Enjeux de la programmation sur les architectures parallèles d'aujourd'hui

École IN2P3 2016 :
Parallélisme sur matériel hétérogène

Laurent Plagne (EDF R&D)
laurent.plagne@edf.fr

23 Mai 2016

EDF
CHANGER L'ENERGIE ENSEMBLE

Sommaire

1. Évolution du matériel depuis 15 ans

1998-2014

- 1998 : Cray T3E 256 nœuds.

1998-2014

- 1998 : Cray T3E 256 nœuds.
- 2014 : Cluster Porthos 516 nœuds.

1998-2014

- 1998 : Cray T3E 256 nœuds.
- 2014 : Cluster Porthos 516 nœuds.

Quelques centaines de nœuds interconnectés...

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds.

Quelques centaines de nœuds interconnectés...

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds. 601 TFlops (DP)

Quelques centaines de nœuds interconnectés...

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds. 601 TFlops (DP)

Quelques centaines de nœuds interconnectés...
... la puissance des super-calculateurs a été multipliée par

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds. 601 TFlops (DP)

Quelques centaines de nœuds interconnectés...
... la puissance des super-calculateurs a été multipliée par

- $\times 3130$!

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds. 601 TFlops (DP)
- 2014 : ARM Mali-T628MP6 (Samsung Galaxy S5) 142 GFlops ;)

Quelques centaines de nœuds interconnectés...
... la puissance des super-calculateurs a été multipliée par

- $\times 3130$!

1998-2014

- 1998 : Cray T3E 256 nœuds. 192 GFlops (DP)
- 2014 : Cluster Porthos 516 nœuds. 601 TFlops (DP)
- 2014 : ARM Mali-T628MP6 (Samsung Galaxy S5) 142 GFlops ;)

Quelques centaines de nœuds interconnectés...
... la puissance des super-calculateurs a été multipliée par

- $\times 3130$!
- Les performances des noeuds ont explosé...

1998-2014 : Évolution des nœuds

Year	Computer
1998	T3E
2014	Porthos

1998-2014 : Évolution des nœuds

Year	Computer	Proc
1998	T3E	Dec α
2014	Porthos	Intel Xeon

1998-2014 : Évolution des nœuds

Year	Computer	Proc	GFlops
1998	T3E	Dec α	0.750
2014	Porthos	Intel Xeon	500
			$\times 1333$

1998-2014 : Évolution des nœuds

Year	Computer	Proc	GFlops	GHz
1998	T3E	Dec α	0.750	0.375
2014	Porthos	Intel Xeon	500	2.6
			$\times$1333	$\times$7

- $\times$7 frequency (the free lunch is over (2003-))

1998-2014 : Évolution des nœuds

Year	Computer	Proc	GFlops	GHz	Cores
1998	T3E	Dec α	0.750	0.375	1
2014	Porthos	Intel Xeon	500	2.6	2×14
			$\times 1333$	$\times 7$	$\times 28$

- $\times 7$ frequency (the free lunch is over (2003-))
- $7 \times 28 = 196$ (// SMP)

1998-2014 : Évolution des nœuds

Year	Computer	Proc	GFlops	GHz	Cores	SIMD
1998	T3E	Dec α	0.750	0.375	1	
2014	Porthos	Intel Xeon	500	2.6	2×14	AVX.2
			$\times 1333$	$\times 7$	$\times 28$	$\times 4$

- $\times 7$ frequency (the free lunch is over (2003-))
- $7 \times 28 = 196$ (// SMP)
- $7 \times 28 \times 4 = 784$ (// + vectorisation)

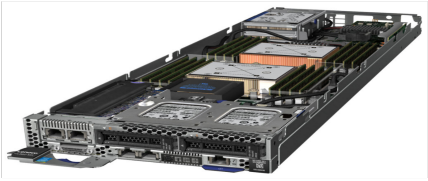
1998-2014 : Évolution des nœuds

Year	Computer	Proc	GFlops	GHz	Cores	SIMD
1998	T3E	Dec α	0.750	0.375	1	
2014	Porthos	Intel Xeon	500	2.6	2×14	AVX.2
			$\times 1333$	$\times 7$	$\times 28$	$\times 4$

- $\times 7$ frequency (the free lunch is over (2003-))
- $7 \times 28 = 196$ (// SMP)
- $7 \times 28 \times 4 = 784$ (// + vectorisation)
- $7 \times 28 \times 4 \times 2 = 1568$ (// SMP + vectorisation +FMA)

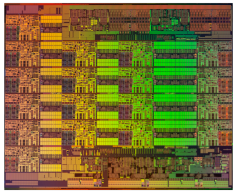
2014 : Cluster Porthos

Cluster Porthos EDF



575 Noeuds

2 procs/noeuds



14 coeurs/proc

Element Width	Vector Width	Instruction	Launch
BYTE	128	PSHUFB	SSE4
DWORD	256	VPERMD VPERMPS	AVX2 New!
QWORD	256	VPERMQ VPERMPD	AVX2 New!

8 float AVX2/coeur

1998-2014 : The memory Wall

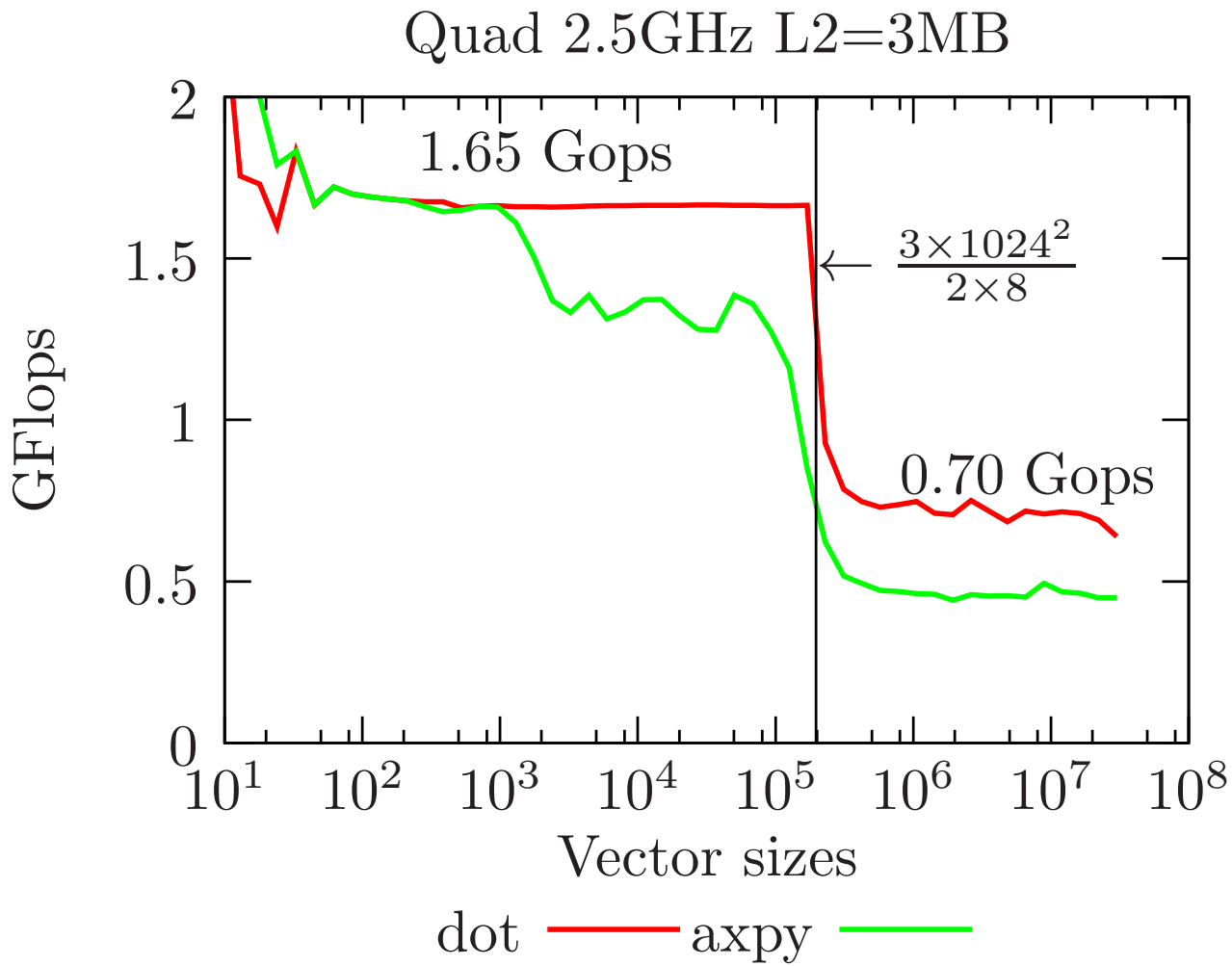
Year	Proc	GFlops	GHz	Cores	SIMD	GB/s
1998	Dec α	0.750	0.375	1		0.6
2014	Intel Xeon	500	2.6	2×14	AVX.2	68
		$\times 1333$	$\times 7$	$\times 28$	$\times 4/8$	$\times 100$

1998-2014 : The memory Wall

Year	Proc	GFlops	GHz	Cores	SIMD	GB/s
1998	Dec α	0.750	0.375	1		0.6
2014	Intel Xeon	500	2.6	2×14	AVX.2	68
		$\times 1333$	$\times 7$	$\times 28$	$\times 4/8$	$\times 100$

Le nombre de mouvements mémoire et l'intensité arithmétique sont les paramètres dominants pour la performance de la plupart des codes...

Nombre de cycles pour *,+ et -



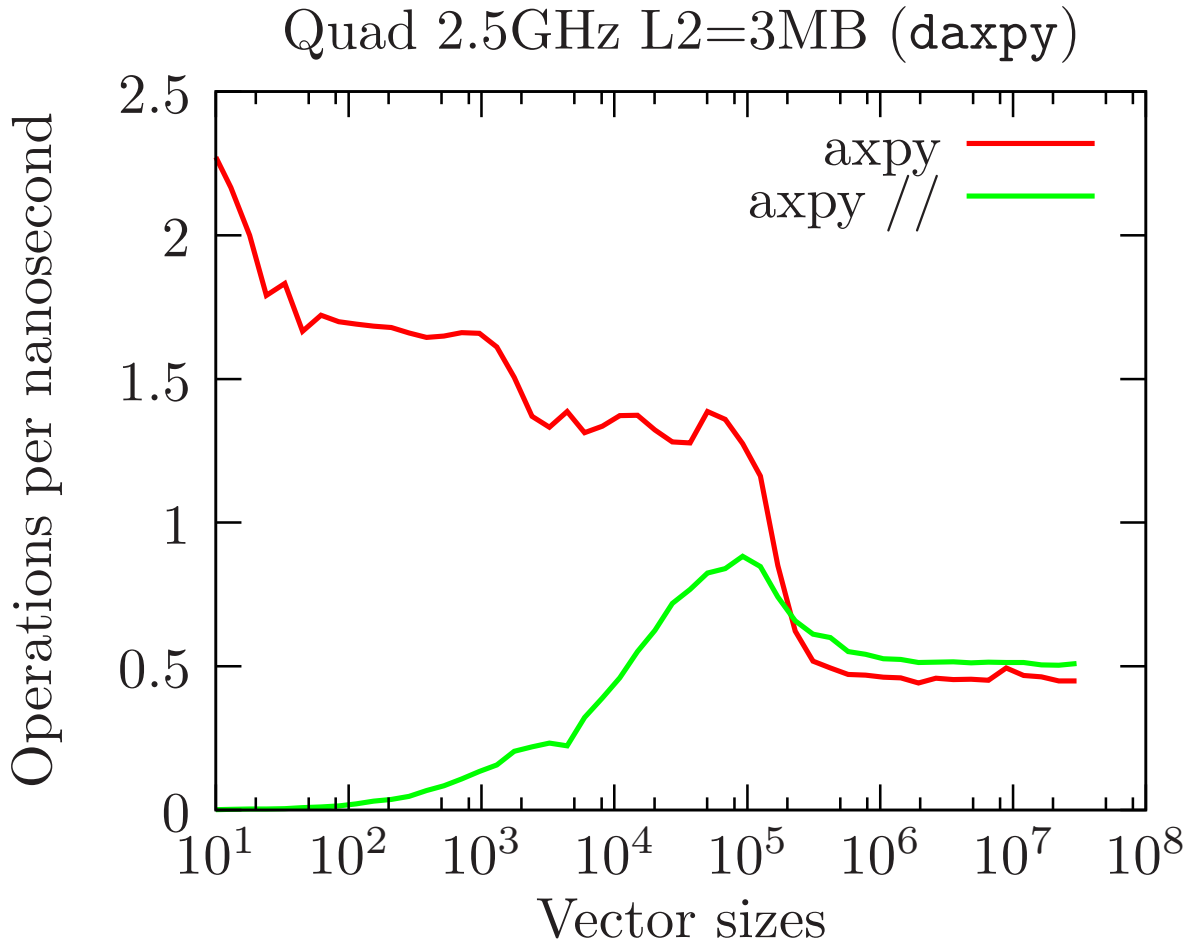

```
for (int i=0;i<N;i++
    Y[i]+=coef*X[i];
}
```

$$P_{axpy} = \frac{2\times\beta}{t_{axpy}(s)}$$

$$\beta = \frac{N}{10^9}$$

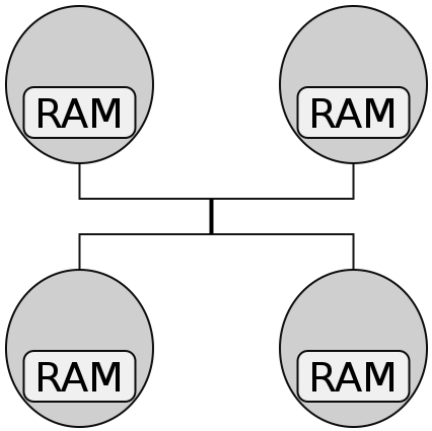
$$\frac{2.5}{1.65} = 1.5 \text{ cycles/flop}$$

Parallélisation `axpy`



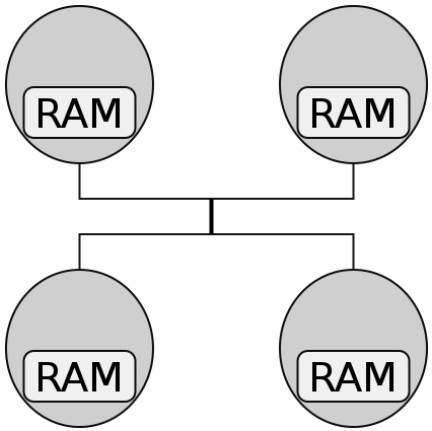
Architectures

Cluster



Architectures

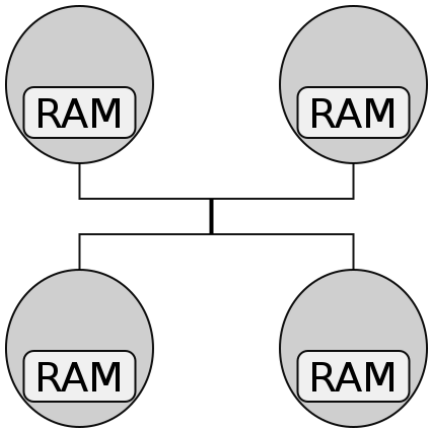
Cluster



Mémoire proche

Architectures

Cluster

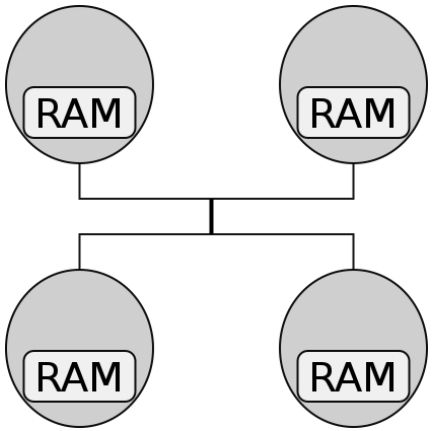


Mémoire proche

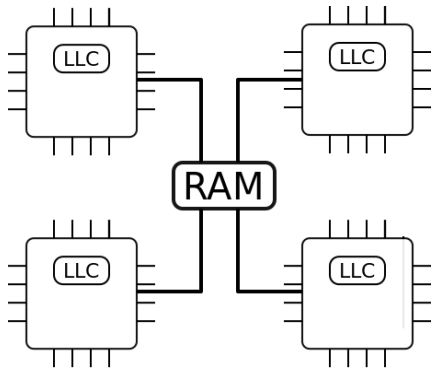
Mémoire distante

Architectures

Cluster



Node

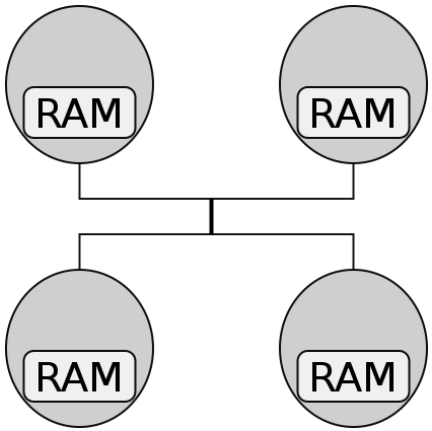


Mémoire proche

Mémoire distante

Architectures

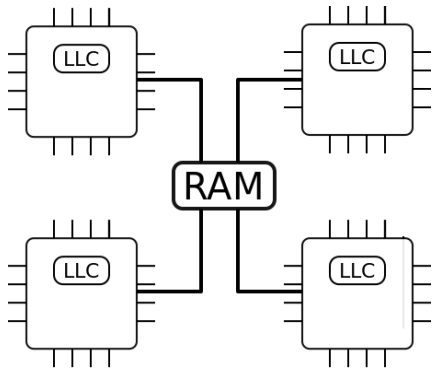
Cluster



Mémoire proche

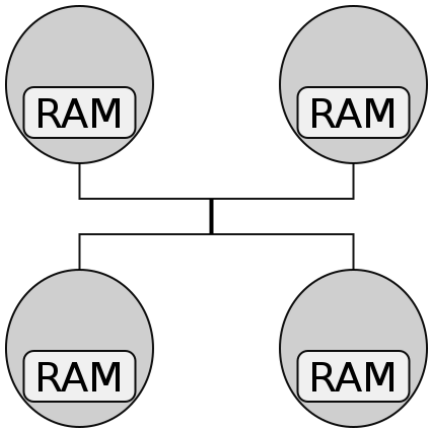
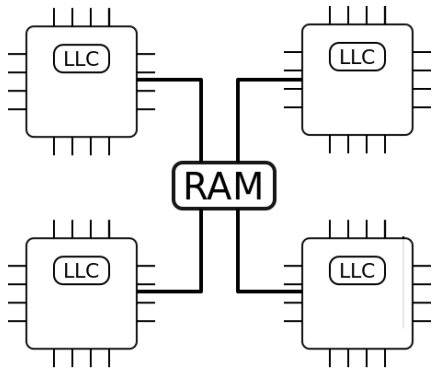
Mémoire distante

Node

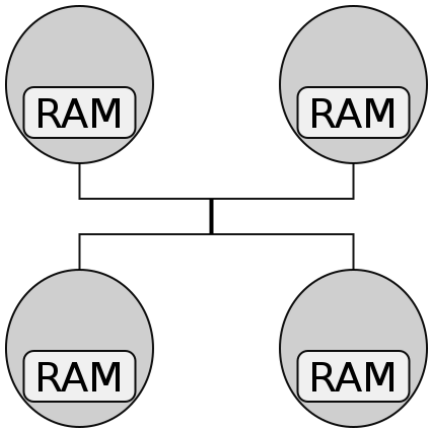
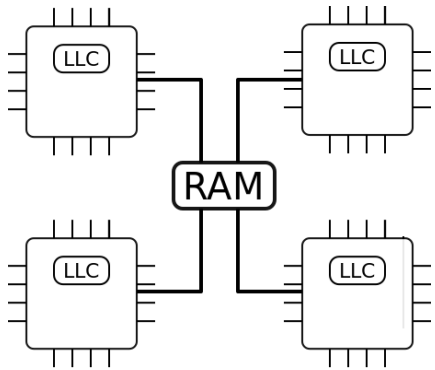


Mémoire proche

Architectures

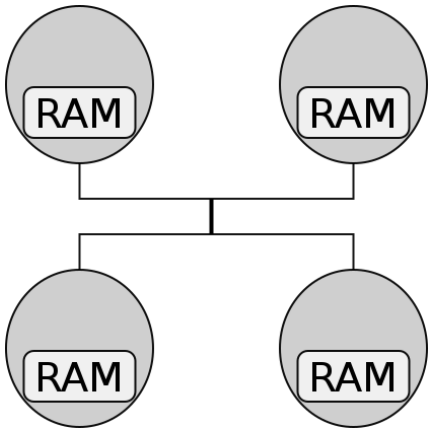
Cluster	Node
	
Mémoire proche	Mémoire proche
Mémoire distante	Mémoire distante

Architectures

Cluster	Node	CPU
		
Mémoire proche	Mémoire proche	
Mémoire distante	Mémoire distante	

Architectures

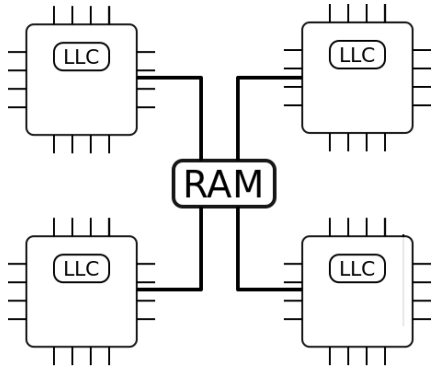
Cluster



Mémoire proche

Mémoire distante

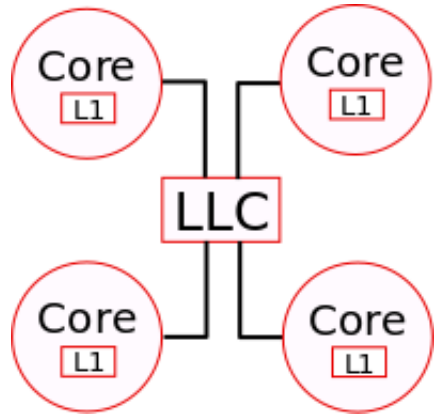
Node



Mémoire proche

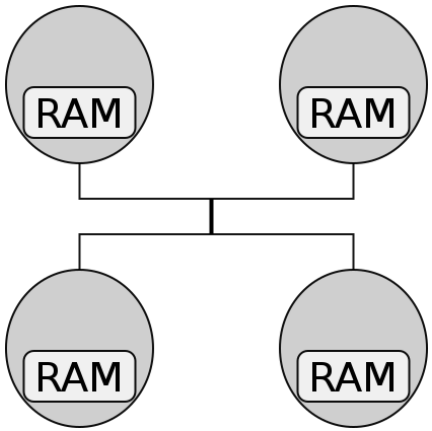
Mémoire distante

CPU



Architectures

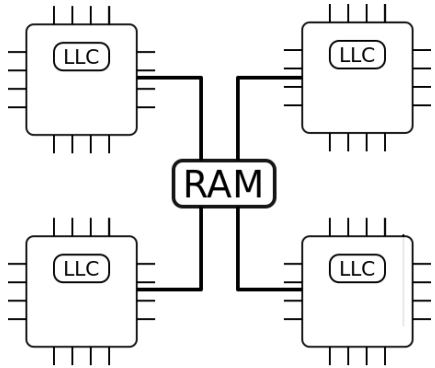
Cluster



Mémoire proche

Mémoire distante

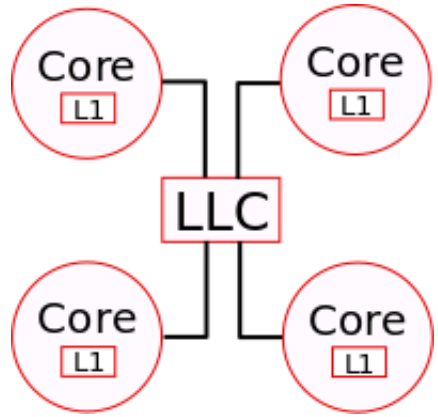
Node



Mémoire proche

Mémoire distante

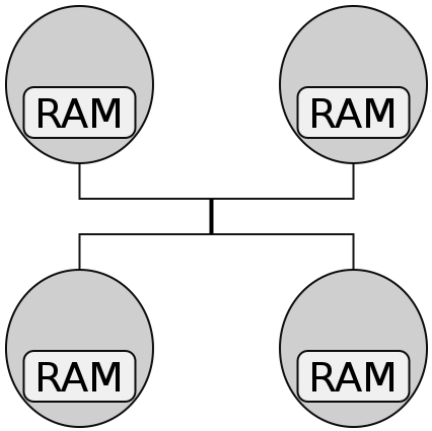
CPU



Mémoire proche

Architectures

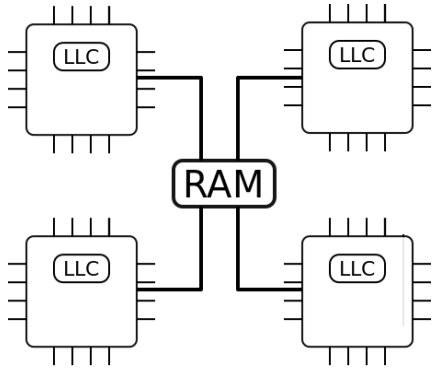
Cluster



Mémoire proche

Mémoire distante

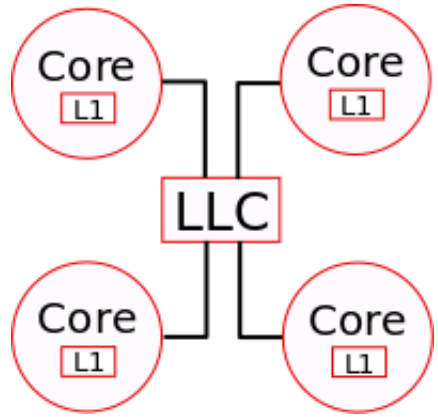
Node



Mémoire proche

Mémoire distante

CPU

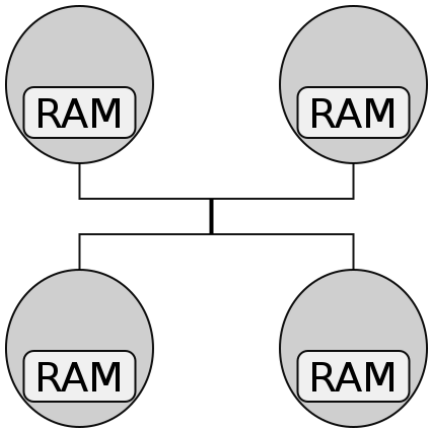


Mémoire proche

Mémoire distante

Architectures

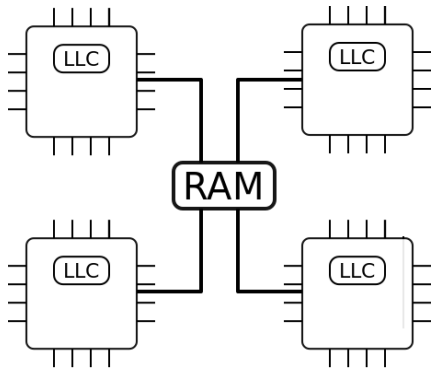
Cluster



Mémoire proche

Mémoire distante

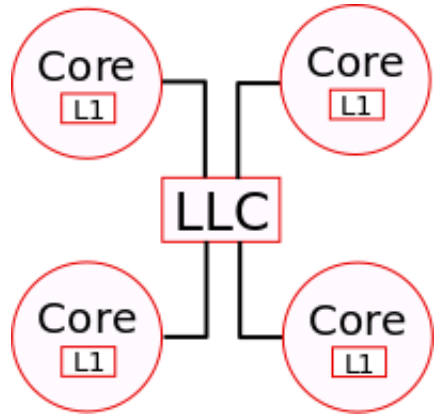
Node



Mémoire proche

Mémoire distante

CPU



Mémoire proche

Mémoire distante

Architecture récursive...

