 | 0 | 6 | 7 | 12 | 8 | 4 | 1 | 2 | 6 | 8 | 4 |
| C | 0 | 0 | 2 | 3 | 8 | 9 | 13 | 9 | 5 | 2 | 4 | 5 |
| G | 0 | 5 | 1 | 0 | 4 | 5 | 9 | 10 | 14 | 10 | 6 | 2 |
| A | 0 | 1 | 10 | 6 | 2 | 1 | 5 | 14 | 10 | 11 | 7 | 11 |

```
GAATTCAG          GAATTCAG
|  |  ||  |       |  ||  |  |
GGA-TC-G          GCAT-C-G

GAATTC-A          GAATTC-A
|  |  ||  |       |  ||  |  |
GGA-TCGA          GCAT-CGA
```

# Smith-Waterman algorithm
# Local alignment - dynamic programming - O(nm)

# Fast Smith-Waterman

- Smith-Waterman algorithm
  - Local alignment
  - Time and space demanding O(mn), often replaced by approximated BLAST
  - Dynamic programming
  - Real-world application
    - It has been accelerated by using FPGA, GCPU (CUDA), SSE2/x86, IBM Cell
- Best software implementation up to now
  - SWPS3: evolution of Farrar's implementation
    - SSE3 + POSIX IPC

BioBits

A matrix $H$ is built as follows:

$$H(i,0) = 0,\ 0 \leq i \leq m$$
$$H(0,j) = 0,\ 0 \leq j \leq n$$
$$H(i,j) = \max \left\{ \begin{array}{rl} 0 & \\ H(i-1,j-1) + & w(a_i, b_j) \quad \text{Match/Mismatch} \\ H(i-1,j) + & w(a_i, -) \quad \text{Deletion} \\ H(i,j-1) + & w(-, b_j) \quad \text{Insertion} \end{array} \right\},\ 1 \leq i \leq m, 1 \leq j \leq n$$

Where:

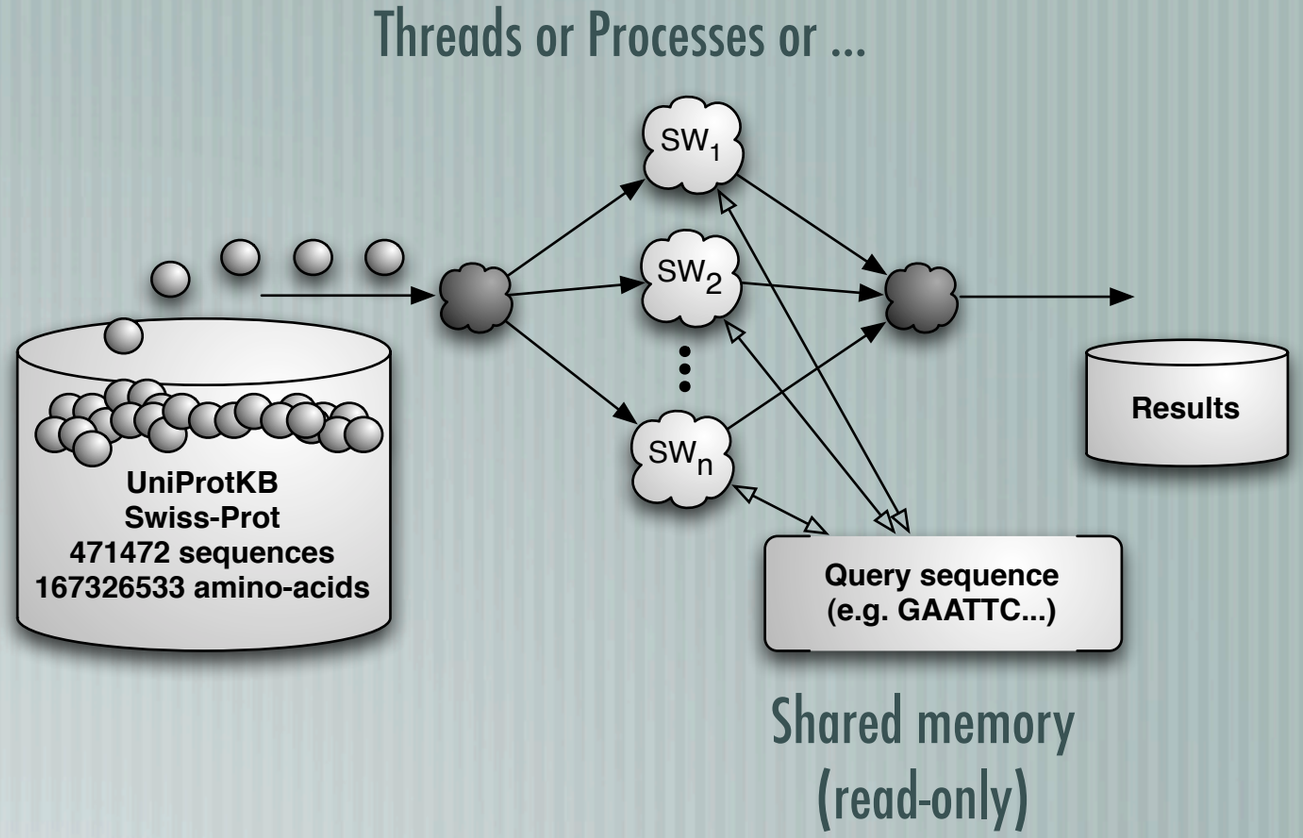
- a, b = Strings over the Alphabet $\Sigma$
- m = length(a)
- n = length(b)
- $H(i,j)$ - is the maximum Similarity-Score between the substring of a of length i, and the substring of b of length j
- $w(c,d),\ c,d \in \Sigma \cup \{'-'\}$, '-' is the gap-scoring scheme