Architectures Hiérarchiques

Cluster

Architectures Hiérarchiques

Cluster	100,1000 nodes

Architectures Hiérarchiques

Cluster Node	100,1000 nodes

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus
CPU	

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus
CPU	2,16 cores

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus
CPU	2,16 cores
Core	

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus
CPU	2,16 cores
Core	simd units

Architectures Hiérarchiques

Cluster	100,1000 nodes
Node	1,2,4,8 cpus
CPU	2,16 cores
Core	simd units
SIMD	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	
CPU	2,16 cores	
Core	simd units	
SIMD	128,512 bits ops	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	
Core	simd units	
SIMD	128,512 bits ops	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	LLC
Core	simd units	
SIMD	128,512 bits ops	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	LLC
Core	simd units	L1/L2
SIMD	128,512 bits ops	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	LLC
Core	simd units	L1/L2
SIMD	128,512 bits ops	Registers

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.	1-100 TB
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.	1-100 TB
Node	1,2,4,8 cpus	Shared Mem.	8-1000 GB
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.	1-100 TB
Node	1,2,4,8 cpus	Shared Mem.	8-1000 GB
CPU	2,16 cores	LLC	8 MB
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.	1-100 TB
Node	1,2,4,8 cpus	Shared Mem.	8-1000 GB
CPU	2,16 cores	LLC	8 MB
Core	simd units	L1/L2	32/256 KB
SIMD	128,512 bits ops	Registers	

Architectures Hiérarchiques

Cluster	100,1000 nodes	Distr. Mem.	1-100 TB
Node	1,2,4,8 cpus	Shared Mem.	8-1000 GB
CPU	2,16 cores	LLC	8 MB
Core	simd units	L1/L2	32/256 KB
SIMD	128,512 bits ops	Registers	$\simeq$400B

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	LLC
Core	simd units	L1/L2
SIMD	128,512 bits ops	Registers

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	1 cycle

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	5/12 cycles
SIMD	128,512 bits ops	Registers	1 cycle

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	43 cycles
Core	simd units	L1/L2	5/12 cycles
SIMD	128,512 bits ops	Registers	1 cycle

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	240 cycles
CPU	2,16 cores	LLC	43 cycles
Core	simd units	L1/L2	5/12 cycles
SIMD	128,512 bits ops	Registers	1 cycle

Architectures Hierarchiques : Latences

Cluster	100,1000 nodes	Distr. Mem.	4000,10000 cycles
Node	1,2,4,8 cpus	Shared Mem.	240 cycles
CPU	2,16 cores	LLC	43 cycles
Core	simd units	L1/L2	5/12 cycles
SIMD	128,512 bits ops	Registers	1 cycle

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.
Node	1,2,4,8 cpus	Shared Mem.
CPU	2,16 cores	LLC
Core	simd units	L1/L2
SIMD	128,512 bits ops	Registers

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	5 TB/s

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	1 TB/s
SIMD	128,512 bits ops	Registers	5 TB/s

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	500 GB/s
Core	simd units	L1/L2	1 TB/s
SIMD	128,512 bits ops	Registers	5 TB/s

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	20-100 GB/S
CPU	2,16 cores	LLC	500 GB/s
Core	simd units	L1/L2	1 TB/s
SIMD	128,512 bits ops	Registers	5 TB/s

Architectures : Débit mémoire

Cluster	100,1000 nodes	Distr. Mem.	40 GB/s
Node	1,2,4,8 cpus	Shared Mem.	20-100 GB/S
CPU	2,16 cores	LLC	500 GB/s
Core	simd units	L1/L2	1 TB/s
SIMD	128,512 bits ops	Registers	5 TB/s

Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs : $\times 1000$

Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs : $\times 1000$
- Accélération des nœuds : $\times 1000$

Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs : $\times 1000$
- Accélération des nœuds : $\times 1000$
- Accélération par la fréquence : $\times 10$

Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs :$\times 1000$
- Accélération des nœuds :$\times 1000$
- Accélération par la fréquence :$\times 10$
- Accélération par le // SMP :$\times 10$

Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs :$\times 1000$
- Accélération des nœuds :$\times 1000$
- Accélération par la fréquence :$\times 10$
- Accélération par le // SMP :$\times 10$
- Accélération par la vectorisation :$\times 10$

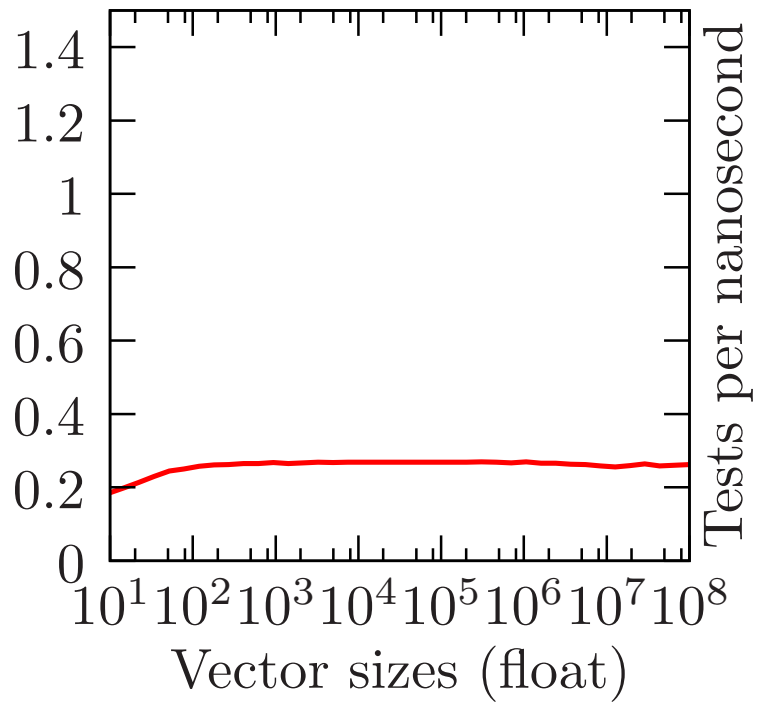
Résumé

Depuis 15 ans :

- Accélération des super-ordinateurs :$\times 1000$
- Accélération des nœuds :$\times 1000$
- Accélération par la fréquence :$\times 10$
- Accélération par le // SMP :$\times 10$
- Accélération par la vectorisation :$\times 10$
- Augmentation de la bande passante :$\times 100$

Test performance(s)

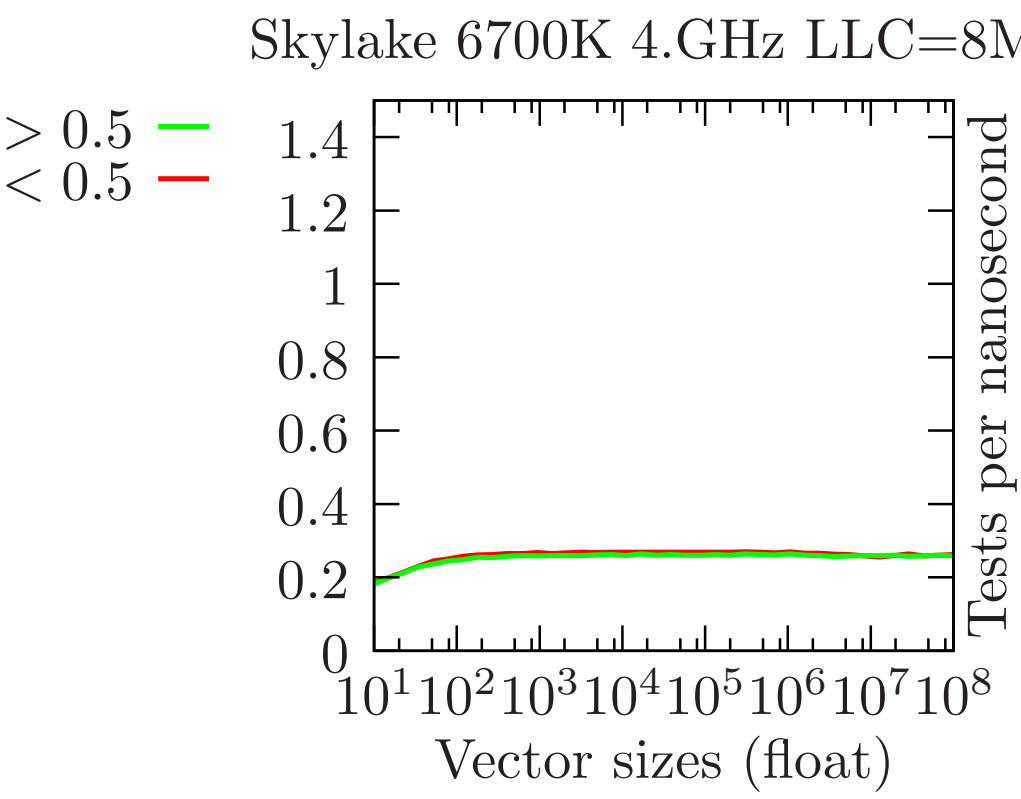
< 0.5 —




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

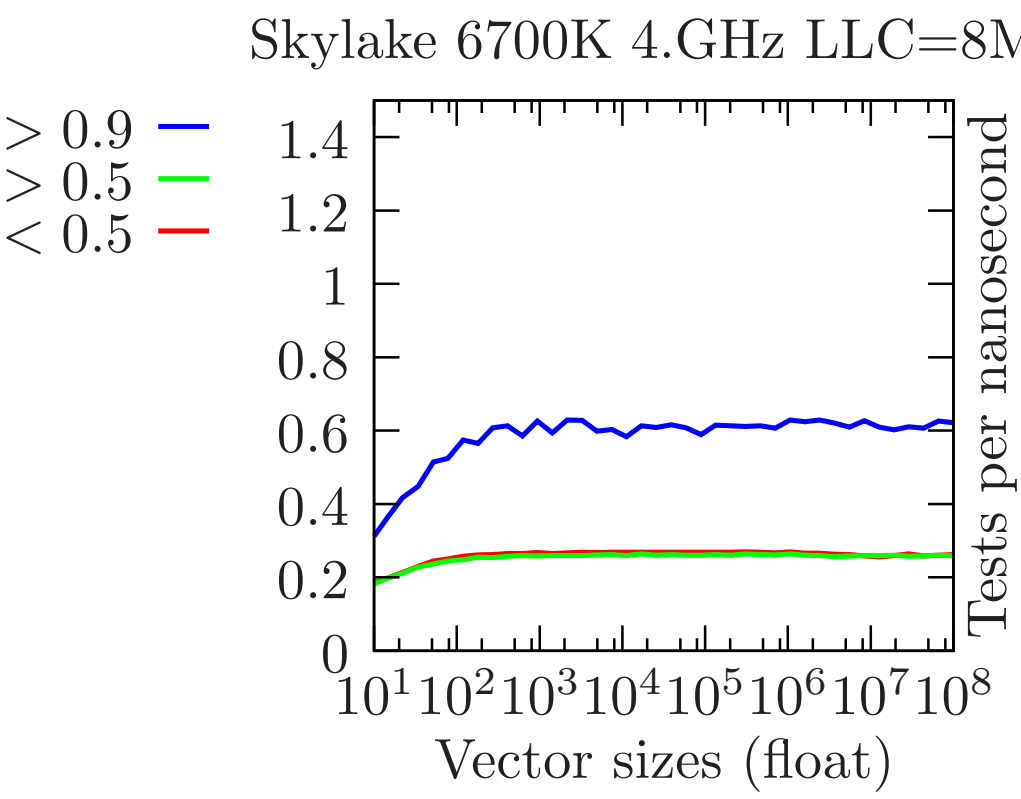
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

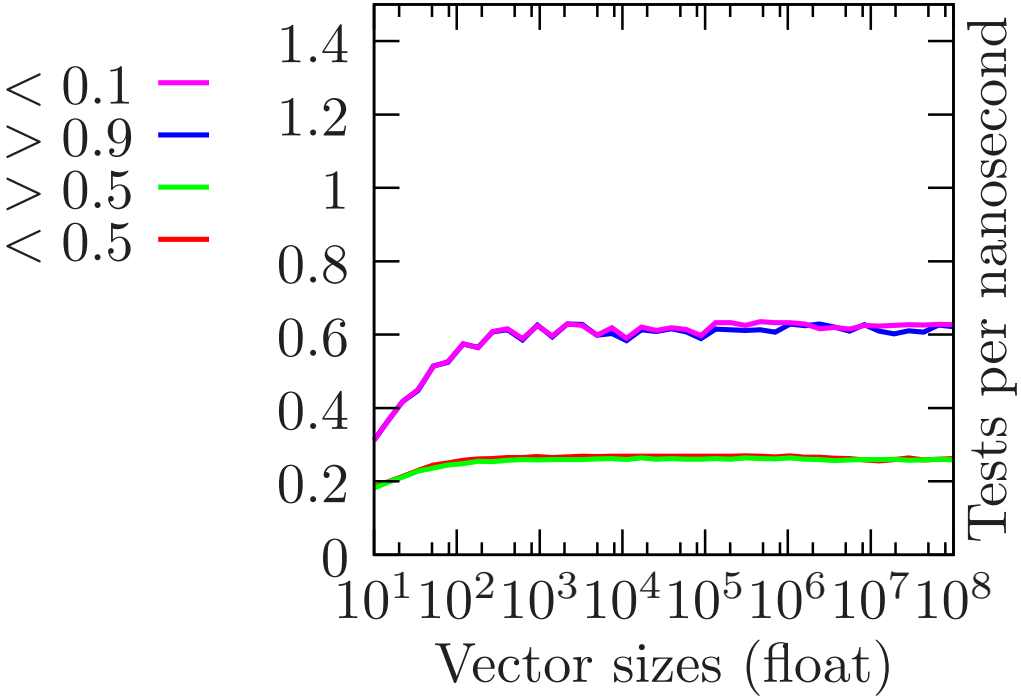
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

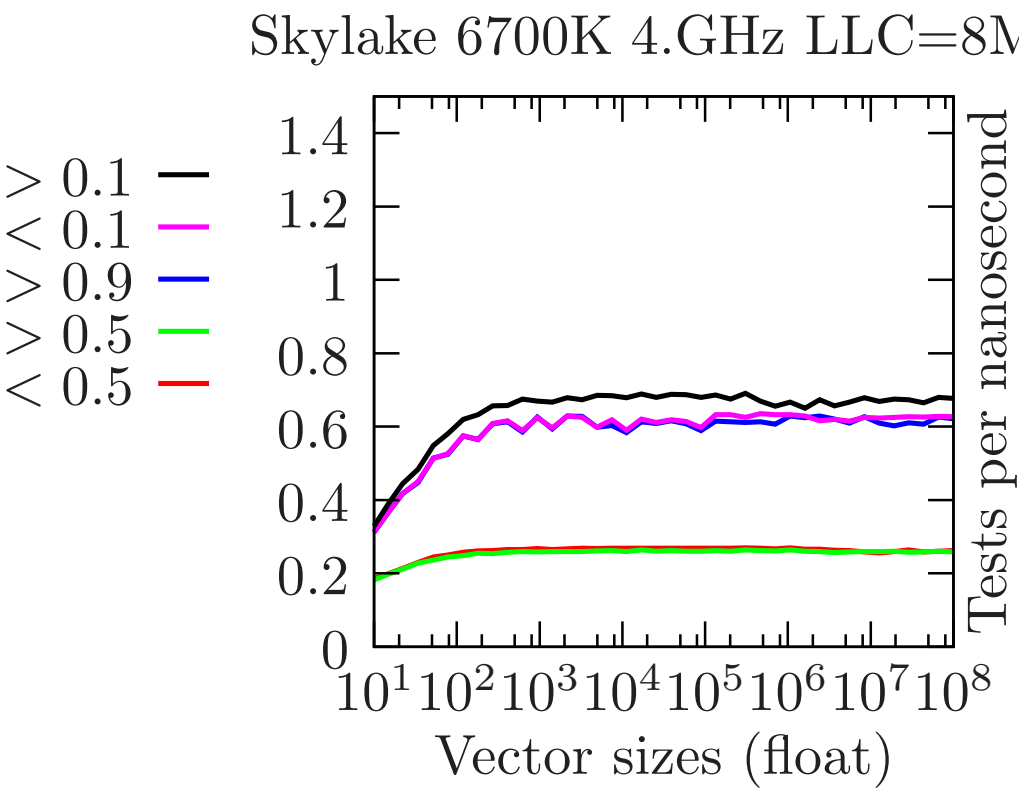
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

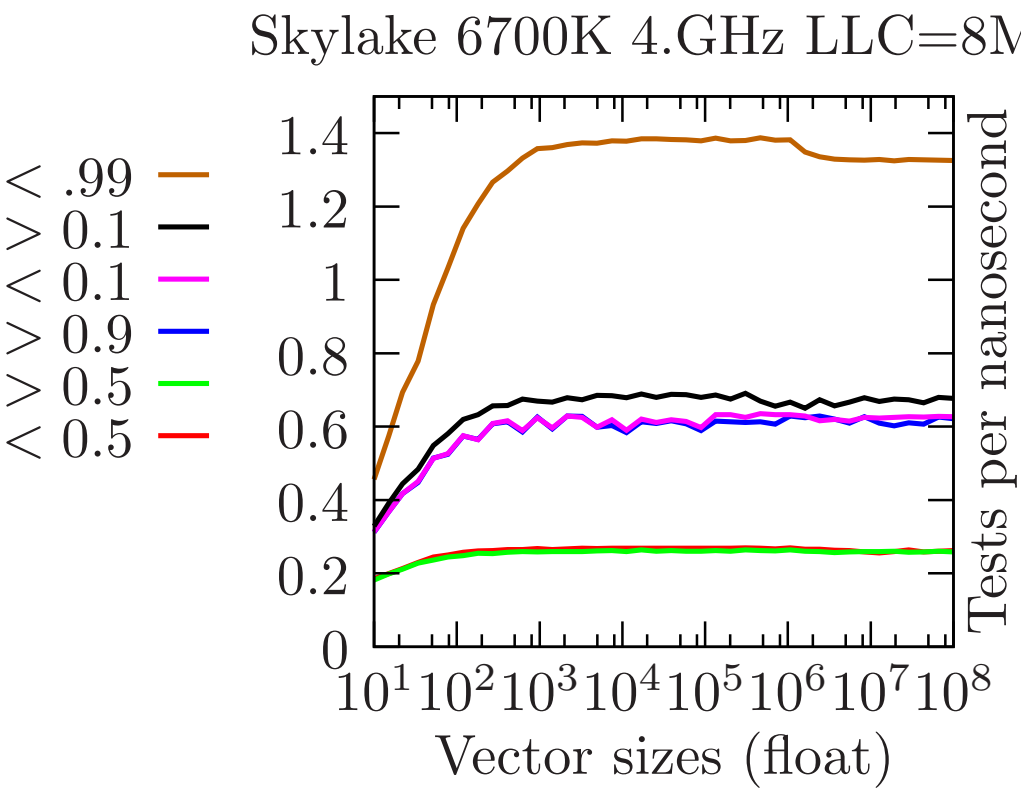
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

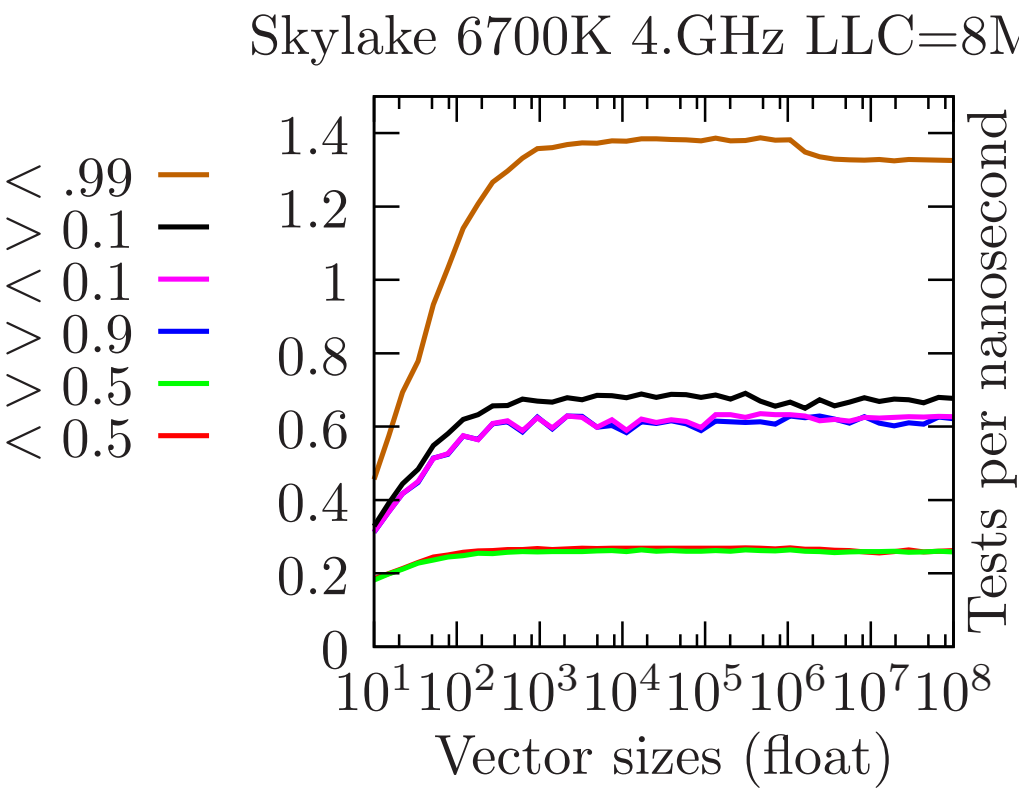
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3

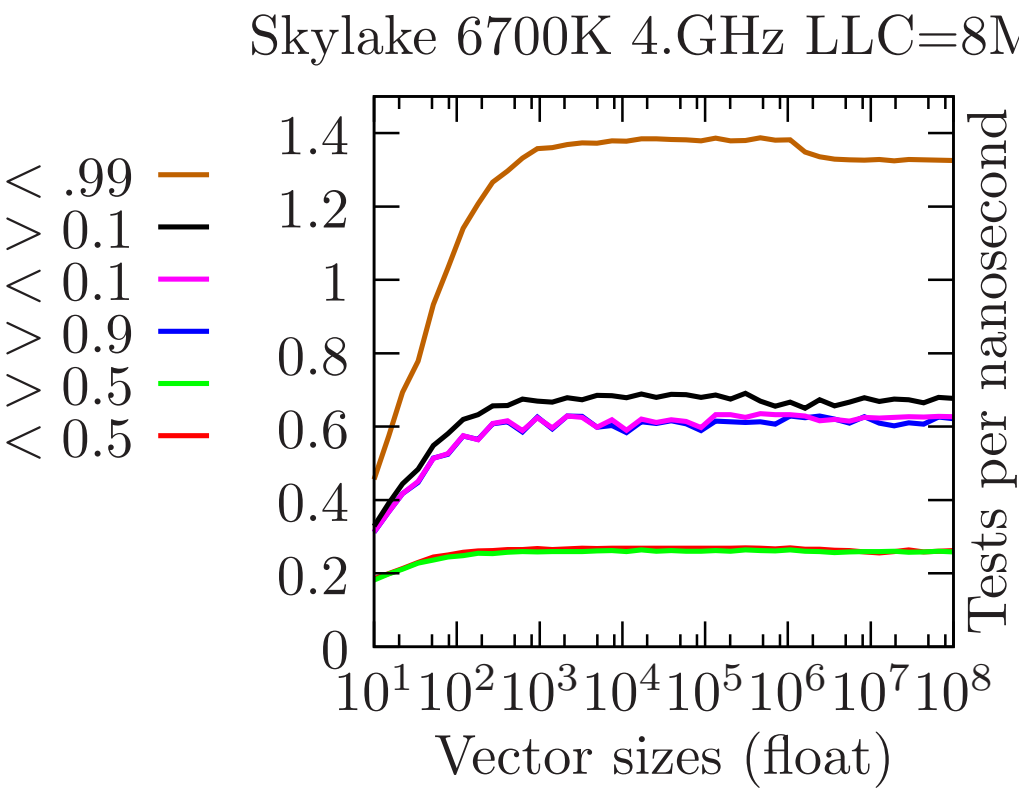
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3
- Un test : 16 cycles

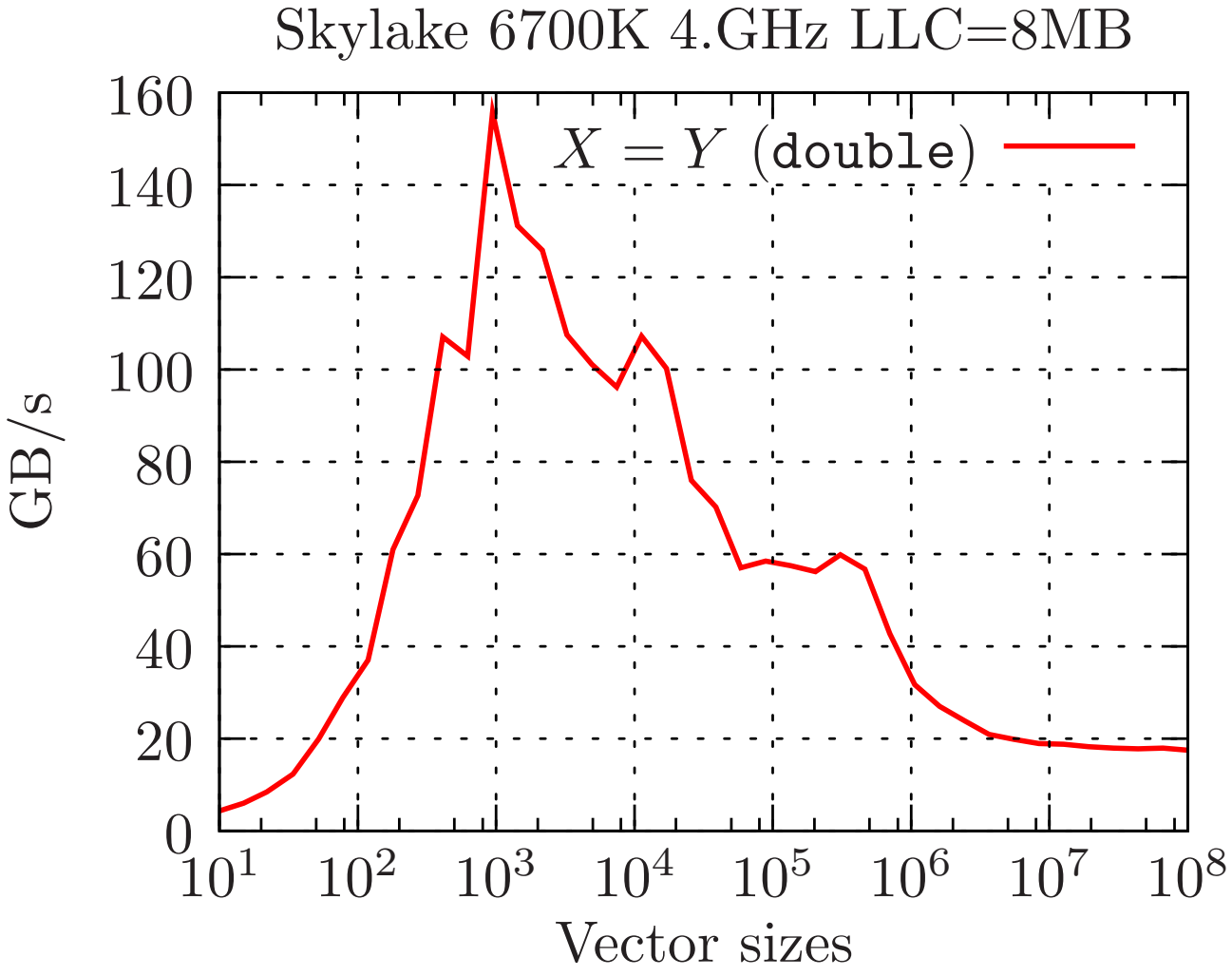
Test performance(s)




```
for (int i=0;i<N;i++){
  if (X[i]<ts){
    X[i]+=0.5;
  }
  else{
     X[i]+=0.2;
  }
}
```

- g++6.1 -O3
- Un test : 16 cycles
- Un test prev : 2-3 cycles

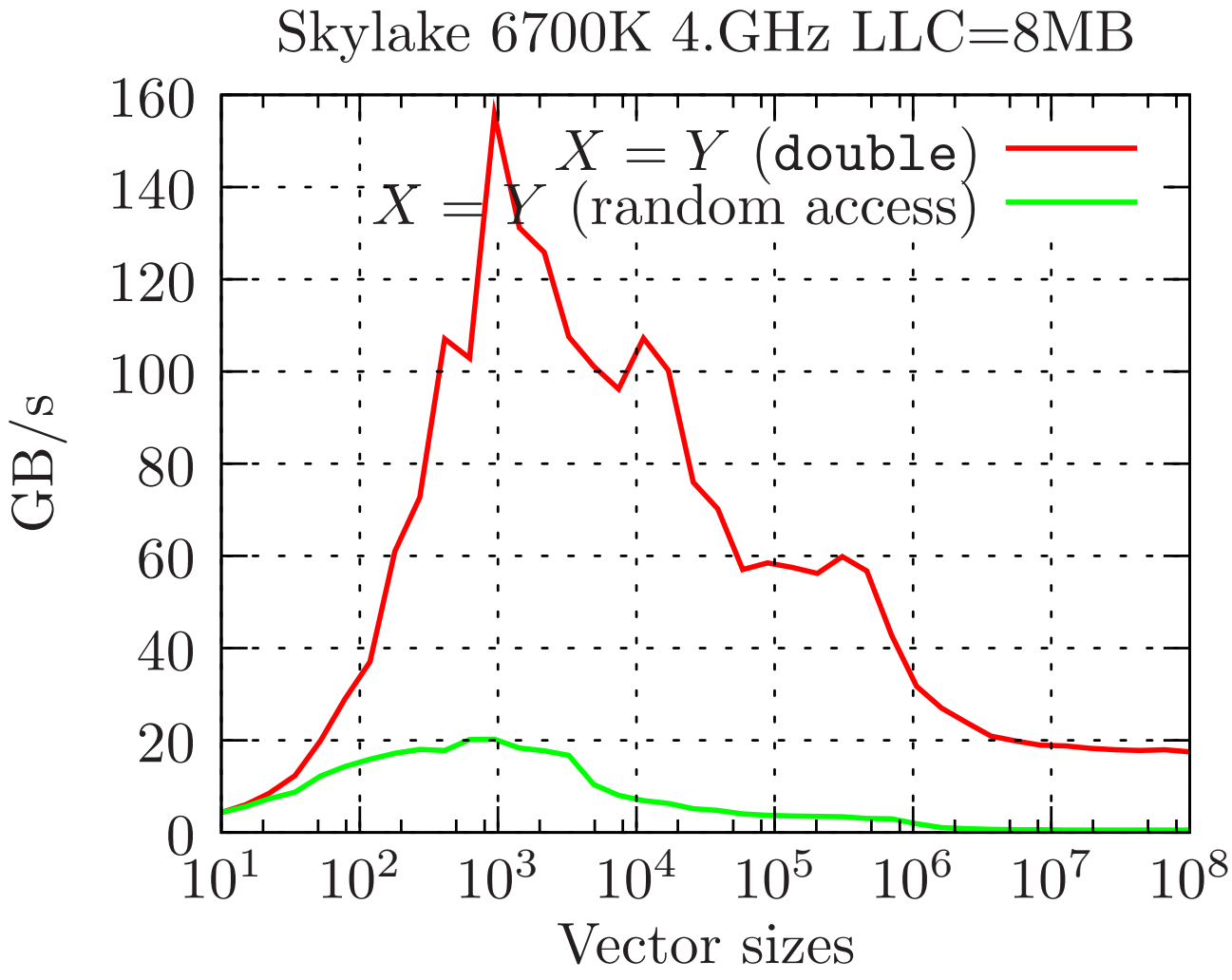
Bande passante (copy)




```
for (int i=0;i<N;i++
    X[i]=Y[i];
}
```

$$B_c = \frac{2 \times \alpha}{t_{copy}(s)}$$

Bande passante RAM



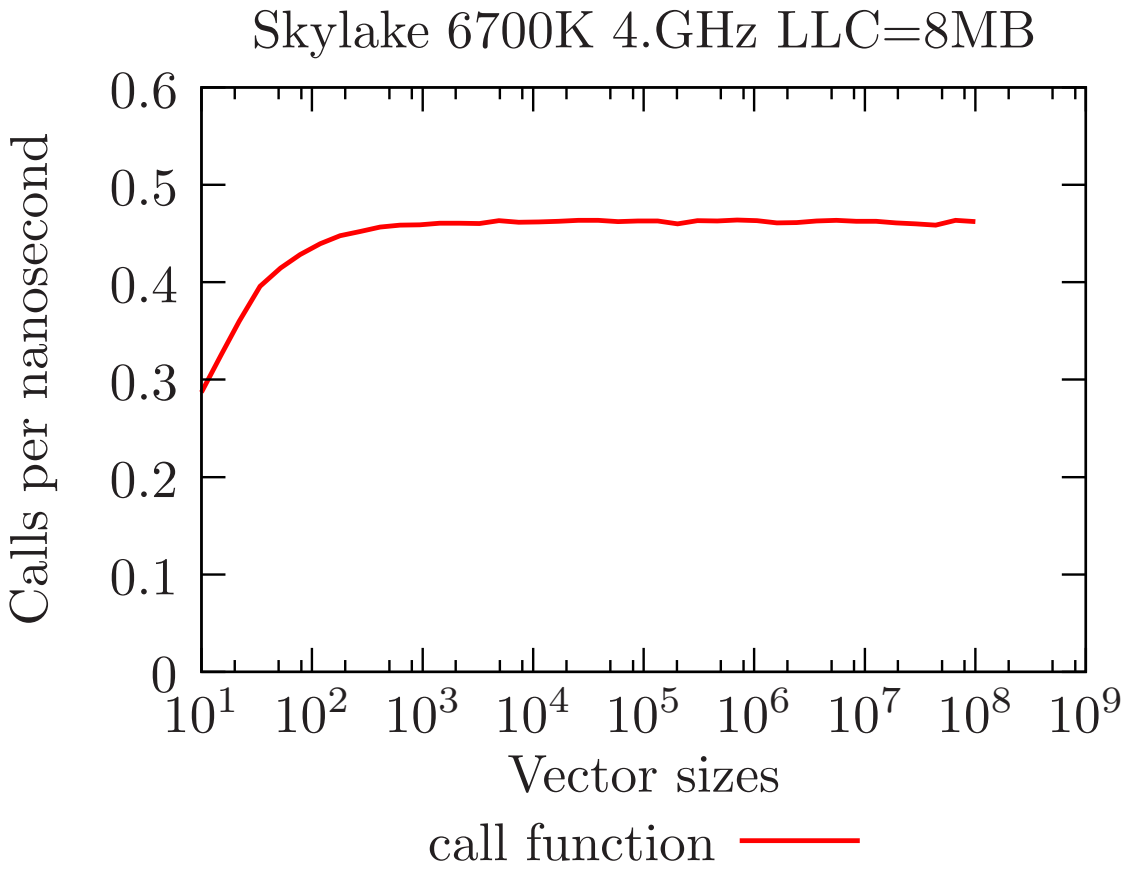

```
for (int i=0;i<N;i++
    X[i]=Y[i];
}
```

$$B_c = \frac{2 \times \alpha}{t_{copy}(s)}$$

```
for (int j=0;j<N;j++
   X[i[j]]=Y[i[j]];
}
```

$B_r \simeq 20$ GB/s :
$\frac{8*4}{20} = 1.6$ cycles/`double`
$B_{ram} \simeq 0.5$ GB/s :
$\frac{8*4}{0.5} = 64$ cycles/`double`

Nombre de cycles pour un `call`




```
double Xi=1.0;
for (int i=0;i<N;i++
    addOne(Xi);
}
```

$\frac{4}{0.45} = 9$ cycles/call

I6700K 4 GHz, LLC=8Mo

Action	Nombre de Cycles
`*`,`+`,`-`	1 (0.25 AVX2)
accès ordonné à la RAM	1.5
$\frac{1}{x}$	2
`if` (prévisible)	< 3
appel de fonction	9
`if`	15
$\sqrt{x}$	20 (3 fast-math)
$\exp x$	43
accès aléatoire à la RAM	64

Accélérateurs : 2014/2015

Proc	GFlops float	GFlops double	GHz	Cores	SIMD	GB/s
Intel Xeon	1000	500	2.6	2×14	AVX.2	68
NVidia	5632	176	1.16			224
GTX 980	$\times 5$					$\times 3.3$
Intel Xeon	2416	1208	1.2	61	AVX512	352
Phi (KNC)	$\times 2.5$	$\times 2.5$				$\times 5$

Accélérateurs : 2014/2015

Proc	GFlops float	GFlops double	GHz	Cores	SIMD	GB/s
Intel Xeon	1000	500	2.6	2×14	AVX.2	68
NVidia	5632	176	1.16			224
GTX 980	$\times 5$					$\times 3.3$
Intel Xeon	2416	1208	1.2	61	AVX512	352
Phi (KNC)	$\times 2.5$	$\times 2.5$				$\times 5$

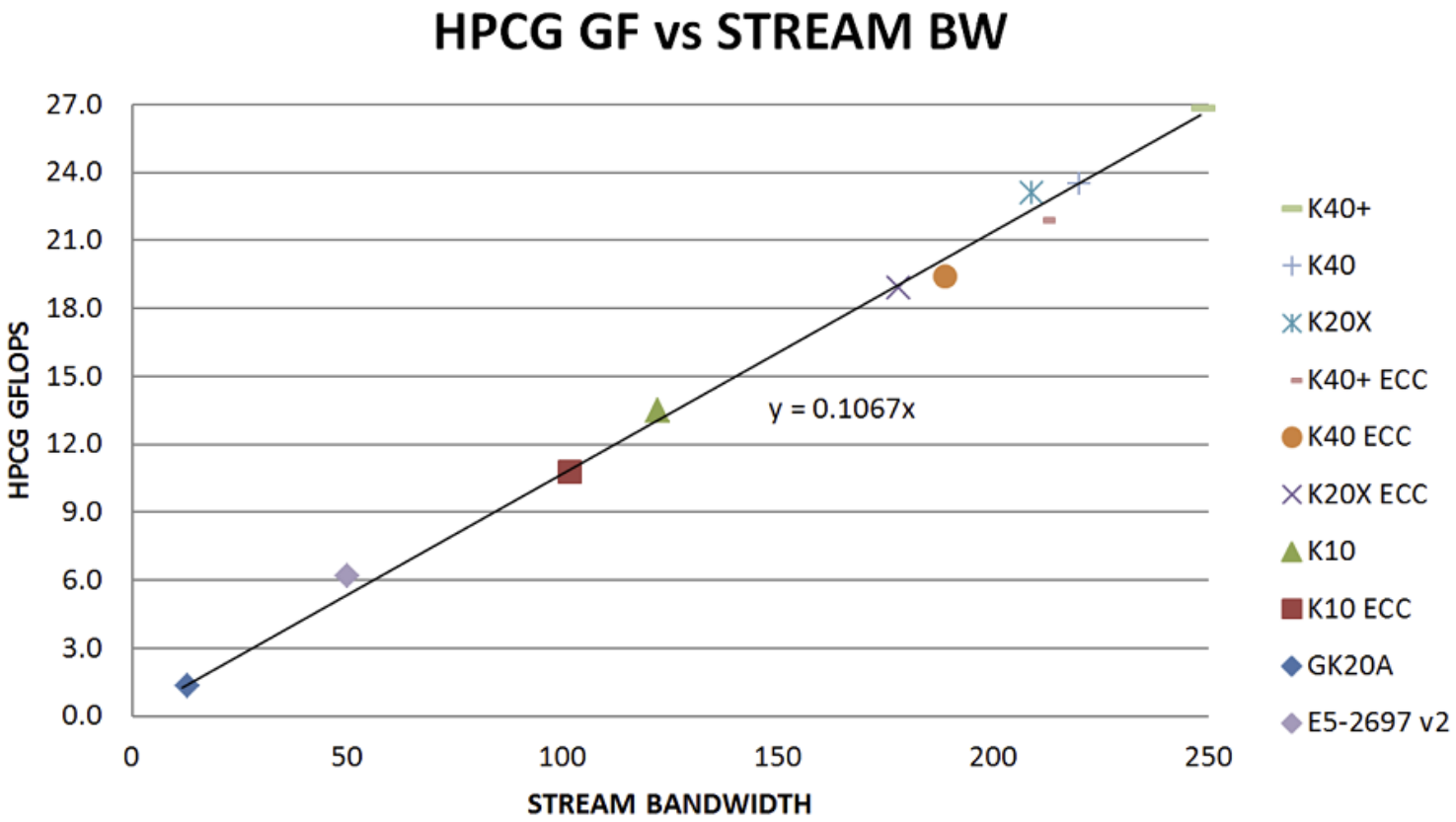
Le nombre de mouvements mémoire et l'intensité arithmétique sont les paramètres dominants pour la performance de la plupart des codes...

HPCG

November 2015 HPCG Results

Site	Computer	Cores	HPL Pflop/s	HPCG Pflop/s	Ratio
GZhou	Th-2 Xeon Phi	3.10^6	33.863 (#1)	0.5800	1.7%
RIKEN	K SPARC64	7.10^5	10.510 (#4)	0.4608	4.3%
ORNL	Titan K20x	5.10^5	17.590 (#2)	0.3223	1.8%
SNL	Trinity Xeon	3.10^5	8.101 (#6)	0.1826	2.25%

HPCG vs STREAM



2. Programmation sur machine parallèle

MPI+X+Y

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	

MPI+X+Y

Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	

MPI+X+Y

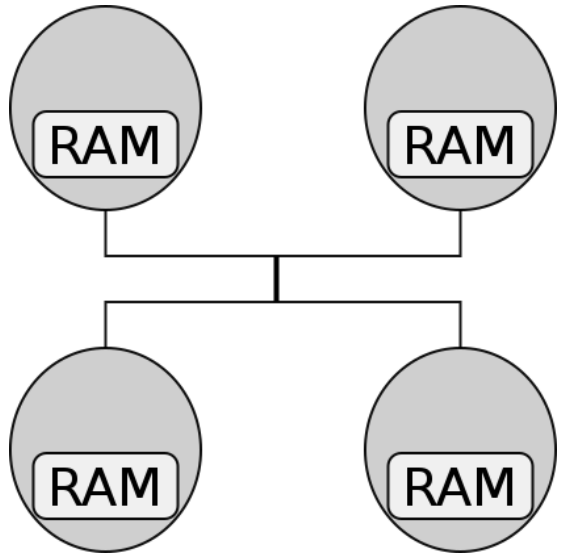
Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	(Multithreading :
CPU	2,16 cores	LLC	OMP,TBB
Core	simd units	L1/L2	Cilk++...)
SIMD	128,512 bits ops	Registers	

MPI+X+Y

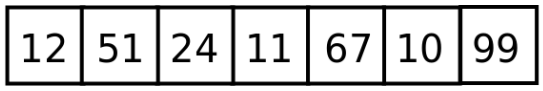
Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	(Multithreading :
CPU	2,16 cores	LLC	OMP,TBB
Core	simd units	L1/L2	Cilk++...)
SIMD	128,512 bits ops	Registers	asm,intrinsics...

Message Passing Interface

- Mémoire distribuée
- Communications explicites
- Communications point à point
- Communications collectives
- Interface bas niveau

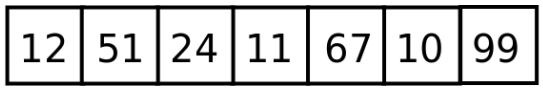


MPI : recherche de minimum



- `int N=7;`

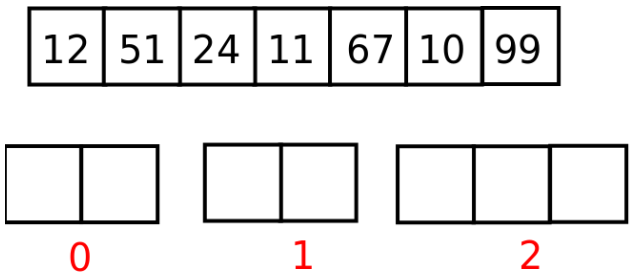
MPI : recherche de minimum



0 1 2

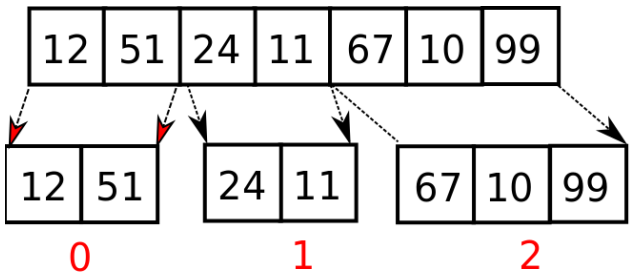
- `int N=7;`
- `int nproc,myrank;`
- `MPI_Comm_size(comm,&nproc);//3`
- `MPI_Comm_rank(comm,&myrank);//0,1,2`

MPI : recherche de minimum



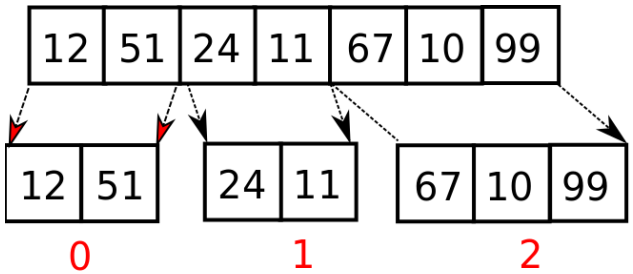
- `int N=7;`
- `int nproc,myrank;`
- `MPI_Comm_size(comm,&nproc);//3`
- `MPI_Comm_rank(comm,&myrank);//0,1,2`
- `int n=N/nproc;//2`
- `int shift=myrank*n;//0,2,4`
- `if (myrank==(nproc-1)) n=N-shift;//3`

MPI : recherche de minimum



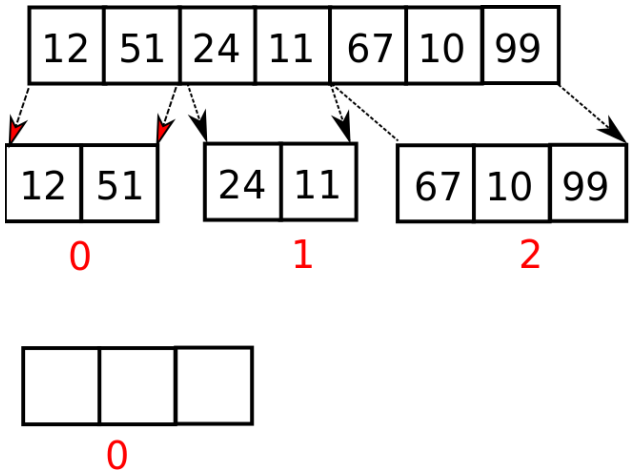
- `int N=7;`
- `int nproc,myrank;`
- `MPI_Comm_size(comm,&nproc);//3`
- `MPI_Comm_rank(comm,&myrank);//0,1,2`
- `int n=N/nproc;//2`
- `int shift=myrank*n;//0,2,4`
- `if (myrank==(nproc-1)) n=N-shift;//3`
- `int local[n]; ...`
  `local[i]=A[shift+i] ...`

MPI : recherche de minimum



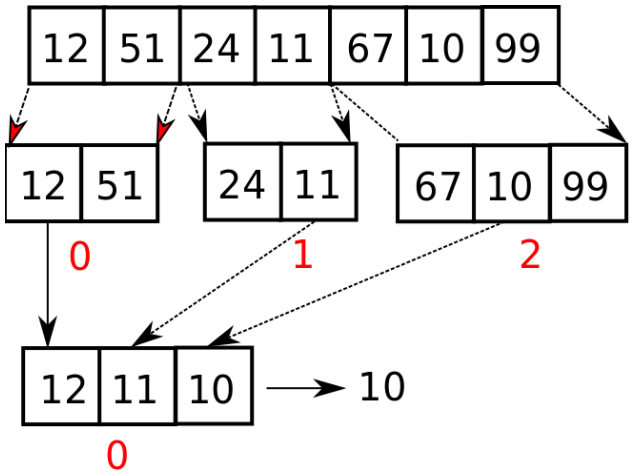
- `int mymin=*std::min(local,local+n);`

MPI : recherche de minimum



- `int mymin=*std::min(local,local+n);`
- `int rec[nproc];`

MPI : recherche de minimum



- `int mymin=*std::min(local,local+n);`
- `int rec[nproc];`
- `MPI_Gather(&mymin,1,MPI_INT, rec,1,MPI_INT,0,comm);`
- `if (myrank==0) mymin=*std::min(rec,rec+nproc);`
- `if (myrank==0) cout<< mymin << endl;`

MPI+X+Y

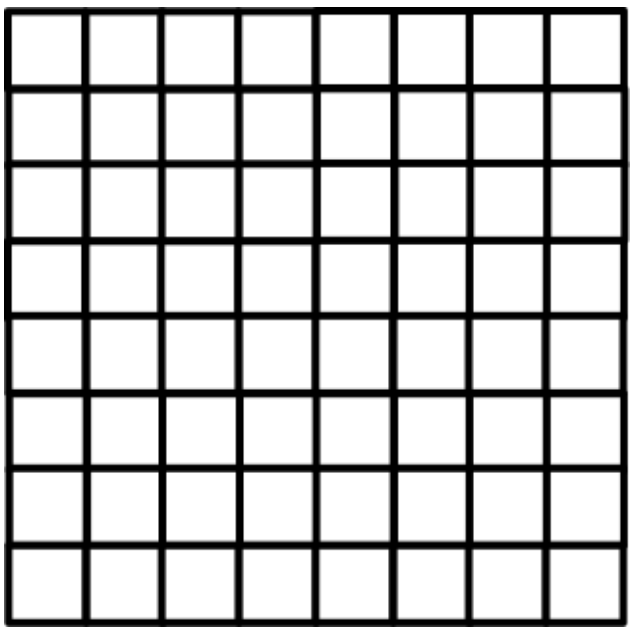
Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	(Multithreading :
CPU	2,16 cores	LLC	OMP,TBB
Core	simd units	L1/L2	Cilk++...)
SIMD	128,512 bits ops	Registers	asm,intrinsics...

Flat MPI (+Y)

Cluster	100,1000 nodes	Distr. Mem.	
Node	1,2,4,8 cpus	Shared Mem.	MPI
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	asm,intrinsics...