- Substitution Matrix: describes the rate at which one character in a sequence changes to other character states over time
- Gap Penalty: describes the costs of gaps, possibly as function of gap length

## Experiment parameters
## Affine Gap Penalty: 10-2k, 5-2k, ...
## Substitution Matrix: BLOSUM50

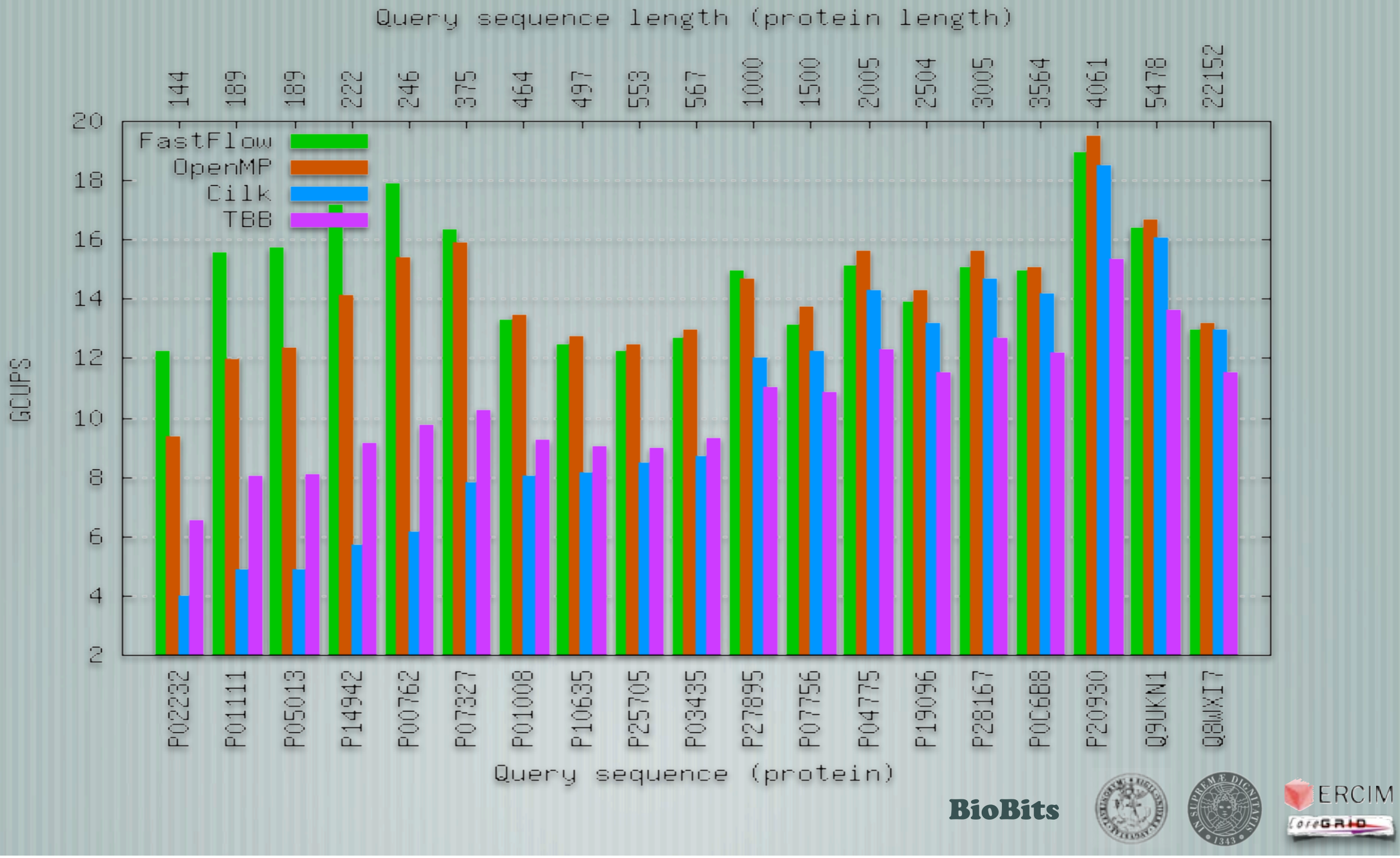
# Smith-Waterman testbed

- Each query sequence (protein) is aligned against the whole protein DB
  - E.g. Compare unknown sequence against a DB of known sequences
- SWPS3 implementation exploits POSIX processes and pipes
  - Faster than POSIX threads + locks
  - http://people.inf.ethz.ch/sadam/swps3/