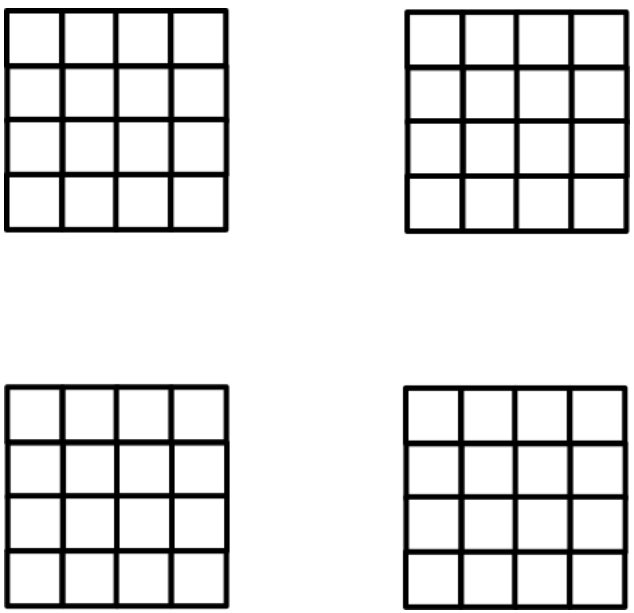
Flat MPI (+Y)

$$\rho_{i,j} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j}$$



Flat MPI (+Y)

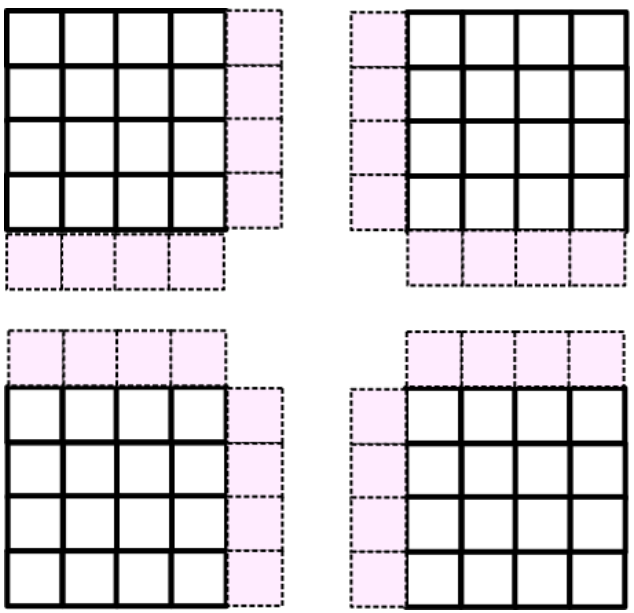
$$\rho_{i,j} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j}$$



- DD

Flat MPI (+Y)

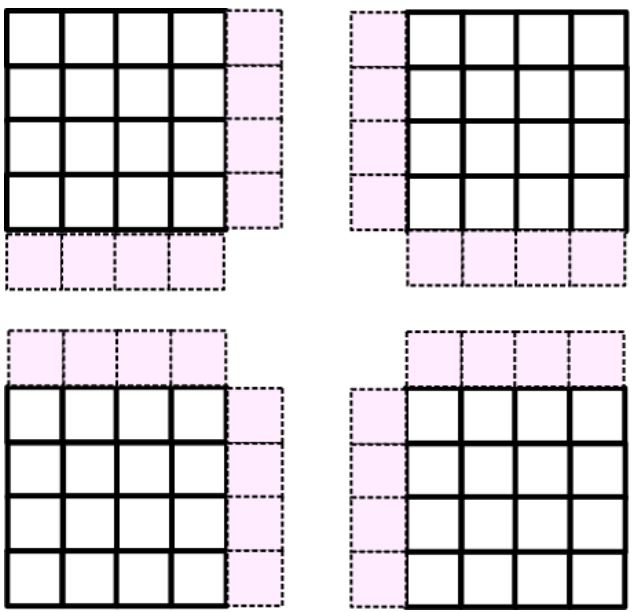
$$\rho_{i,j} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j}$$



- DD
- Surcoût mémoire

Flat MPI (+Y)

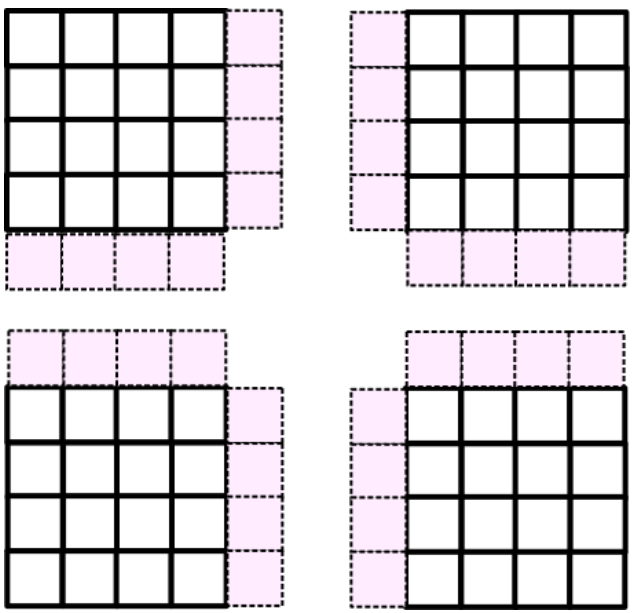
$$\rho_{i,j} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j}$$



- DD
- Surcoût mémoire
- Granularité trop fine

Flat MPI (+Y)

$$\rho_{i,j} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j}$$



- DD
- Surcoût mémoire
- Granularité trop fine
- DD $\rightarrow$ convergence ?
- Équilibrage de charge

MPI+X+Y

Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	(Multithreading :
CPU	2,16 cores	LLC	OMP,TBB
Core	simd units	L1/L2	Cilk++...)
SIMD	128,512 bits ops	Registers	asm,intrinsics...

Multithreading

- Un thread ($\rightsquigarrow$) ou fil d'exécution [1], est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur.
- Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle.
- Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'appel.

(wikipedia)

1. ou tâche ou processus léger

Multithreading : Outils de programmation

Le développement sur la base de bibliothèques de thread bas niveau [2] est très (très) délicat et ne devrait concerner que les développeurs d'OS et de bibliothèques/outils/langages de multi-threading de plus au niveau :

- OpenMP (F77/C/C++)
- Intel Threading Building Blocks (C++) ($\simeq$ MS PPL)
- Cilk++
- ...

2. e.g. thread posix

OpenMP

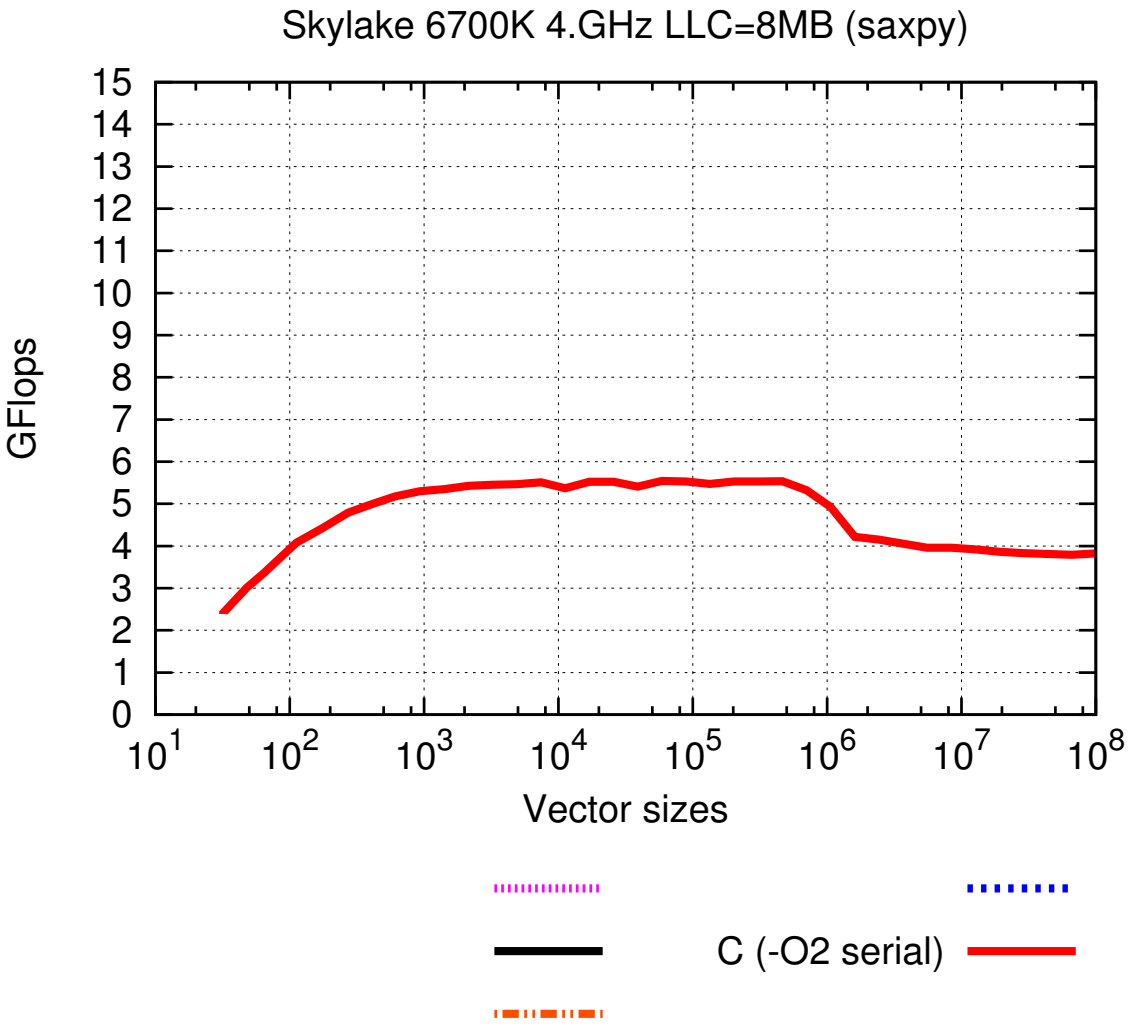
Expression du MT sous la forme de directives (pragma) qui peuvent être insérées dans des codes sources écrits en différents langages (Fortran/C/C++). Les compilateurs qui supportent OpenMP (gcc, icpc, ifort,...) vont générer du code parallèle à partir de ces directives :

- `#pragma omp parallel for`
- `#pragma omp parallel reduce`

Les résultats des programmes OpenMP doivent être les mêmes [3] selon que les directives sont ignorées ou pas.

3. aux erreurs d'arrondi près

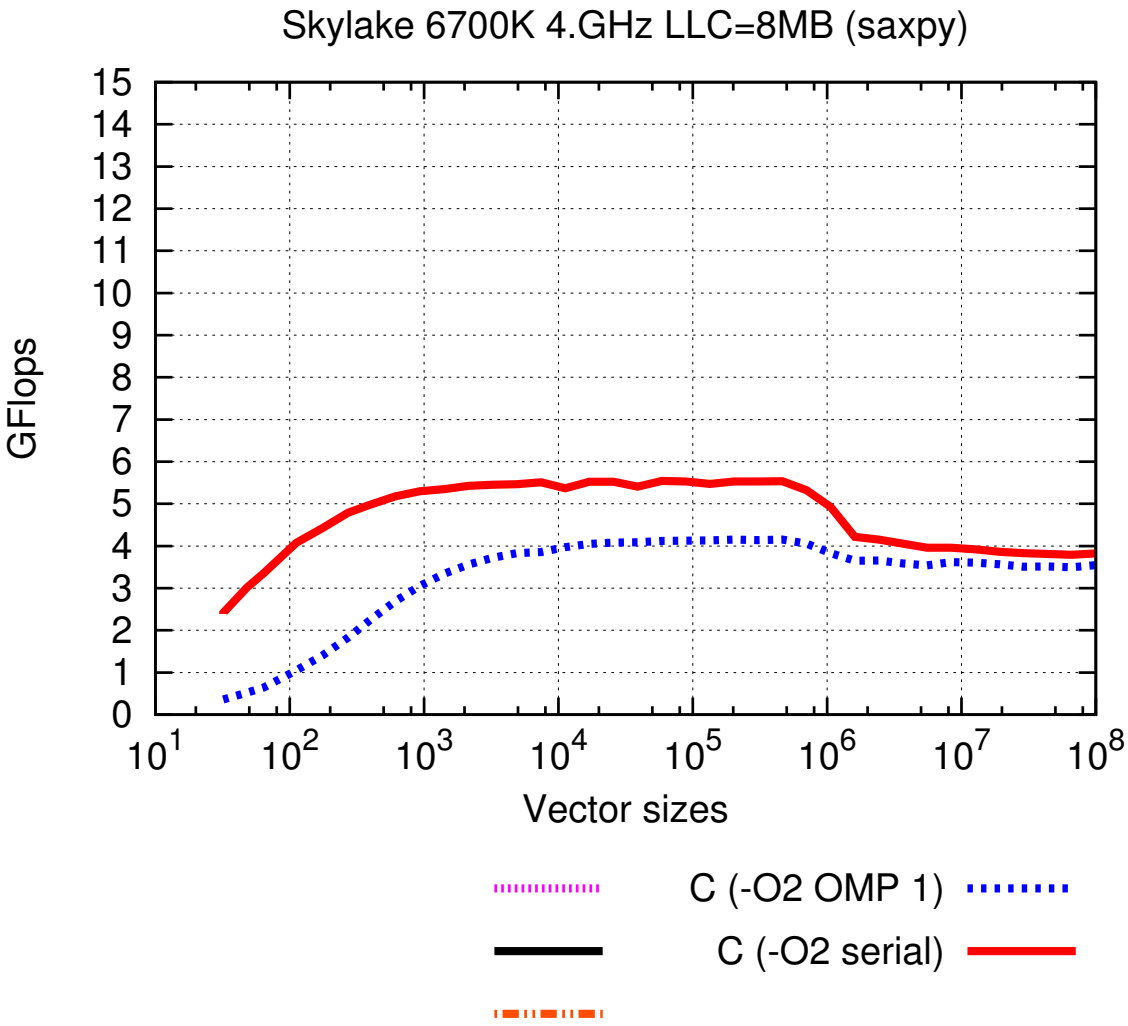
OpenMP SAXPY




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2)

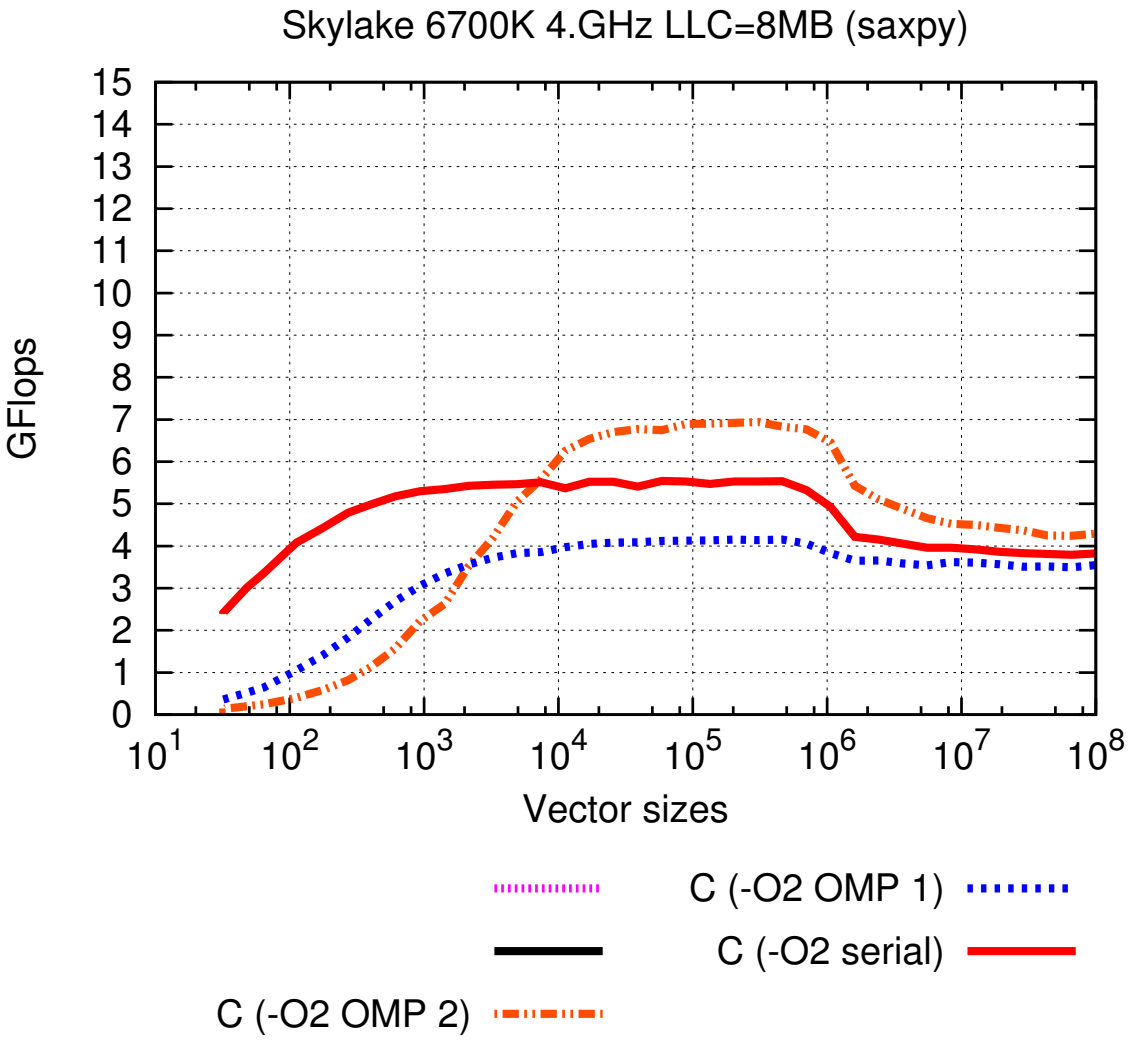
OpenMP SAXPY




```
#pragma omp parallel for
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...
  - 1

OpenMP SAXPY

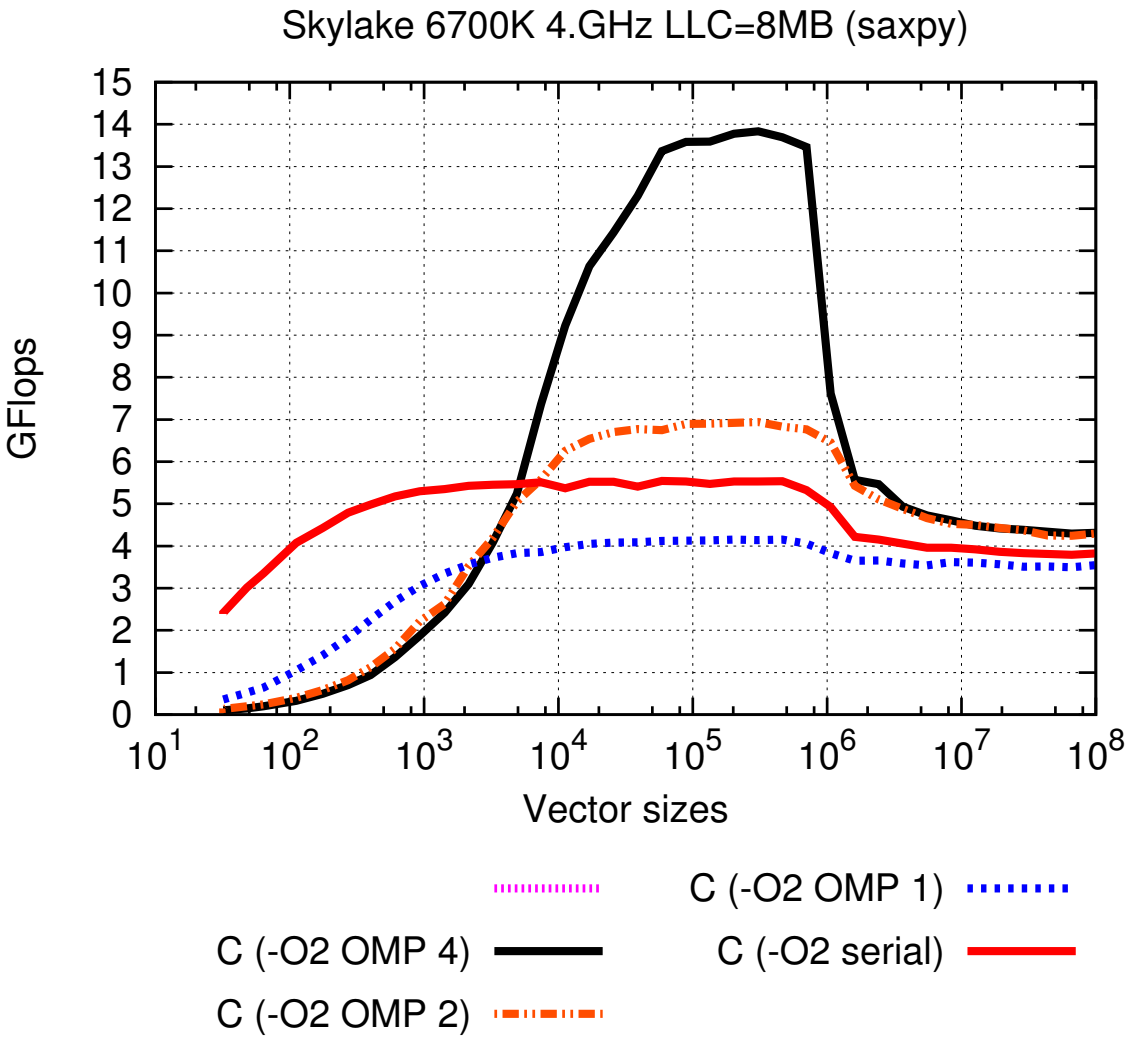



```
#pragma omp parallel for
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

  2

OpenMP SAXPY

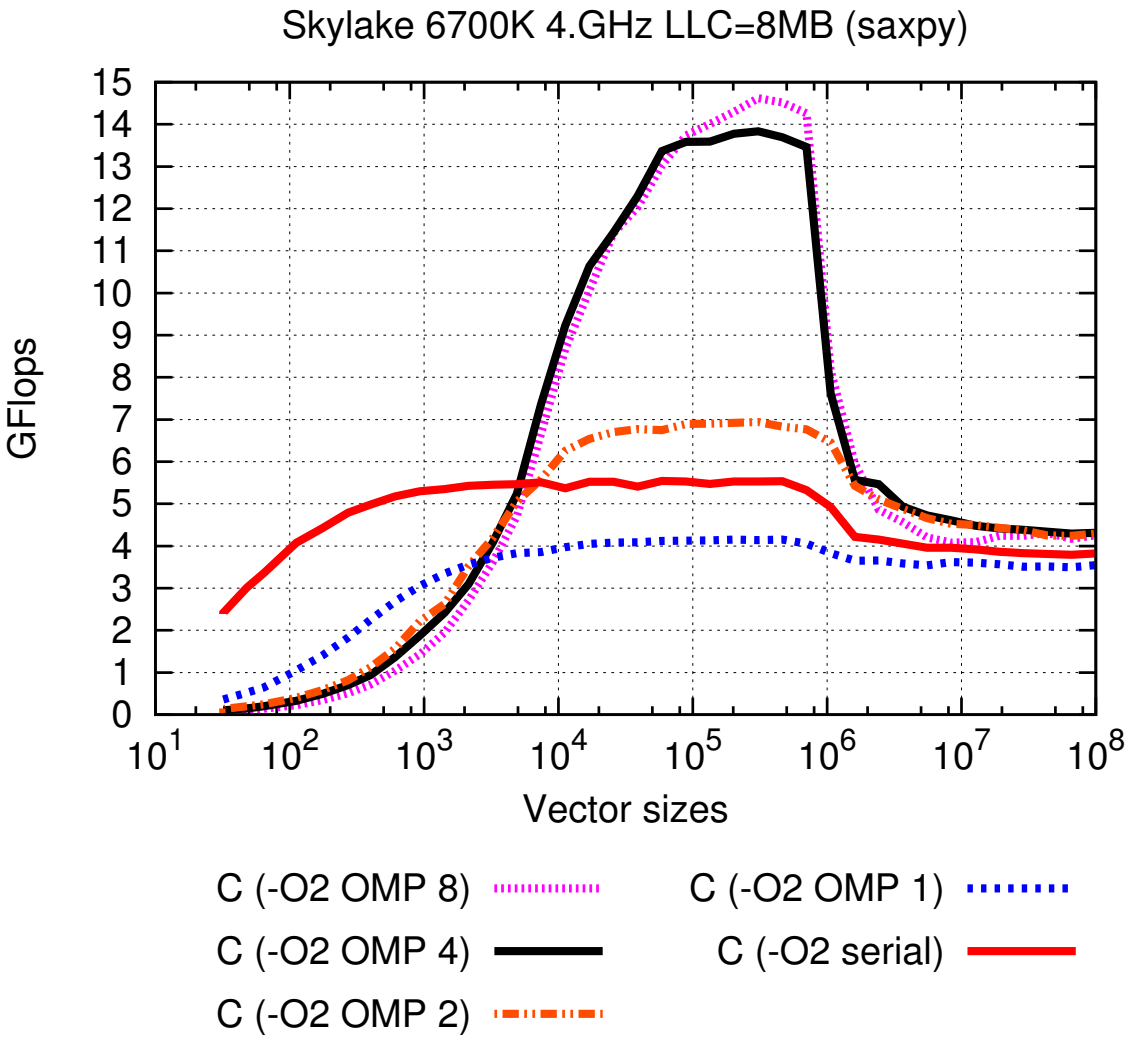



```
#pragma omp parallel for
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

  4

OpenMP SAXPY

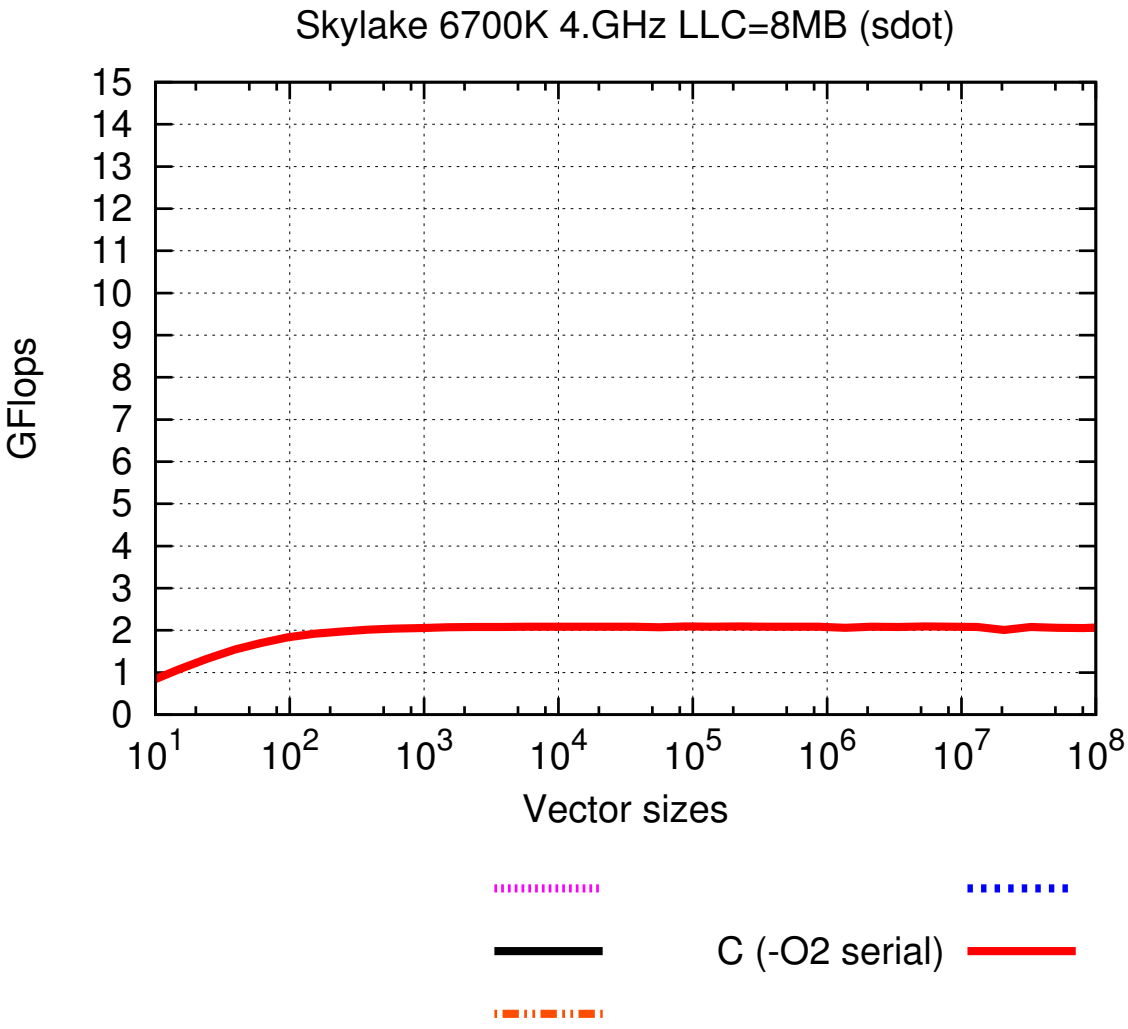



```
#pragma omp parallel for
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

8

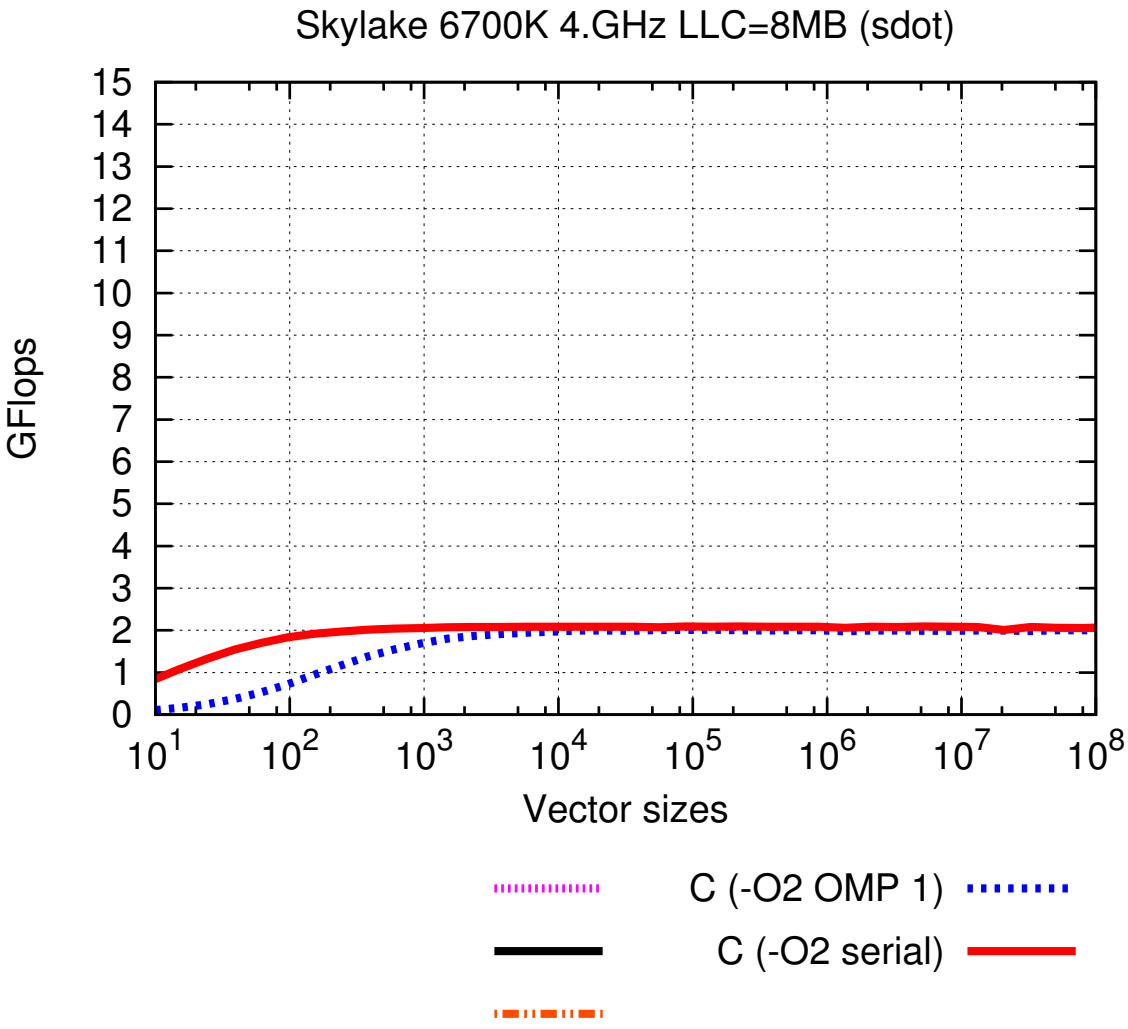
OpenMP SDOT




```
float sum=0.f;
for (int i=0;i<N;i++){
   sum+=X[i]*Y[i];
}
```

- g++6.1 (-O2)

OpenMP SDOT

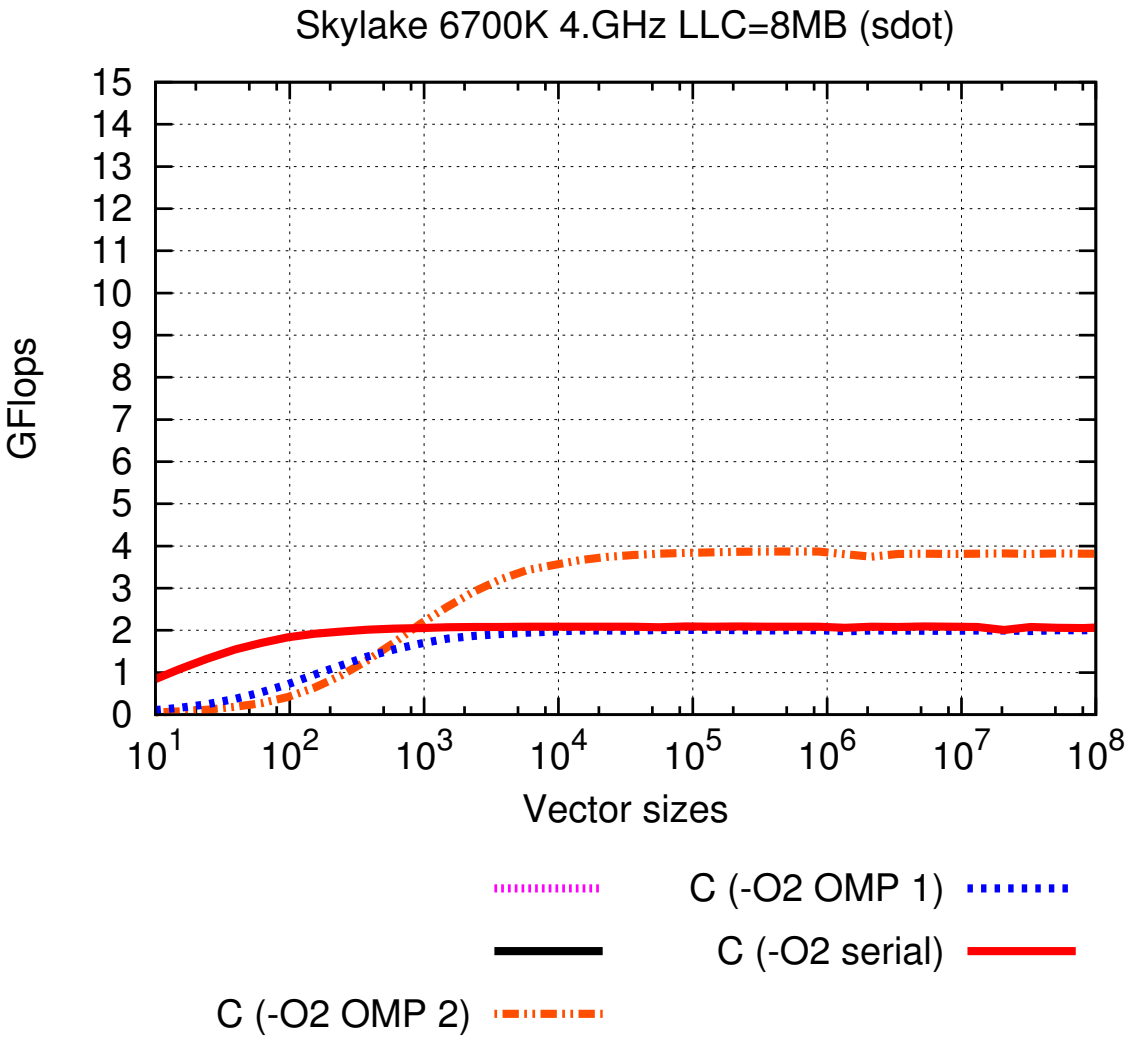



```
#pragma omp parallel \
   for reduction(+:sum)
float sum=0.f;
for (int i=0;i<N;i++){
   sum+=X[i]*Y[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

  1

OpenMP SDOT

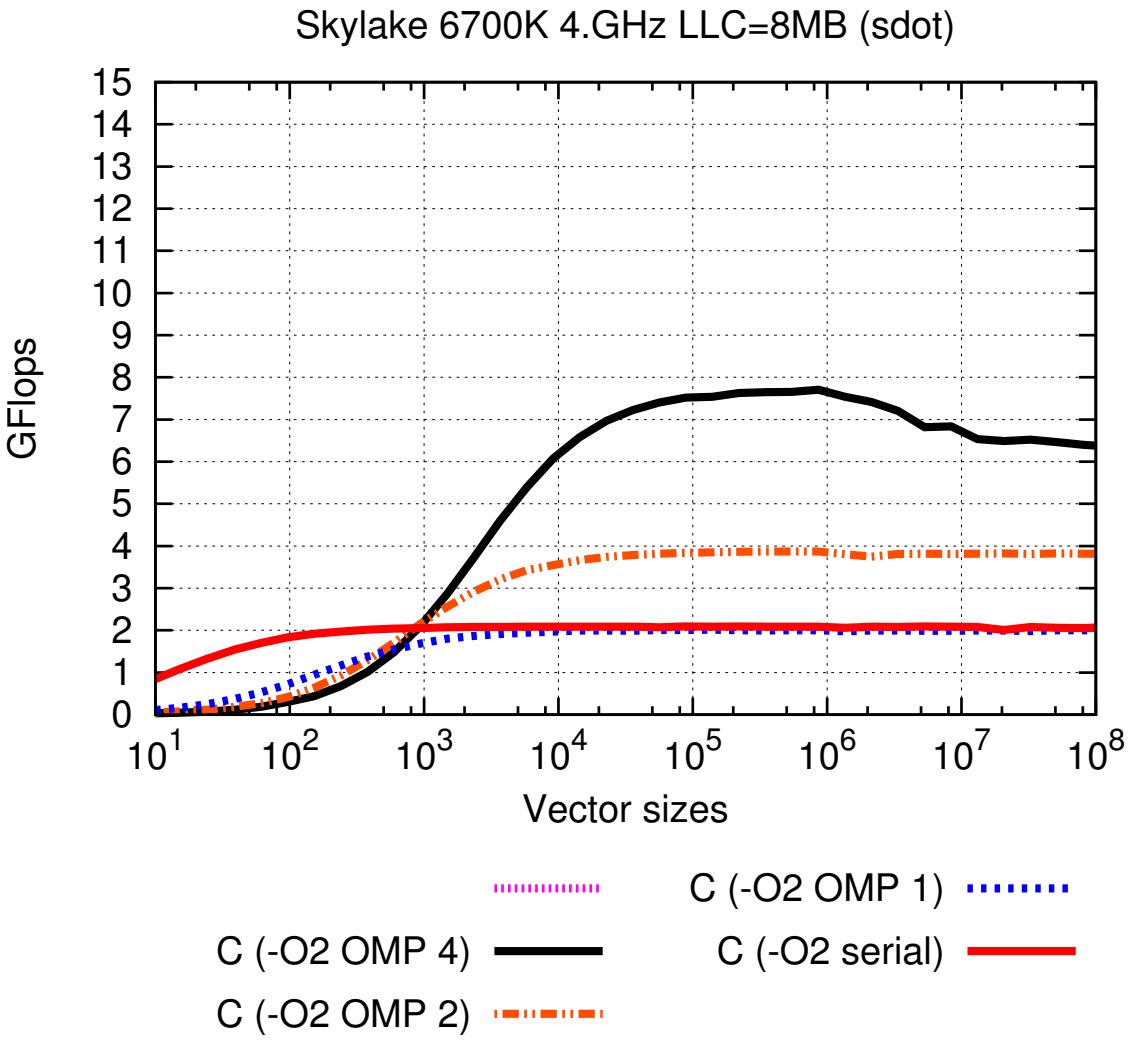



```
#pragma omp parallel \
   for reduction(+:sum)
float sum=0.f;
for (int i=0;i<N;i++){
   sum+=X[i]*Y[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

  2

OpenMP SDOT

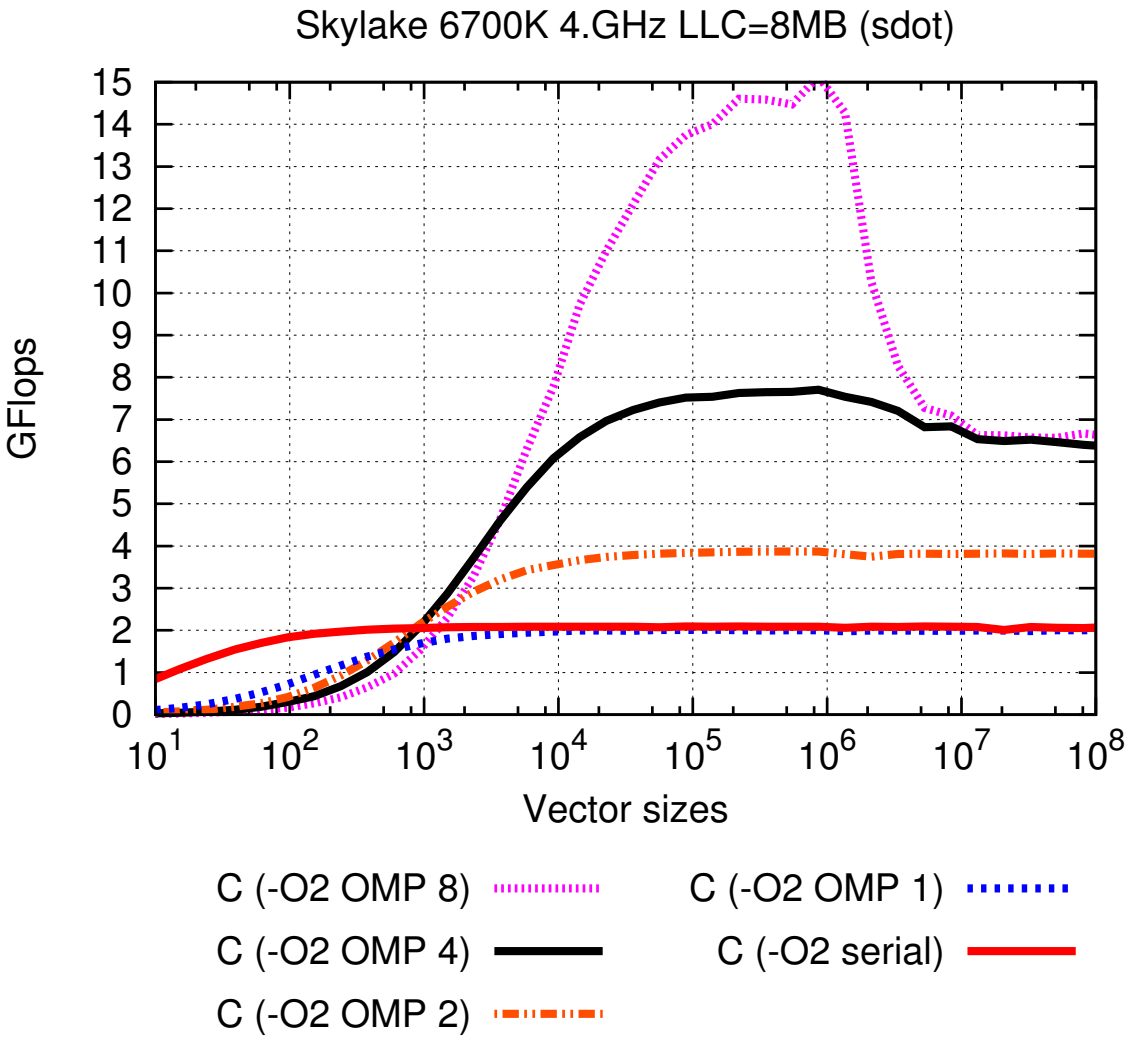



```
#pragma omp parallel \
   for reduction(+:sum)
float sum=0.f;
for (int i=0;i<N;i++){
   sum+=X[i]*Y[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

4

OpenMP SDOT




```
#pragma omp parallel \
   for reduction(+:sum)
float sum=0.f;
for (int i=0;i<N;i++){
   sum+=X[i]*Y[i];
}
```

- g++6.1 (-O2)
- OMP_NUM_THREADS=...

8

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

- Collection de patrons d'algorithmes parallèles :
  - `tbb::parallel_for<>`
  - `tbb::parallel_reduce<>`
  - `tbb::parallel_do<>`
  - ...

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

- Collection de patrons d'algorithmes parallèles :
  - `tbb::parallel_for<>`
  - `tbb::parallel_reduce<>`
  - `tbb::parallel_do<>`
  - ...
- Ordonnancement dynamique.

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

- Collection de patrons d'algorithmes parallèles :
  - `tbb::parallel_for<>`
  - `tbb::parallel_reduce<>`
  - `tbb::parallel_do<>`
  - ...
- Ordonnancement dynamique.
- Équilibrage de charge automatique (vol de tâche).

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

- Collection de patrons d'algorithmes parallèles :
  - `tbb::parallel_for<>`
  - `tbb::parallel_reduce<>`
  - `tbb::parallel_do<>`
  - ...
- Ordonnancement dynamique.
- Équilibrage de charge automatique (vol de tâche).
- Notion de programmation par tâche.

Intel Threading Building Blocks (C++)

TBB : Bibliothèque générique C++ pour écrire des codes MT.

- Collection de patrons d'algorithmes parallèles :
  - `tbb::parallel_for<>`
  - `tbb::parallel_reduce<>`
  - `tbb::parallel_do<>`
  - ...
- Ordonnancement dynamique.
- Équilibrage de charge automatique (vol de tâche).
- Notion de programmation par tâche.
- Open source (GPLv2) et multiOS.

TBB Axpy : Ranges et Functors

```
float coef=2.f;
int N=100;
float X[N];
float Y[N];
...
//Range<int>: Intervalle d'entiers splitable
tbb::blocked_range<int> myRange(0,N); //loop range
//myRange.begin()->0 et myRange.end()->N

//Functor: Class with operator()
AxpyFunctor myFunctor(X,Y,coef);//user defined


tbb::parallel_for(myRange,myFunctor);
```

TBB Axpy : Ranges et Functors

```
struct AxpyFunctor{
 float * X_;
 float * Y_;
 float coef_;

 AxpyFunctor(float * X,
             float * Y,
             float coef):X_(X),Y_(Y),coef_(coef){}

 inline void
 operator ()(const blocked_range<int> & r) const{
   for (int i=r.begin() ; i!=r.end() ; i++){
     Y_[i]+=coef_*X_[i];
   }
 }
};
```

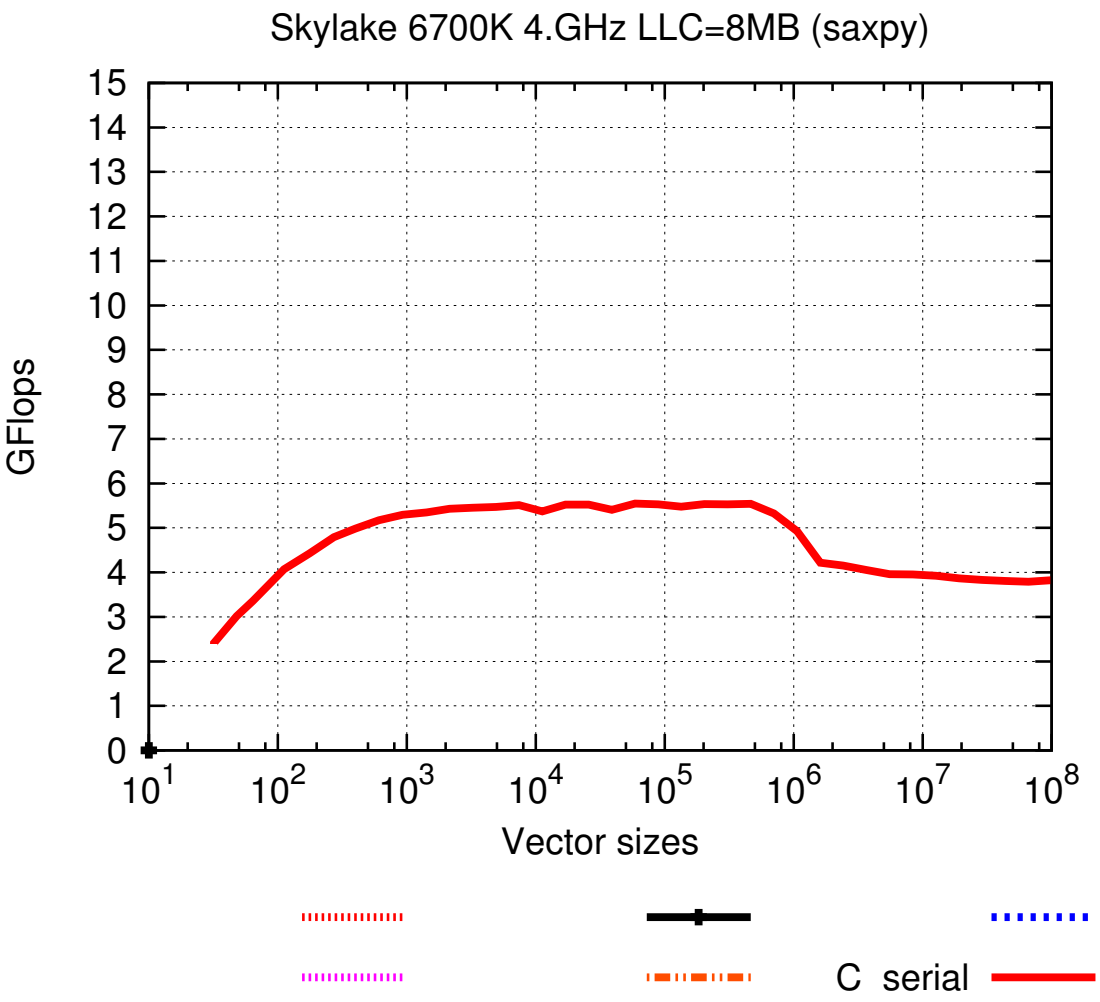
TBB Axpy : Ranges et Functors

```
tbb::blocked_range<int> myRange(0,N);

AxpyFunctor myFunctor(X,Y,coef);

tbb::parallel_for(myRange,myFunctor);
//same result as myFunctor(myRange)
```

TBB Axpy : Ranges et Functors

```
//thread number
tbb::task_scheduler_init init(4);
//grain size
tbb::blocked_range<int> myRange(0,N,128);

AxpyFunctor myFunctor(X,Y,coef);

tbb::parallel_for(myRange,myFunctor);
//same result as myFunctor(myRange)
```

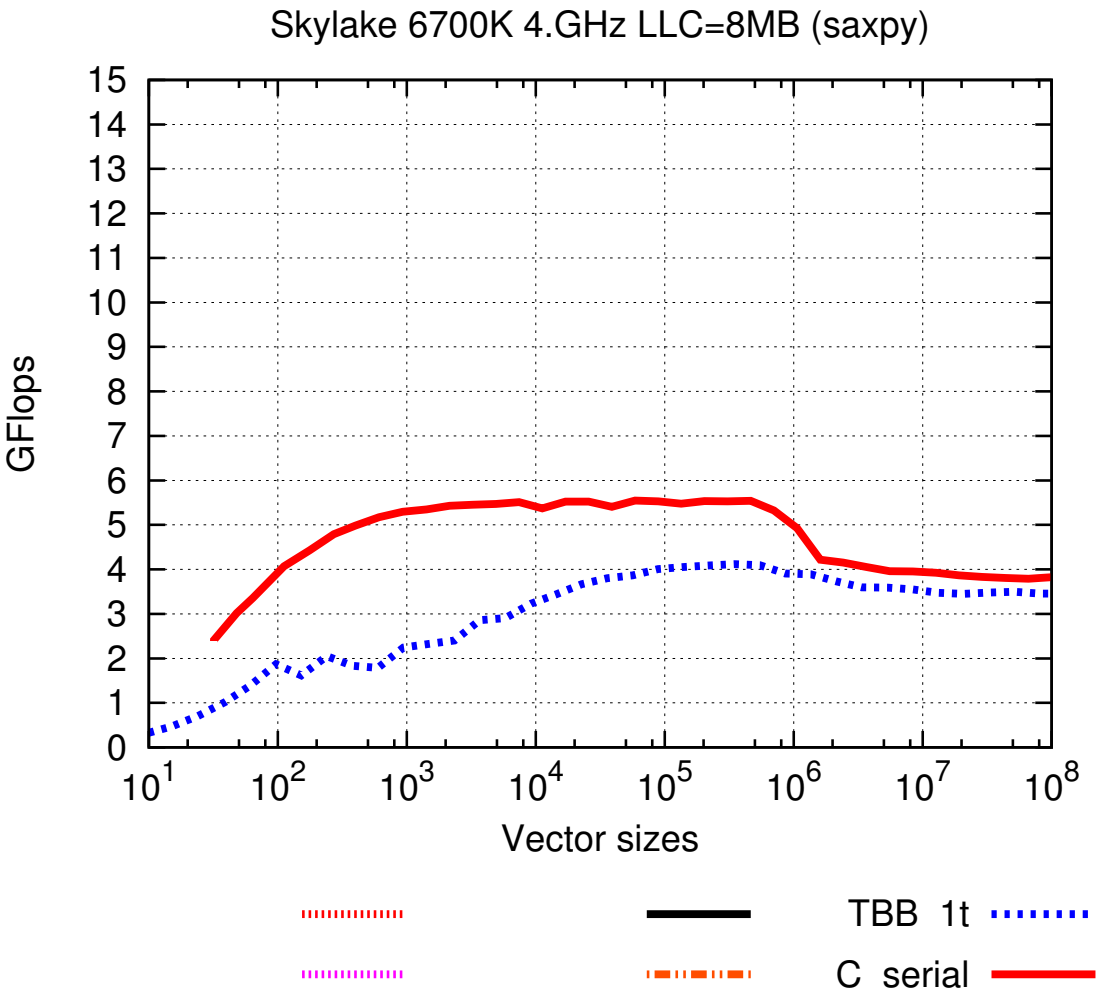
TBB SAXPY




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2) ; TBB4.4

TBB SAXPY

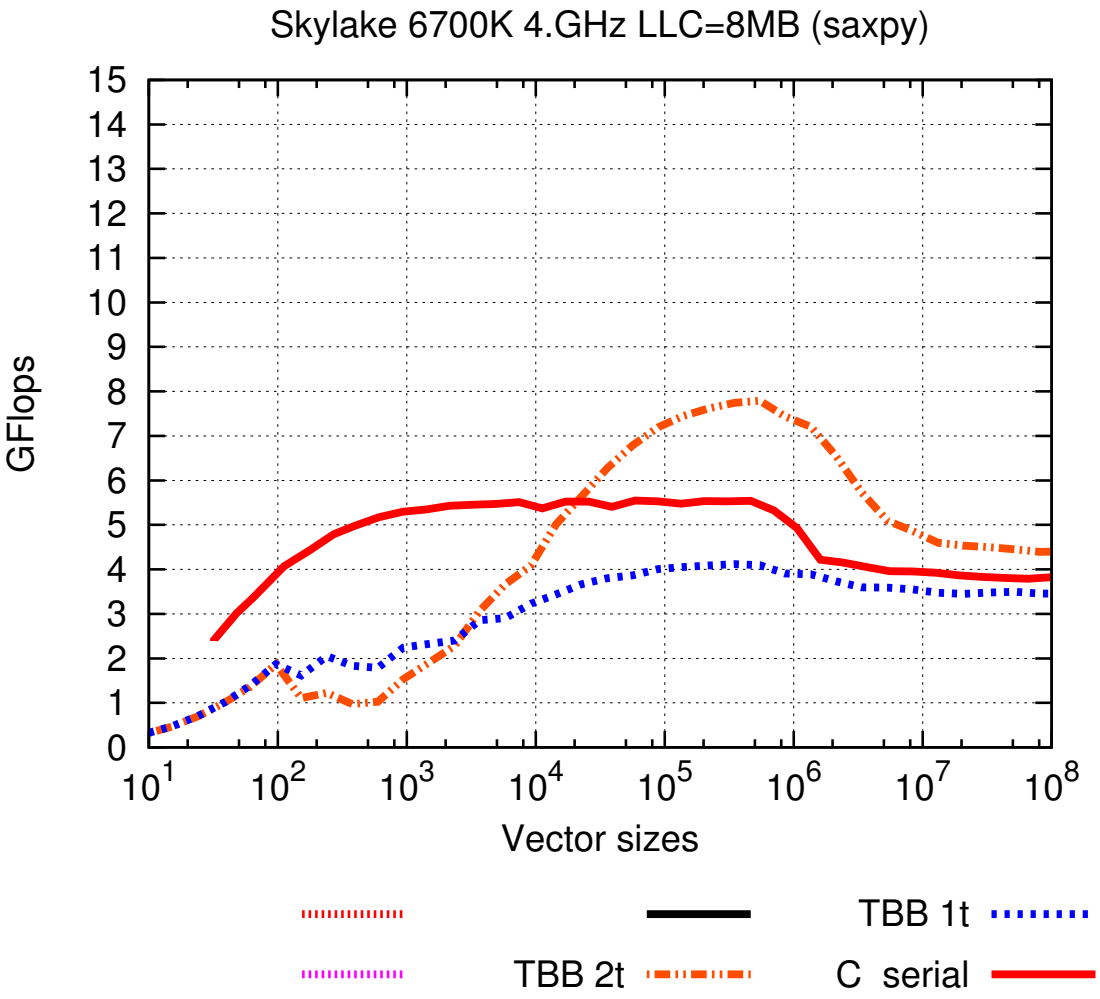