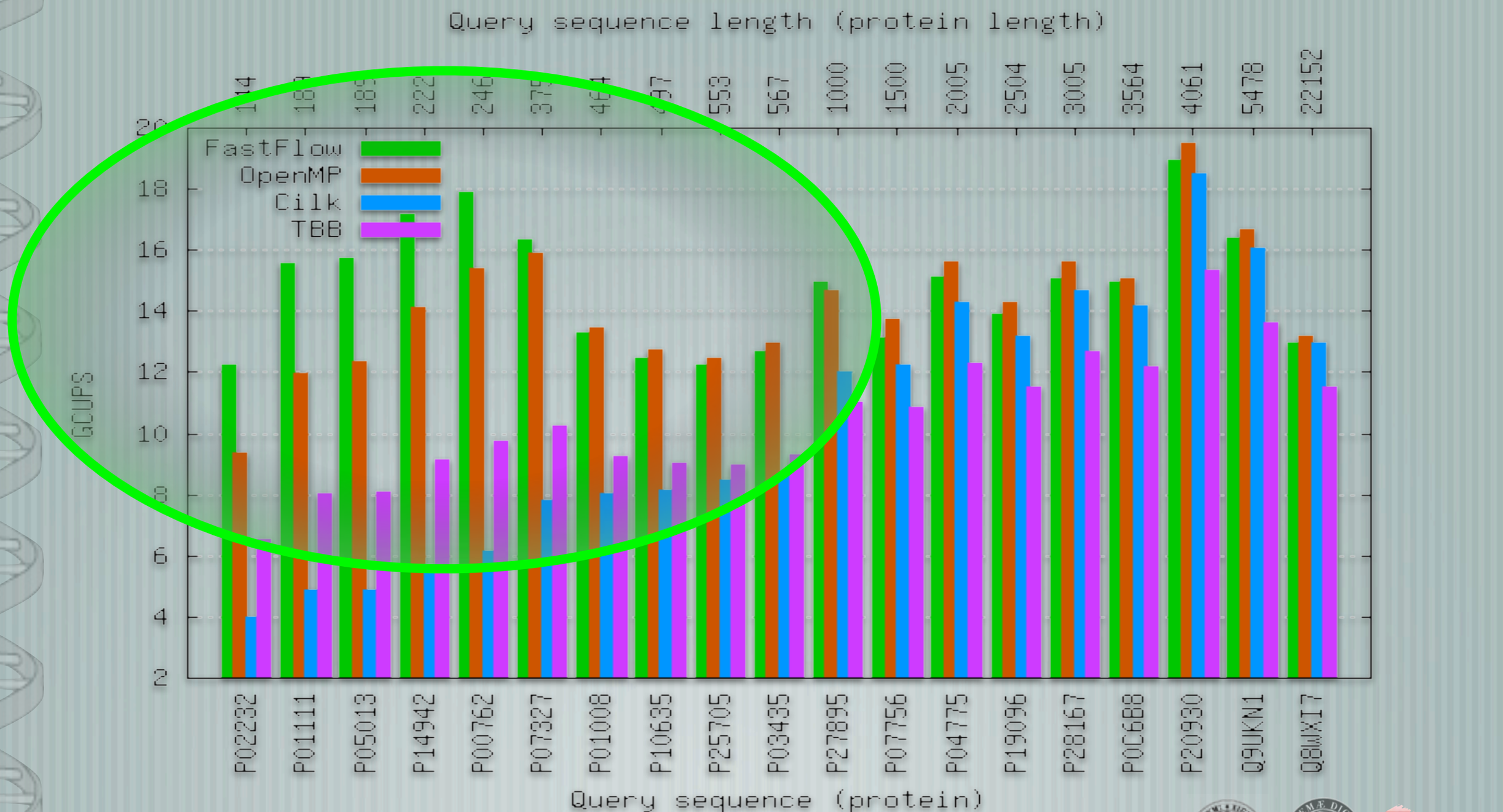



BioBits

# Smith Waterman (10-2k gap penalty)



# Smith Waterman (10-2k gap penalty)

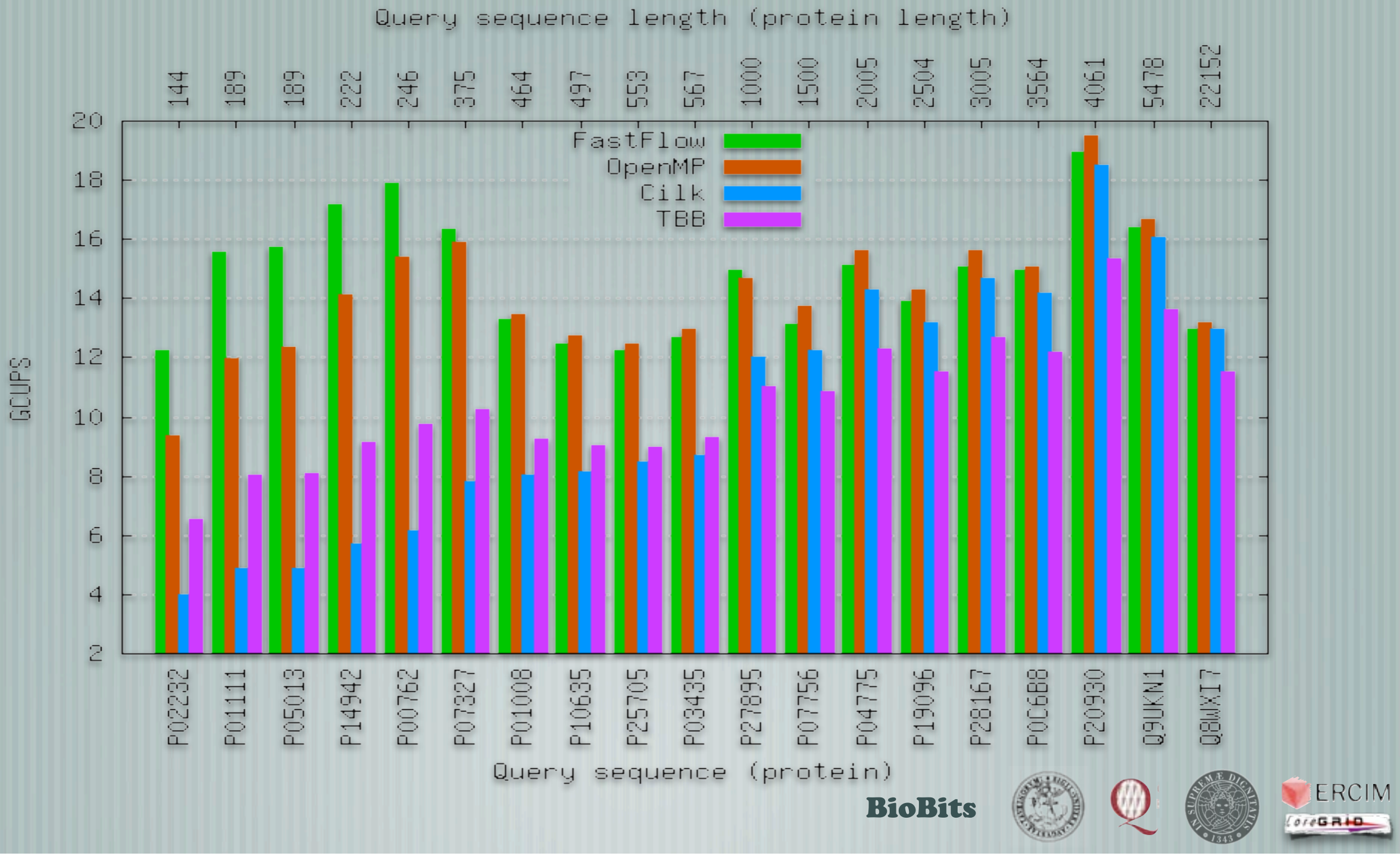



BioBits

ERCIM

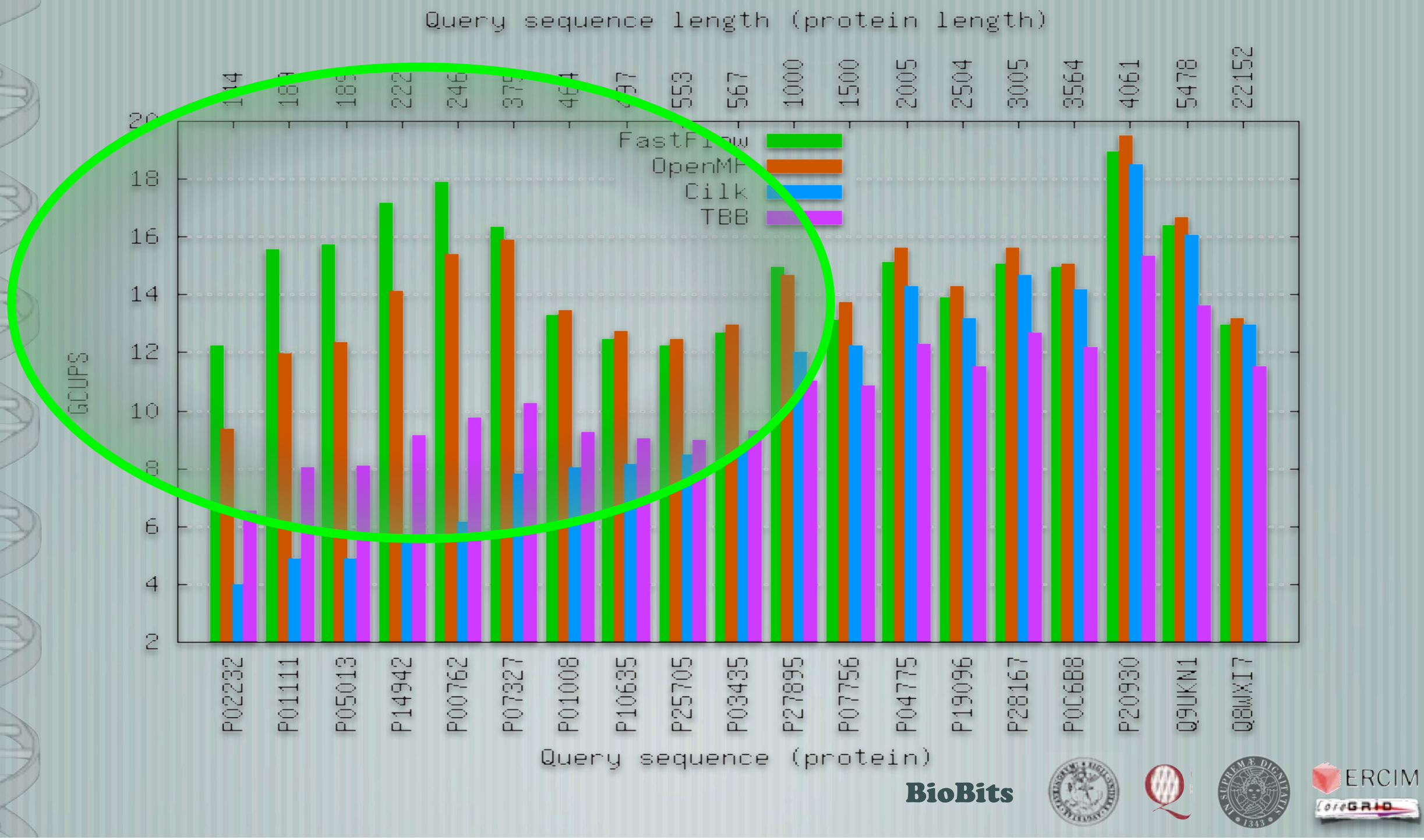
LoreGRID

# Smith Waterman (5-2k gap penalty)




BioBits

# Smith Waterman (5-2k gap penalty)




BioBits

# Is FastFlow easy to use? Yes!

```
1  // Original code
2  #define N 1024
3  long A[N][N],B[N][N],C[N][N];
4  int main() {
5    // < init A,B,C>
6
7    for(int i=0;i<N;++i) {
8      for(int j=0;j<N;++j) {
9
10       int _C=0;