```
blocked_range<int> r(0,N,128);
AxpyFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

  1↝

TBB SAXPY

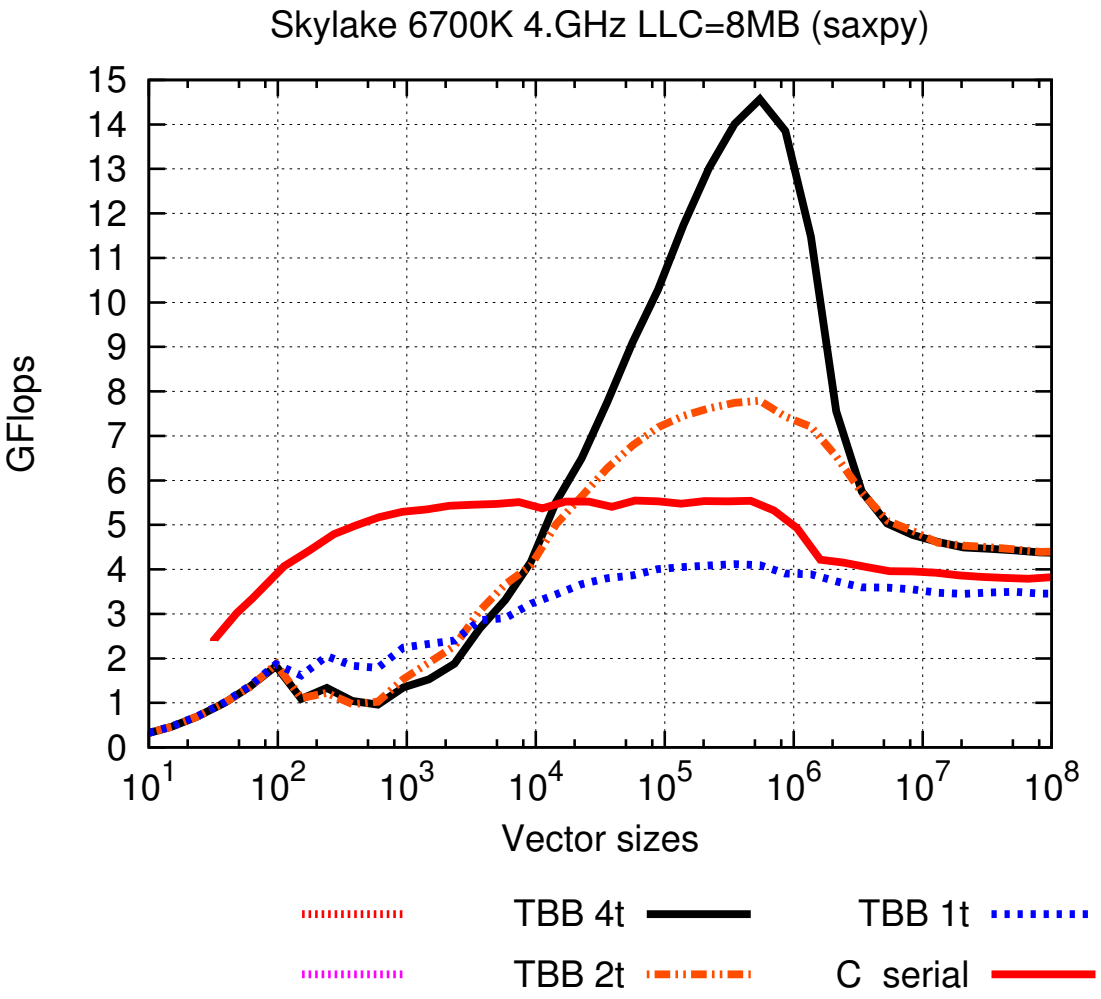



```
blocked_range<int> r(0,N,128);
AxpyFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

2⇝

TBB SAXPY

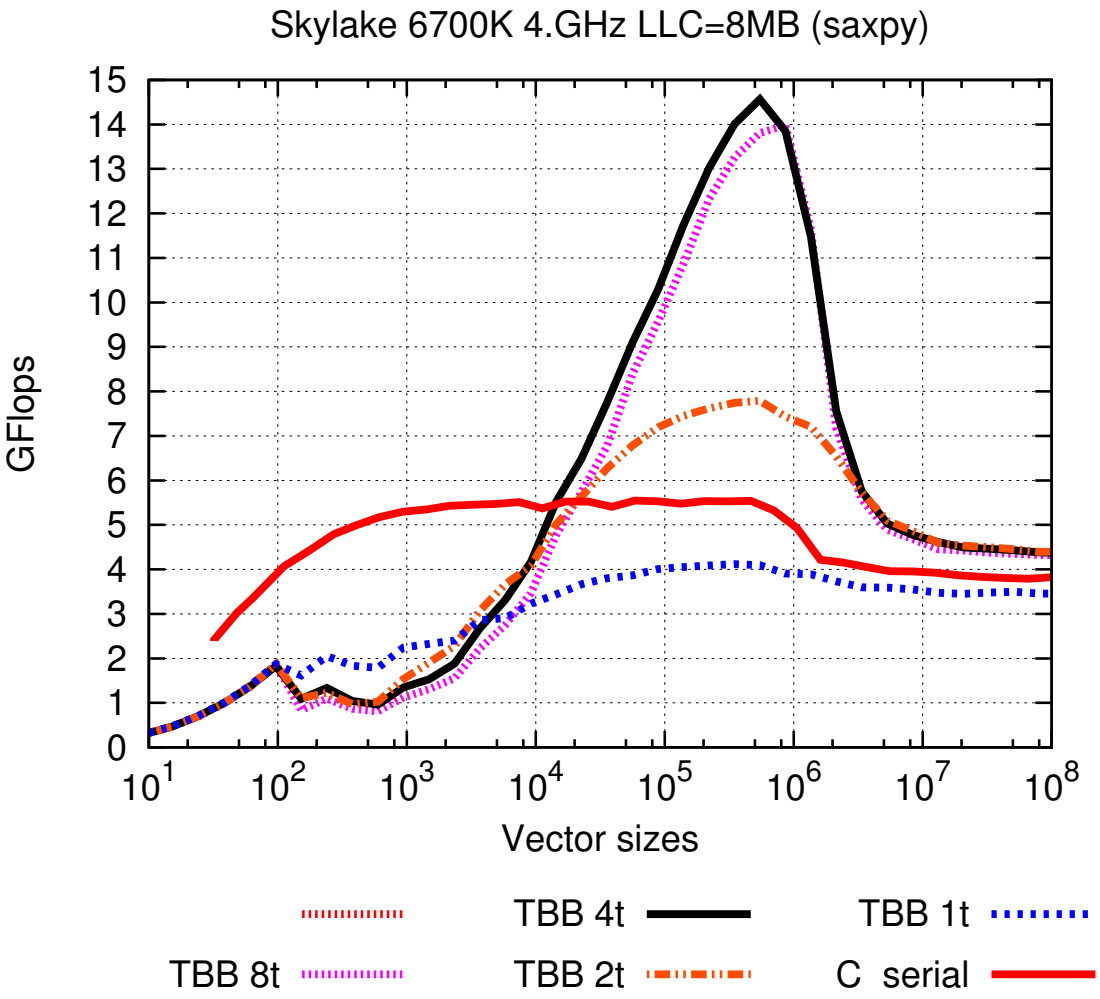



```
blocked_range<int> r(0,N,128);
AxpyFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

4↝

TBB SAXPY

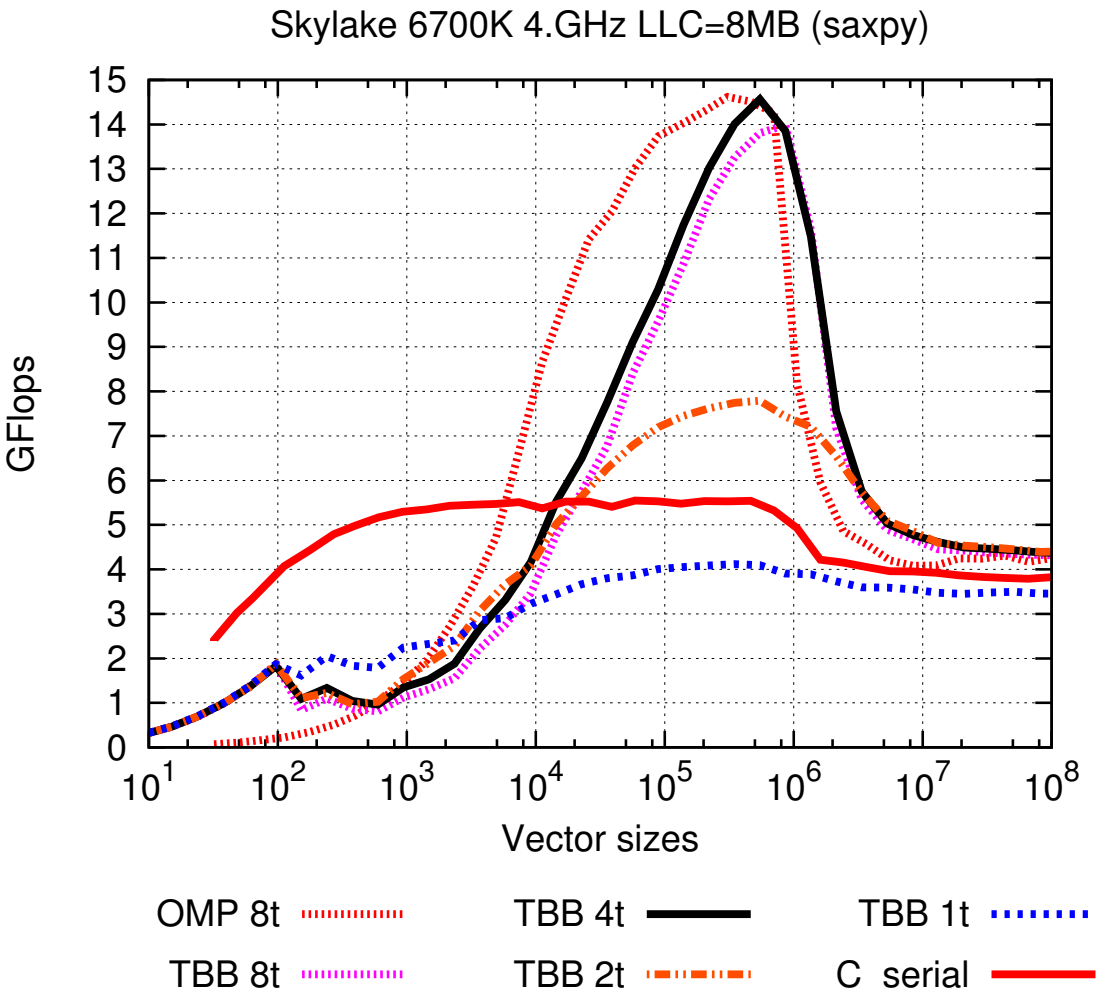



```
blocked_range<int> r(0,N,128);
AxpyFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

8↝

TBB SAXPY




```
blocked_range<int> r(0,N,128);
AxpyFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

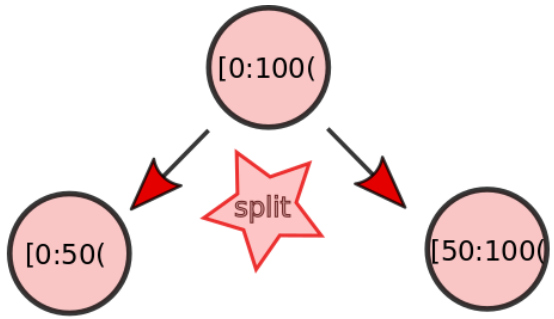
8↝

TBB : réductions

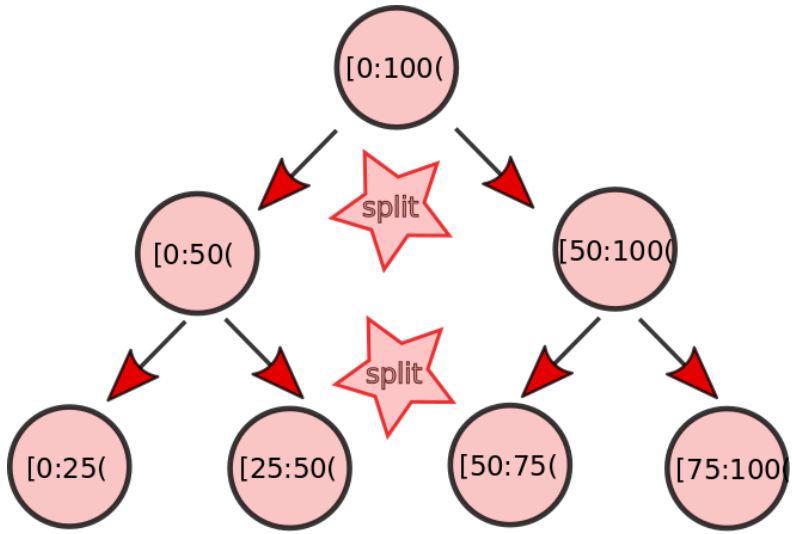
```
int N=100;
float X[N];
float Y[N];
...

tbb::blocked_range<int> myRange(0,N);

DotFunctor myFunctor(X,Y);

tbb::parallel_reduce(myRange,myFunctor);
//same result as myFunctor(myRange)
```

TBB reduce : split/join

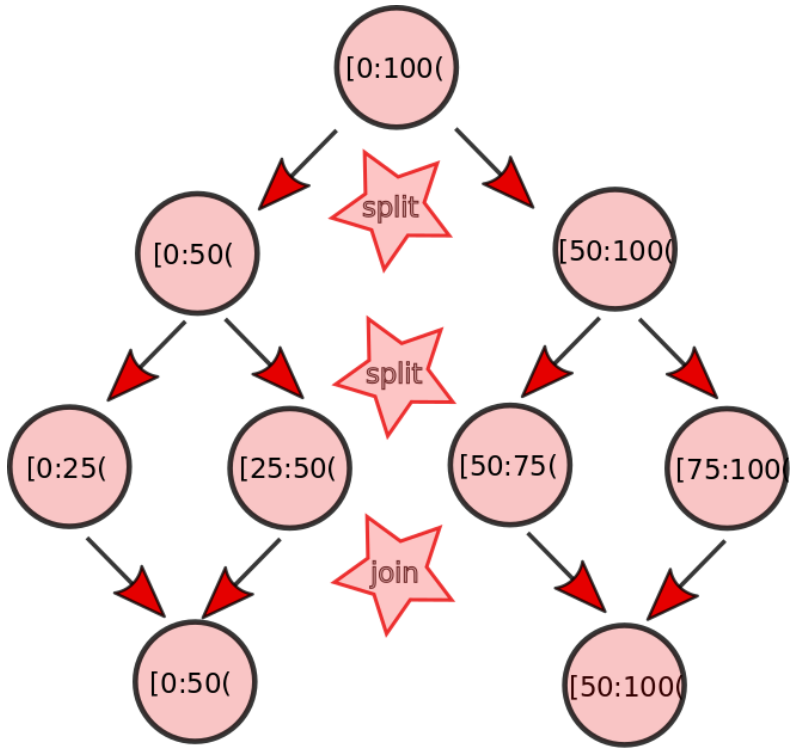
TBB reduce : split/join



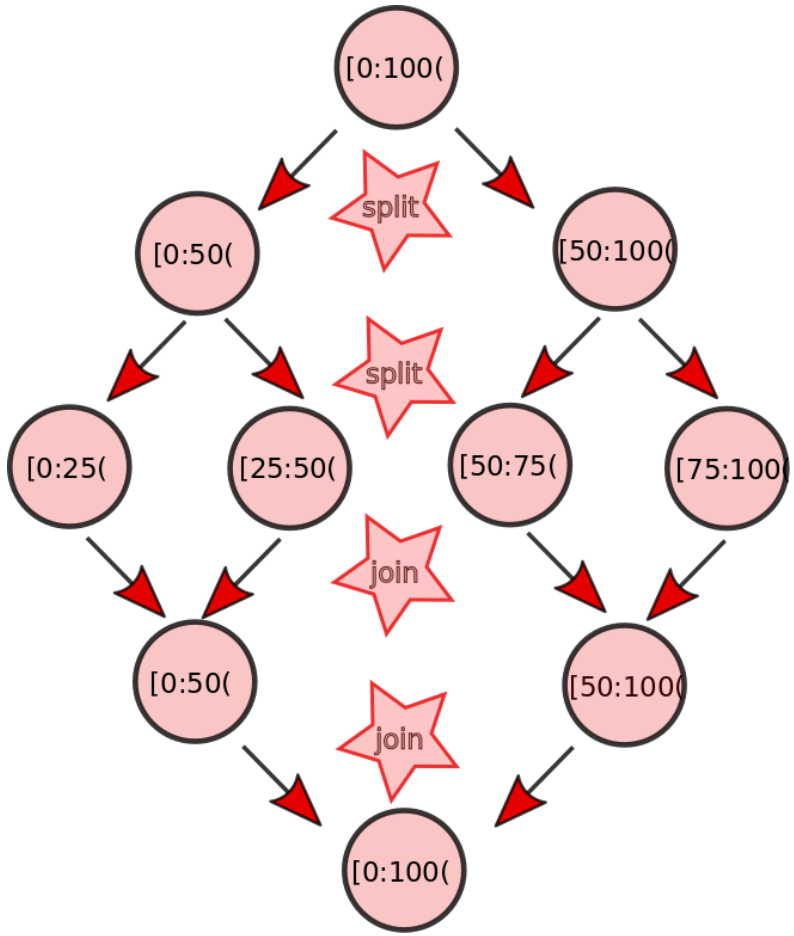
TBB reduce : split/join



TBB reduce : split/join



TBB reduce : split/join



TBB DotFunctor

```
struct DotFunctor{
 float * X_;
 float * Y_;
 float s_;
 DotFunctor(float * X,
            float * Y):X_(X),Y_(Y),s_(0.){}



 inline void   //non const !
 operator ()(const blocked_range<int> & r){
   for (int i=r.begin() ; i!=r.end() ; i++){
     s_+=X_[i]*Y_[i];
   }
 }



};
```

TBB DotFunctor

```
struct DotFunctor{
 float * X_;
 float * Y_;
 float s_;
 DotFunctor(float * X,
            float * Y):X_(X),Y_(Y),s_(0.){}
//split Ctor !
 DotFunctor(DotFuntor & o,
            tbb::split):X_(o.X_),Y_(o.Y_),s_(0.){}
 inline void   //non const !
 operator ()(const blocked_range<int> & r){
   for (int i=r.begin() ; i!=r.end() ; i++){
     s_+=X_[i]*Y_[i];
   }
 }



};
```

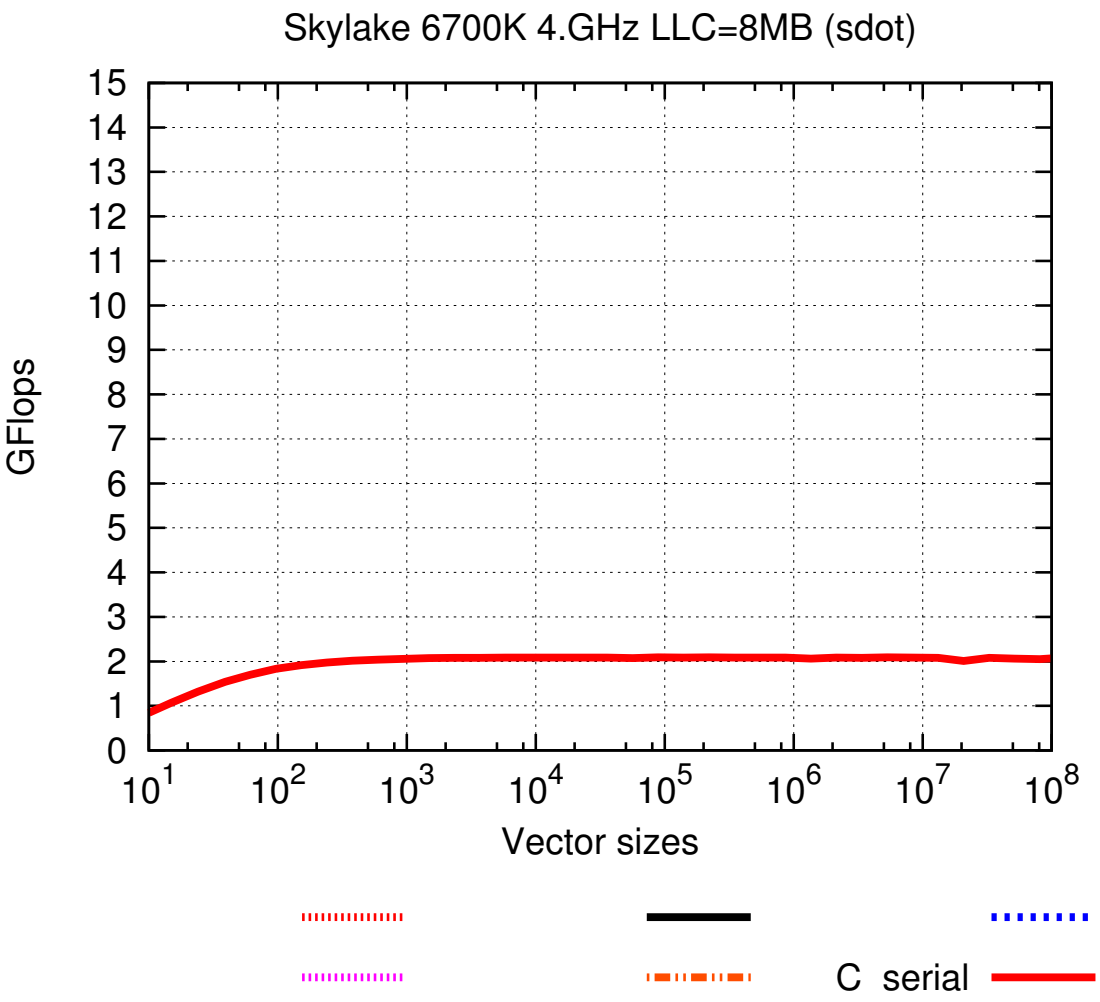
TBB DotFunctor

```
struct DotFunctor{
 float * X_;
 float * Y_;
 float s_;
 DotFunctor(float * X,
            float * Y):X_(X),Y_(Y),s_(0.){}
//split Ctor !
 DotFunctor(DotFuntor & o,
            tbb::split):X_(o.X_),Y_(o.Y_),s_(0.){}
 inline void   //non const !
 operator ()(const blocked_range<int> & r){
   for (int i=r.begin() ; i!=r.end() ; i++){
     s_+=X_[i]*Y_[i];
   }
 }
 void join(DotFunctor & other){
   s_ += other.s_;
 }
};
```

TBB Ranges et Functors

```
tbb::blocked_range<int> myRange(0,N);

DotFunctor myFunctor(X,Y);

tbb::parallel_reduce(myRange,myFunctor);
//same result as myFunctor(myRange)
```