11       for(int k=0;k<N;++k)
12         _C += A[i][k]*B[k][j];
13       C[i][j]=_C;
14
15     }
16   }
17 }
```

```
20 // FastFlow accelerated code
21 #define N 1024
22 long A[N][N],B[N][N],C[N][N];
23 int main() {
24   // < init A,B,C>
25
26   ff::ff_farm<> farm(true /* accel */);
27   std::vector<ff::ff_node *> w;
28   for(int i=0;i<PAR_DEGREE;++i)
29     w.push_back(new Worker);
30   farm.add_workers(w);
31   farm.run_then_freeze();
32
33   for (int i=0;i<N;i++) {
34     for(int j=0;j<N;++j) {
35       task_t * task = new task_t(i,j);
36       farm.offload(task);
37     }
38   }
39   farm.offload((void *)ff::FF_EOS);
40   farm.wait();  // Here join
41 }
42
43 // Includes
44 struct task_t {
45   task_t(int i,int j):i(i),j(j) {}
46   int i; int j;};
47
48 class Worker: public ff::ff_node {
49 public: // Offload target service
50   void * svc(void *task) {
51     task_t * t = (task_t *)task;
52     int _C=0;
53     for(int k=0;k<N;++k)
54       _C += A[t->i][k]*B[k][t->j];
55     C[t->i][t->j] = _C;
56     delete t;
57     return GO_ON;
58     }
59 };
```

# Conclusions

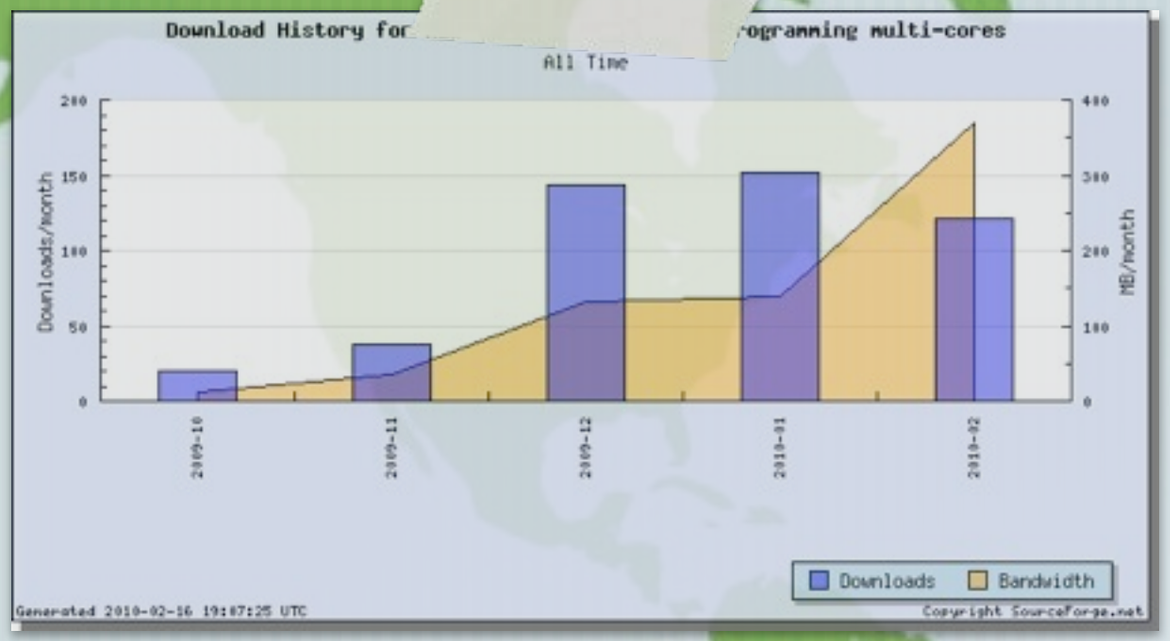
- FastFlow efficiently supports streaming applications on commodity SCM (e.g. Intel core architecture)
  - More efficiently than POSIX (standard or CAS lock), Cilk, OpenMP, TBB
- Smith Waterman algorithm with FastFlow
  - Obtained from SWPS3 by syntactically substituting read and write on POSIX pipes with fastflow push and FastFlow pop an push
    - In turn, POSIX pipes are faster than POSIX threads + locks in this case
  - Scores twice the speed of best known parallel implementation (SWPS3) on the same hardware (Intel 2 x Quad-core 2.5 GHz)