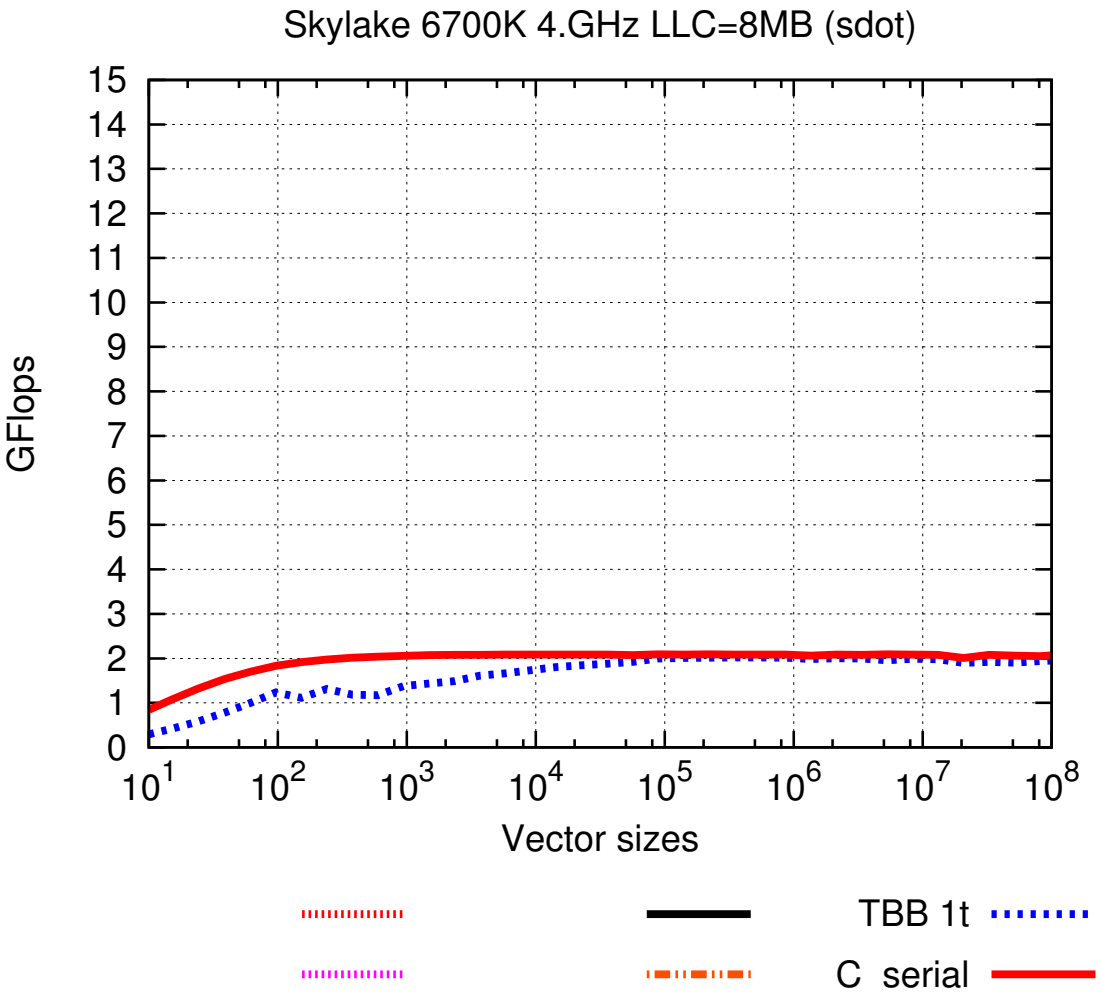
TBB SDOT




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1 (-O2) ; TBB4.4

TBB SDOT

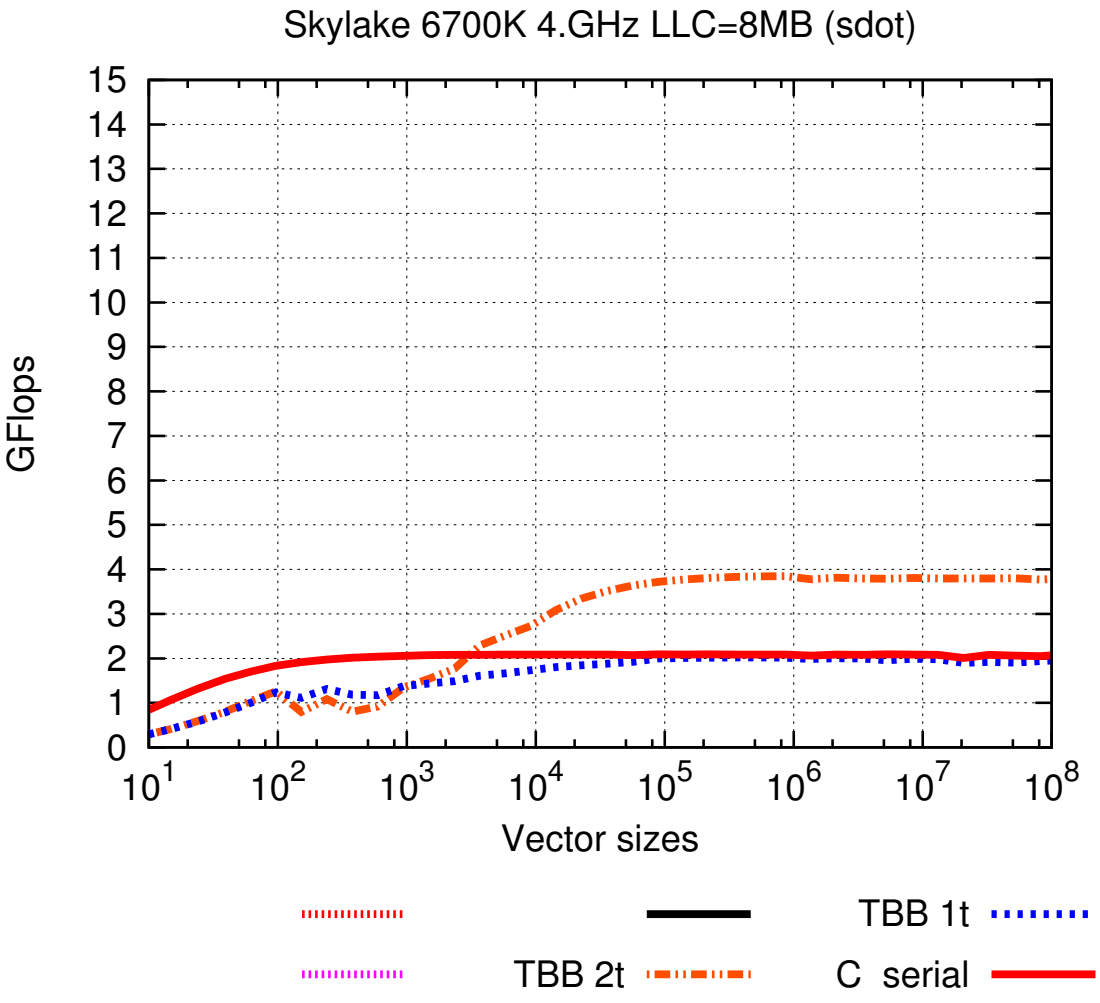



```
blocked_range<int> r(0,N,128);
DotFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

  1↝

TBB SDOT

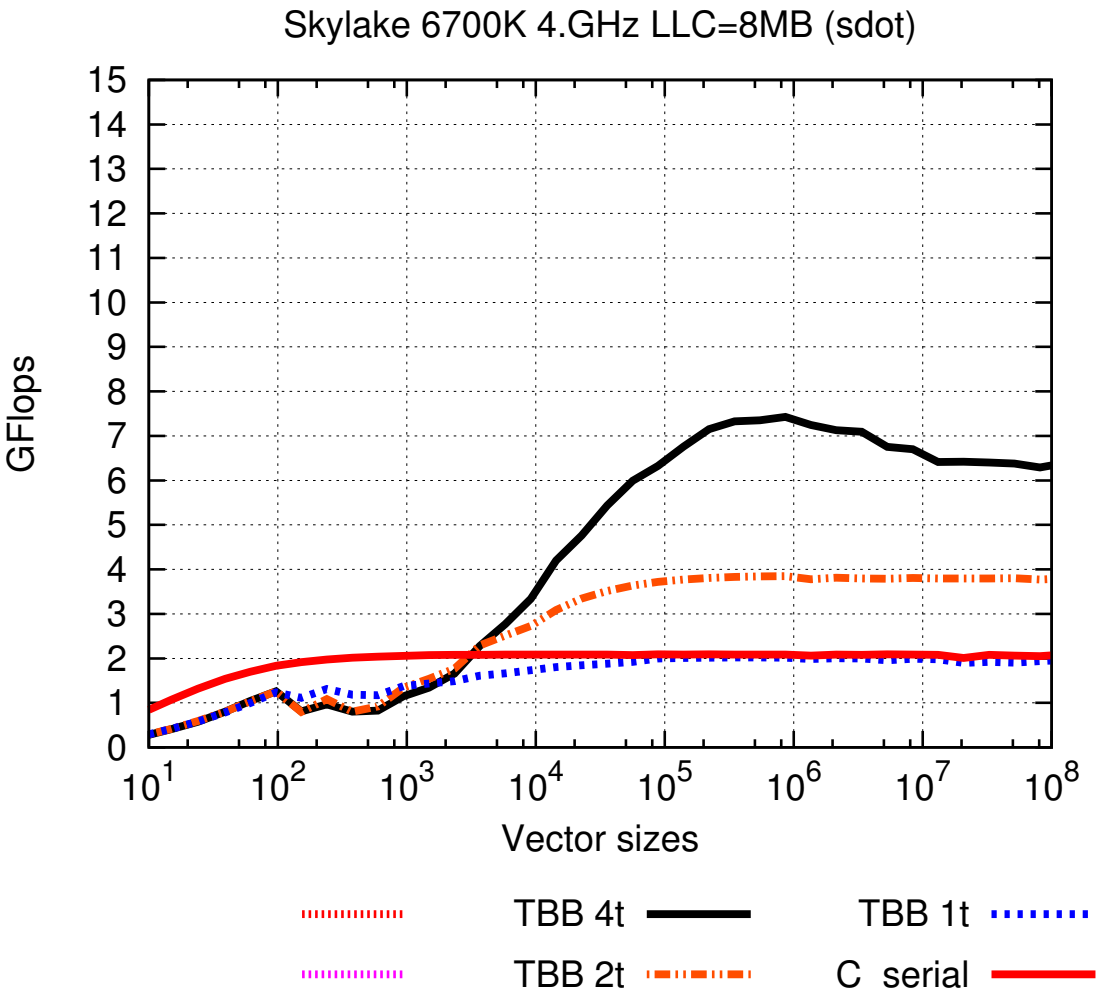



```
blocked_range<int> r(0,N,128);
DotFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

2⇝

TBB SDOT

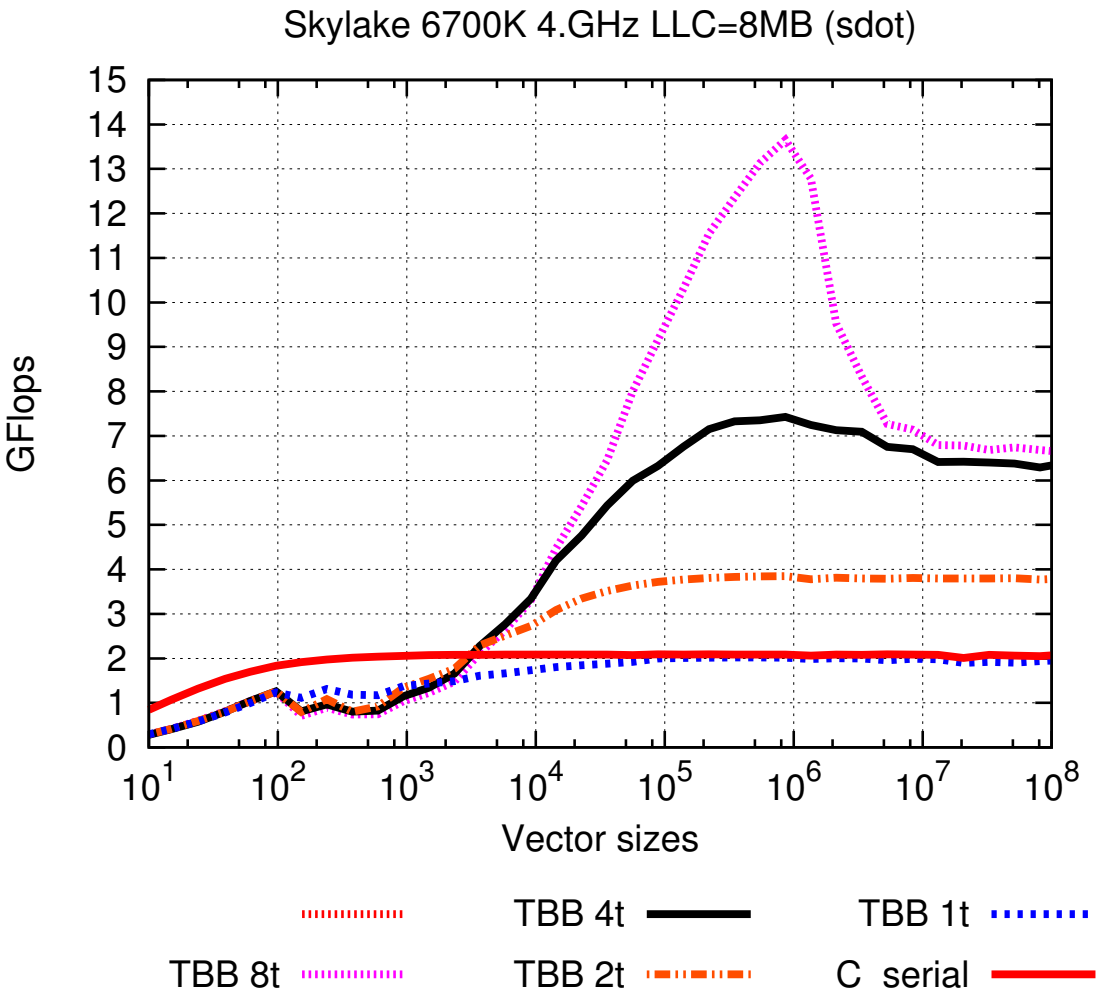



```
blocked_range<int> r(0,N,128);
DotFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

4↝

TBB SDOT

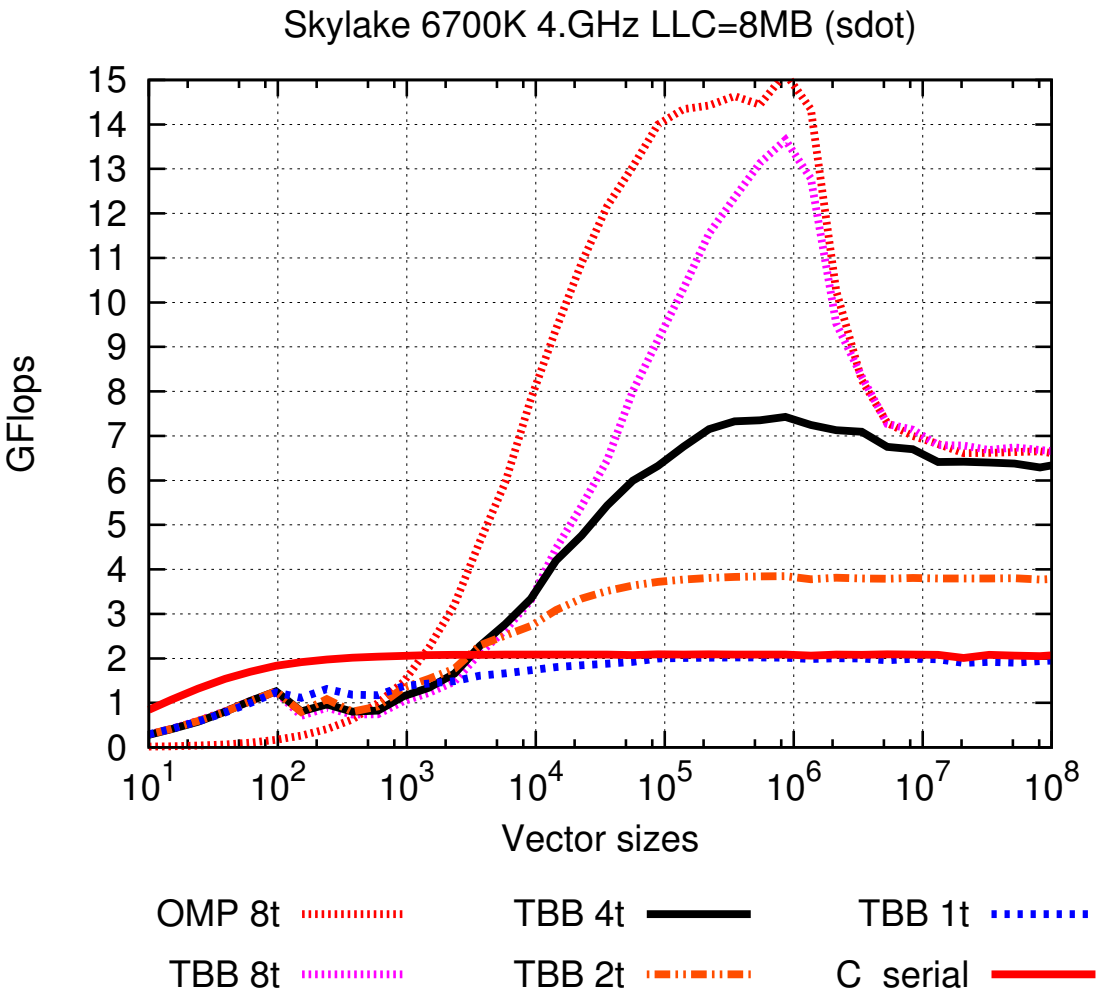



```
blocked_range<int> r(0,N,128);
DotFunctor f(X,Y,coef);
parallel_for(r,f);
```

- g++6.1 (-O2) ; TBB4.4

8↝

TBB SDOT




```
blocked_range<int> r(0,N,128);
DotFunctor f(X,Y,coef);
parallel_for(r,f);
```

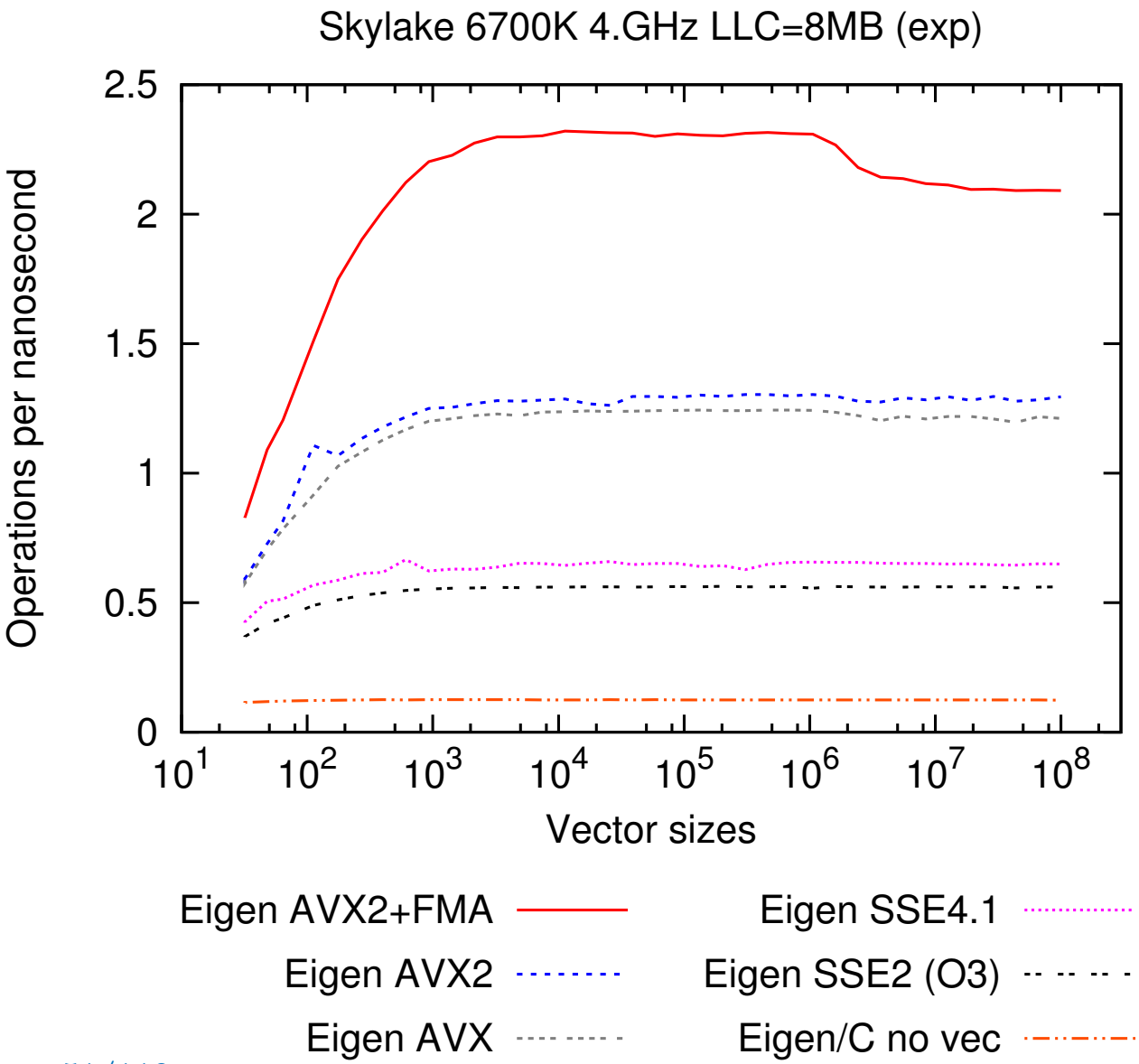
- g++6.1 (-O2) ; TBB4.4

8↝

MPI+X+Y

Cluster	100,1000 nodes	Distr. Mem.	MPI
Node	1,2,4,8 cpus	Shared Mem.	(Multithreading :
CPU	2,16 cores	LLC	OMP,TBB
Core	simd units	L1/L2	Cilk++...)
SIMD	128,512 bits ops	Registers	asm,intrinsics...

Exemple SIMD : exp x




```
for (int i=0;i<N;i++
  X[i]=exp(X[i]);
}
```

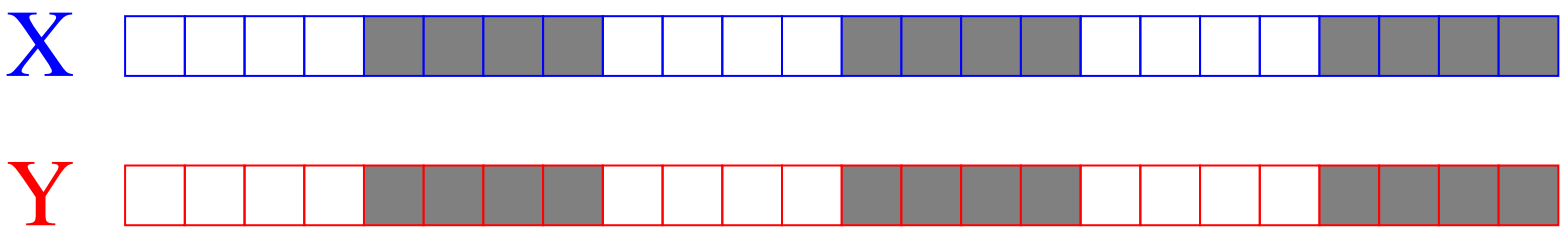
Parallélisme SIMD

SIMD : Single Instruction Multiple Data) :
→ CPU (MMX,SSE,AVX,AVX2,AVX512,AltiVec),GPU

Exemple AXPY (C++)

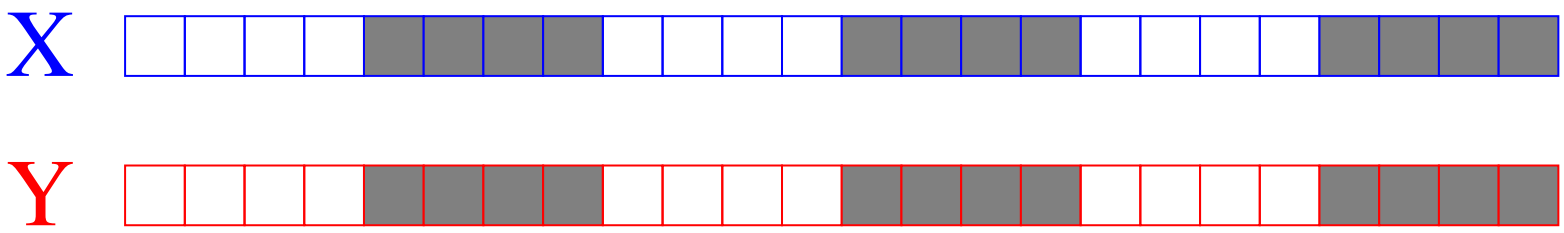
```
size_t N=1000;
std::vector<float> X(N);
std::vector<float> Y(N);
float a=2.f;

...