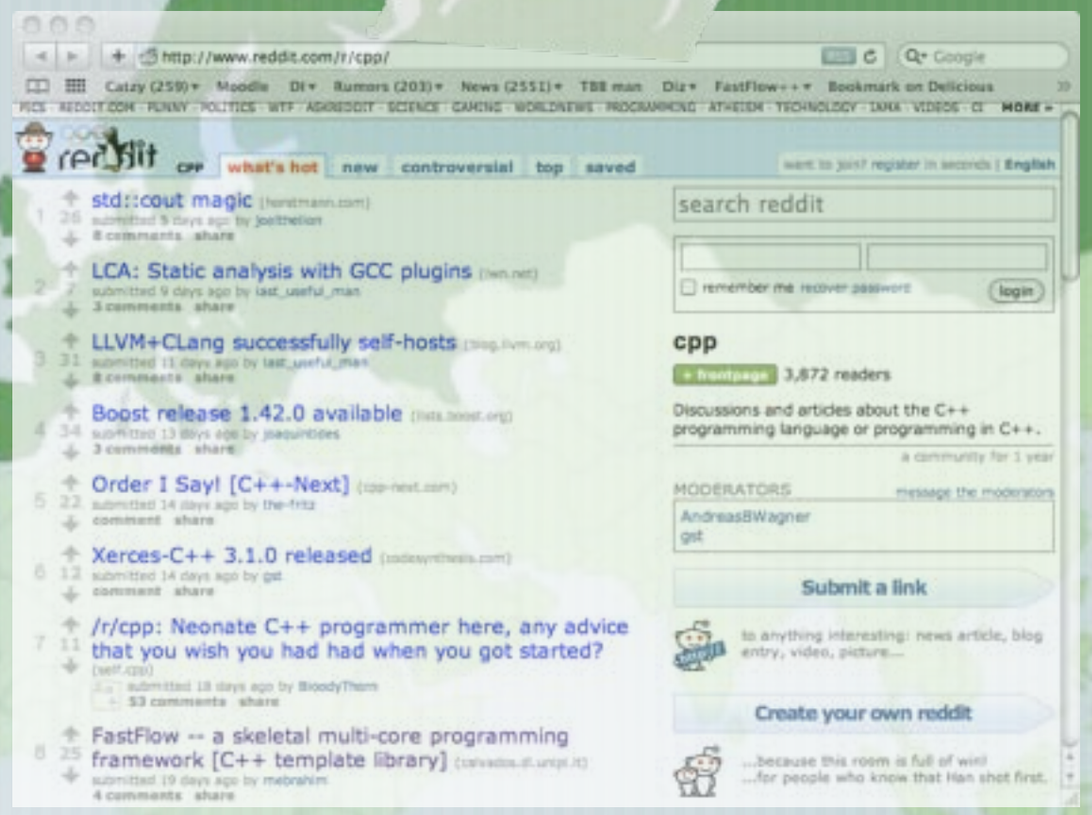
BioBits

# DON'T BELIEVE?

# CHECK IT OUT!

# THANK YOU! QUESTIONS?




Started in Nov ‘09
at today over 600 downloads

In the top ten of reddit.com/c++
from several weeks

**http://sourceforge.net/projects/mc-fastflow/**