for (size_t i=0;i<N;i++){
  Y[i]+=a*X[i];
}
```

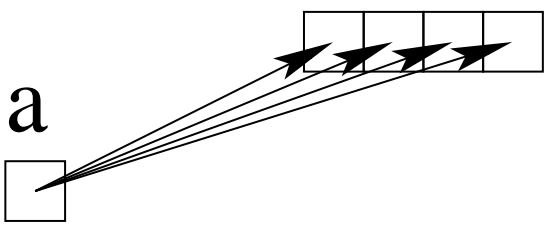
Exemple AXPY (AVX)



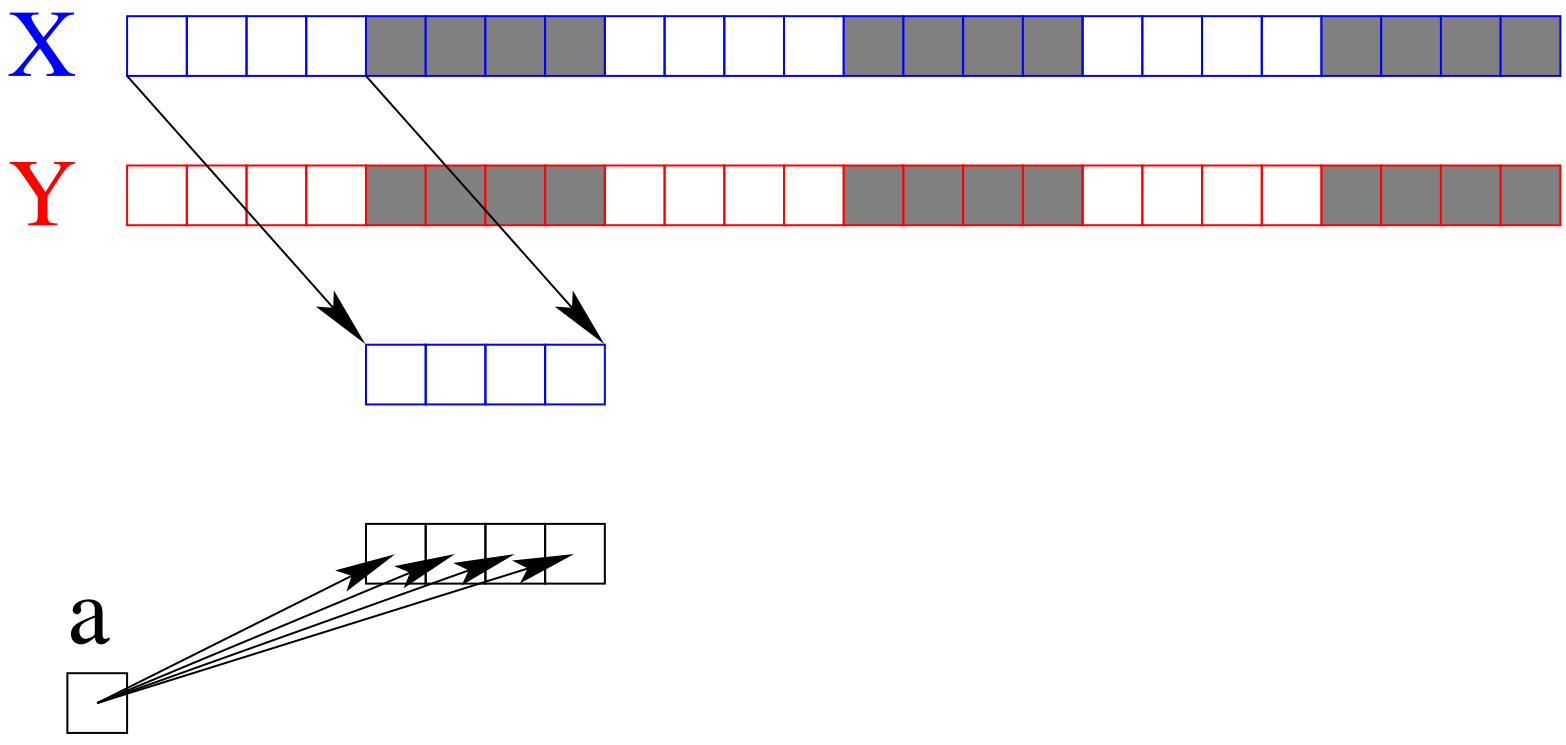
a

Exemple AXPY (AVX)

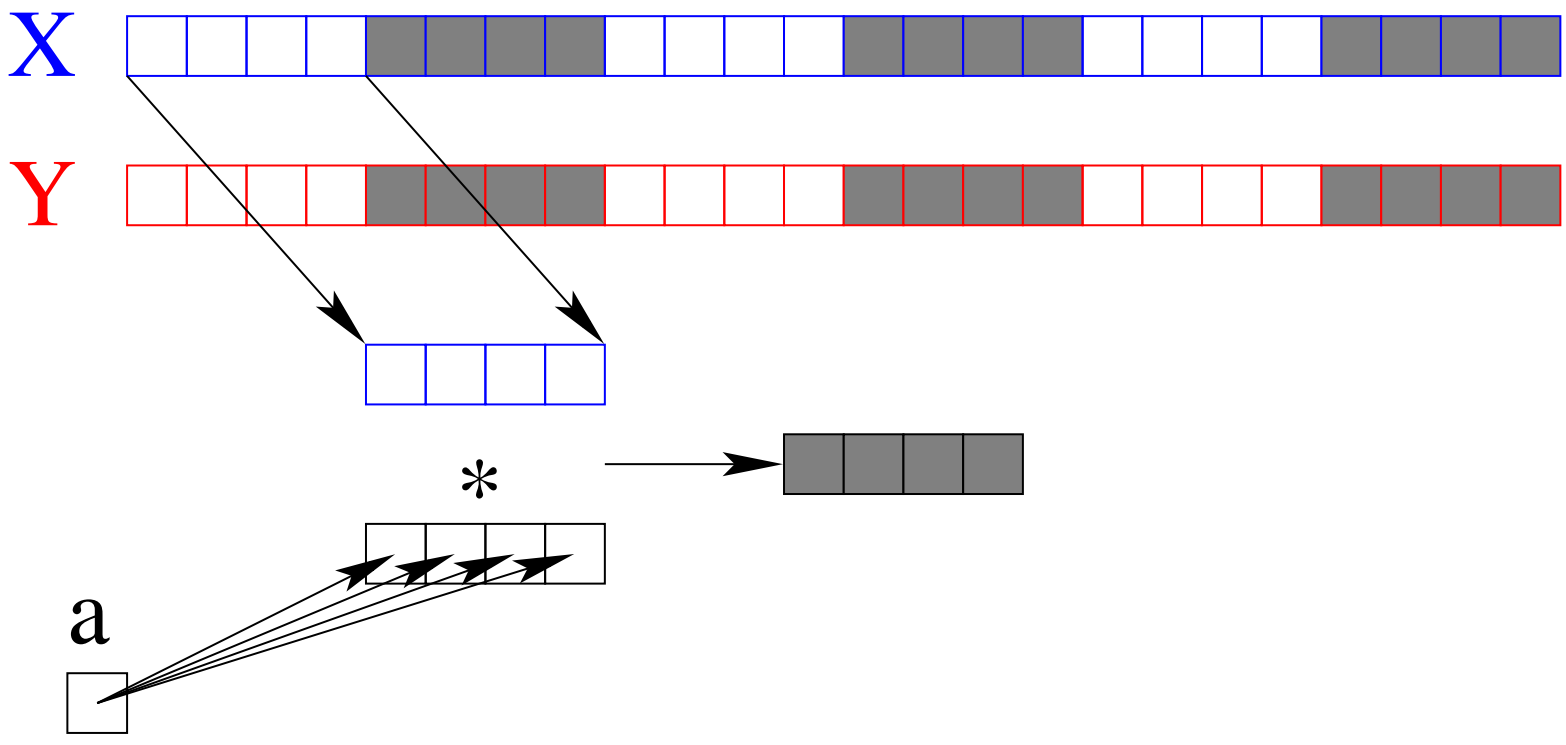




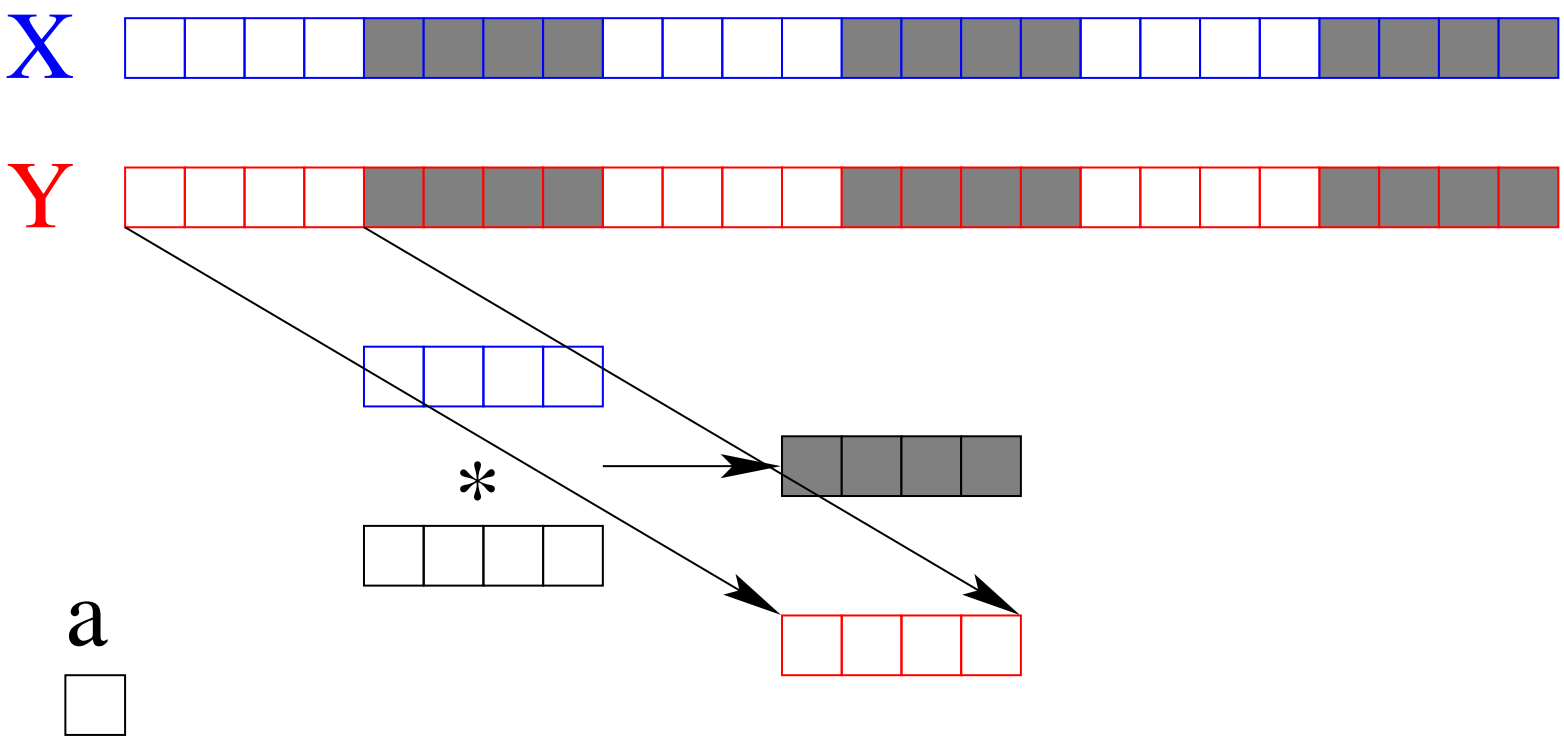
Exemple AXPY (AVX)



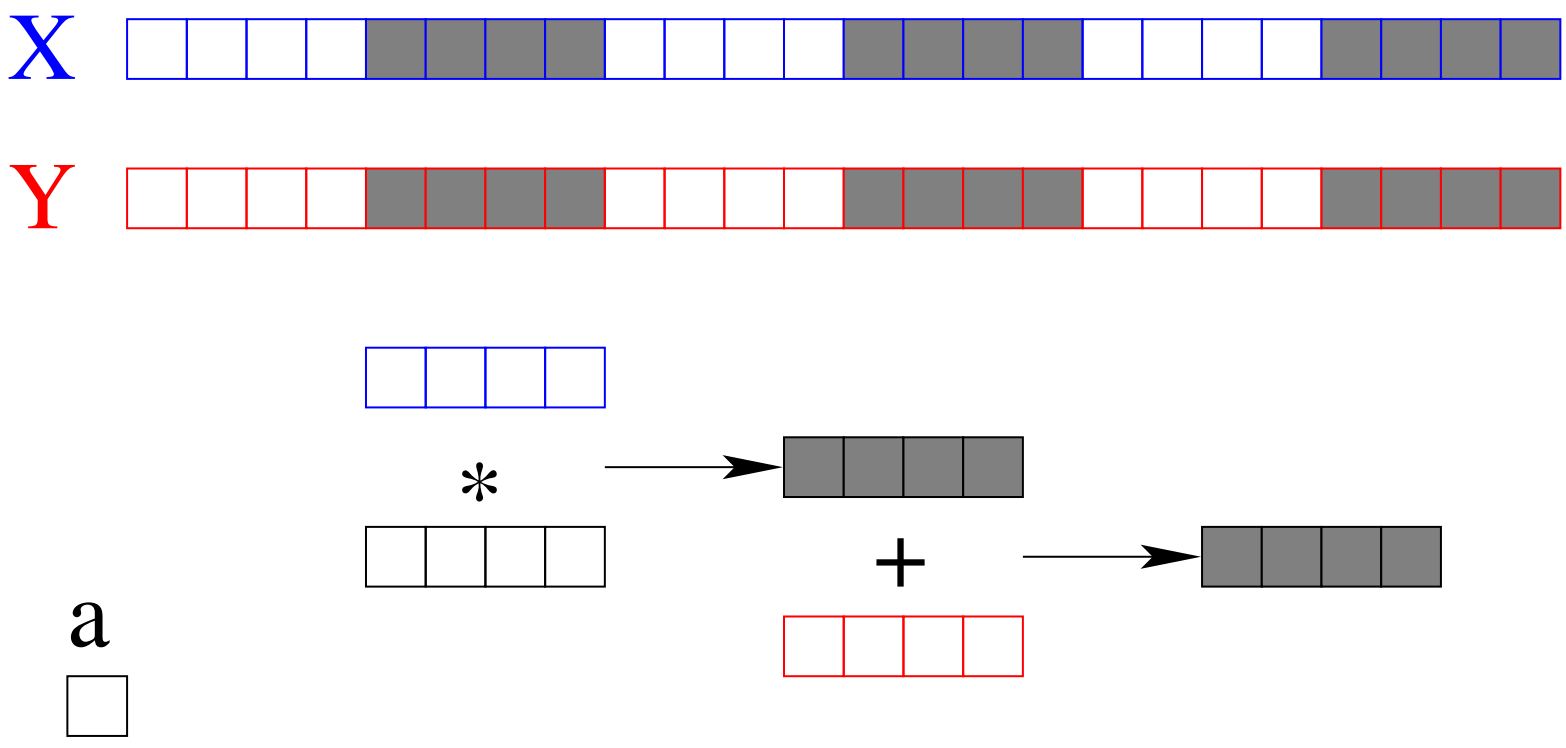
Exemple AXPY (AVX)



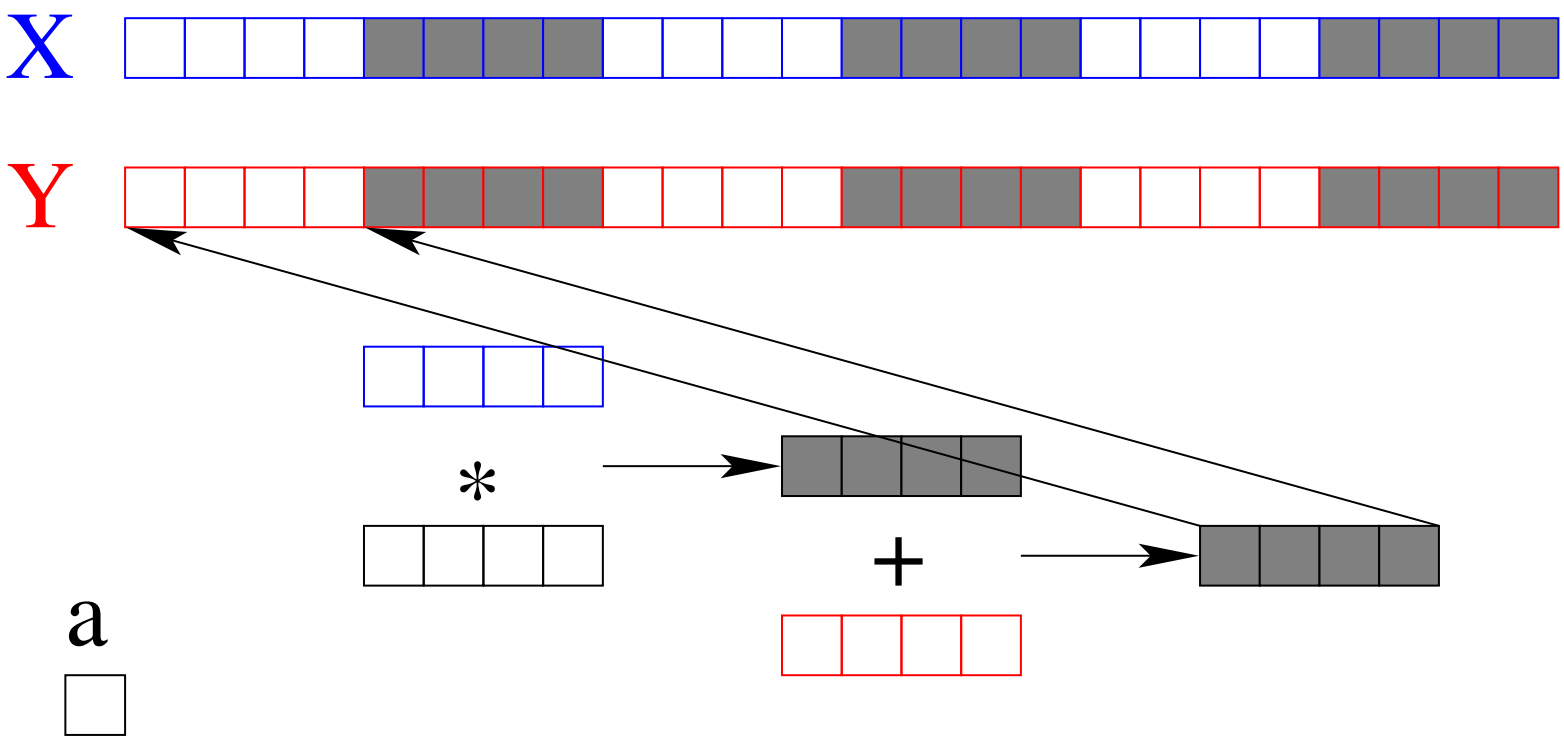
Exemple AXPY (AVX)



Exemple AXPY (AVX)



Exemple AXPY (AVX)



Exemple AXPY (C++)

```
void axpy_avx(int n, float alpha, const float ↩
    *x, float *y)
{
  __m256 v_alpha;
  int i;

  v_alpha = _mm256_broadcast_ss(&alpha);

  for(i=0; i+7<n; i+=8)
    _mm256_store_ps(y+i,
        _mm256_add_ps(_mm256_load_ps(y+i),
          _mm256_mul_ps(v_alpha,
            _mm256_load_ps(x+i))));

  for(; i<n; i++)
    y[i] += alpha * x[i];
}
```

Exemple AXPY (C++ Eigen)

http://eigen.tuxfamily.org

```
Eigen::VectorXf X(N),Y(N);
float a=2.0f

Y+=a*X;
```

- g++ -g -O3 -msse4.1 -march=native -DNDEBUG
- g++ -g -O3 -mavx -DNDEBUG
- g++ -g -O3 -march=native -DNDEBUG

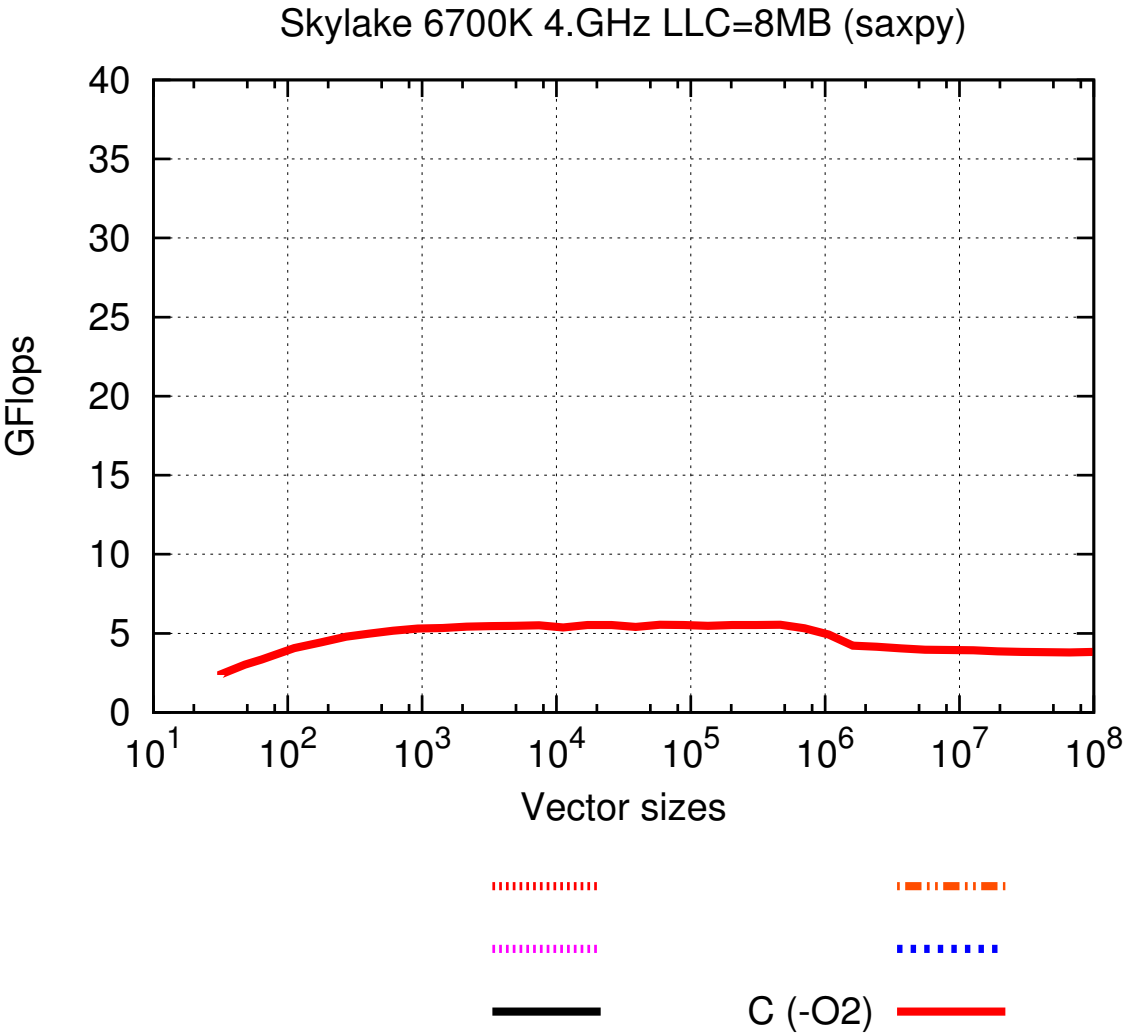
Exemple AXPY (C++ Eigen Map)

```
std::vector<float> Xu(N),Yu(N);

typedef Eigen::Map<Eigen::VectorXf> MapVector;

MapVector X(&Xu[0],N),Y(&Yu[0],N);

float a=2.0f

Y+=a*X;
```

Exemple AXPY (C++ Eigen Map Align)

```
typedef tbb::cache_aligned_allocator<float> ↩
    Allocator;

std::vector<float,Allocator> Xu(N),Yu(N);

Eigen::Map<Eigen::VectorXf,Eigen::Aligned> ↩
    X(&Xu[0],N),Y(&Yu[0],N);

  float a=2.0f
  Y+=a*X;
```

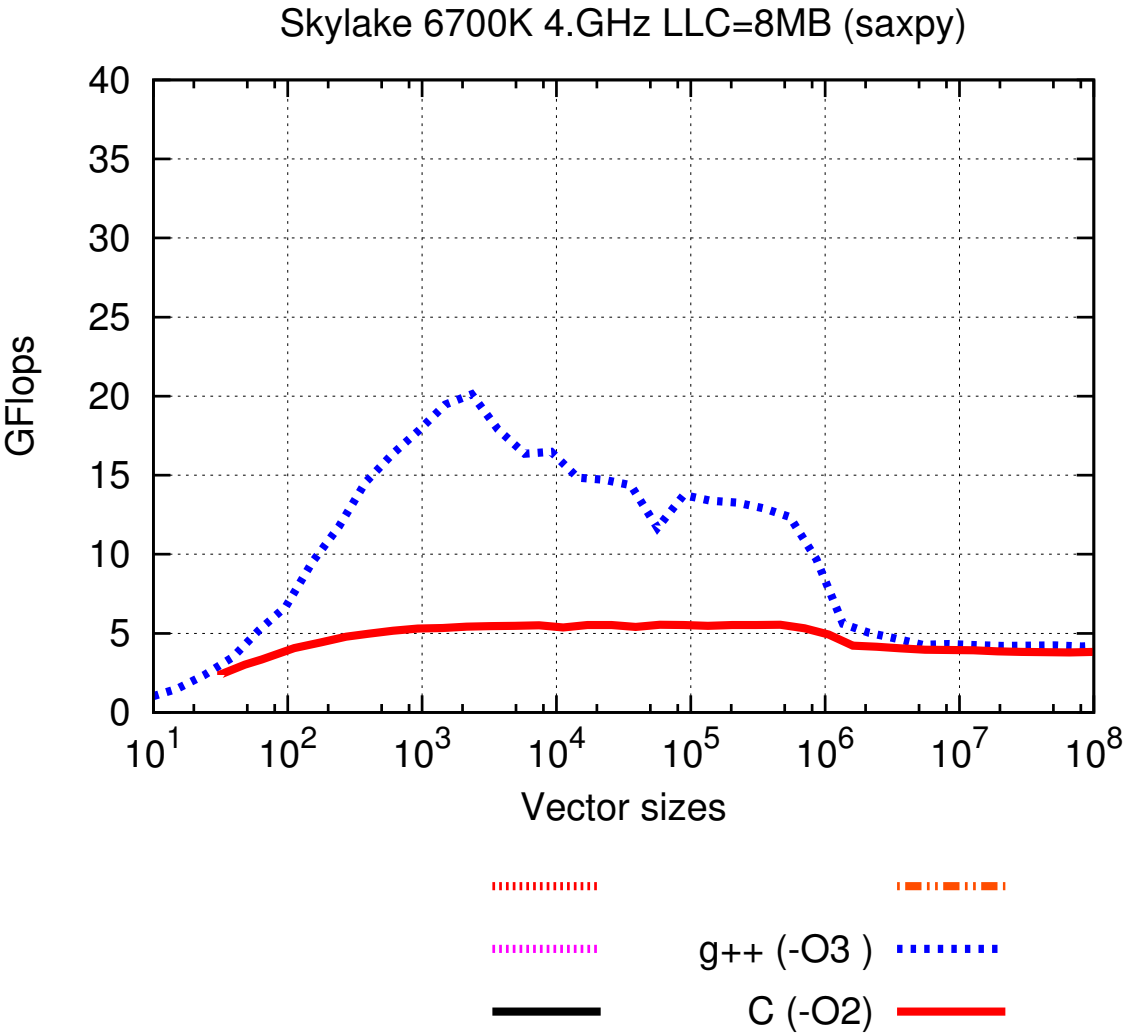
SAXPY SIMD




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1
- -O2

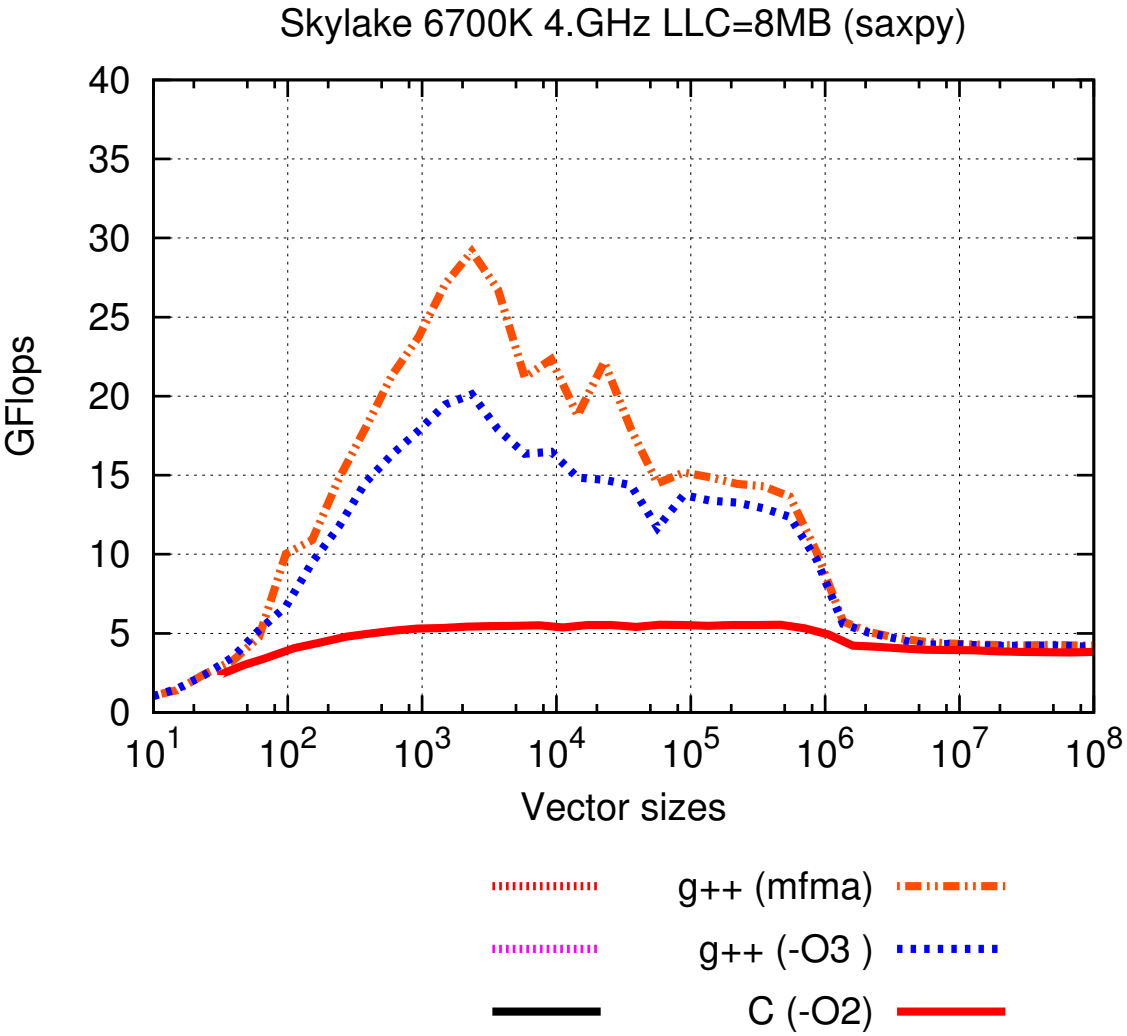
SAXPY SIMD




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1
- -O3

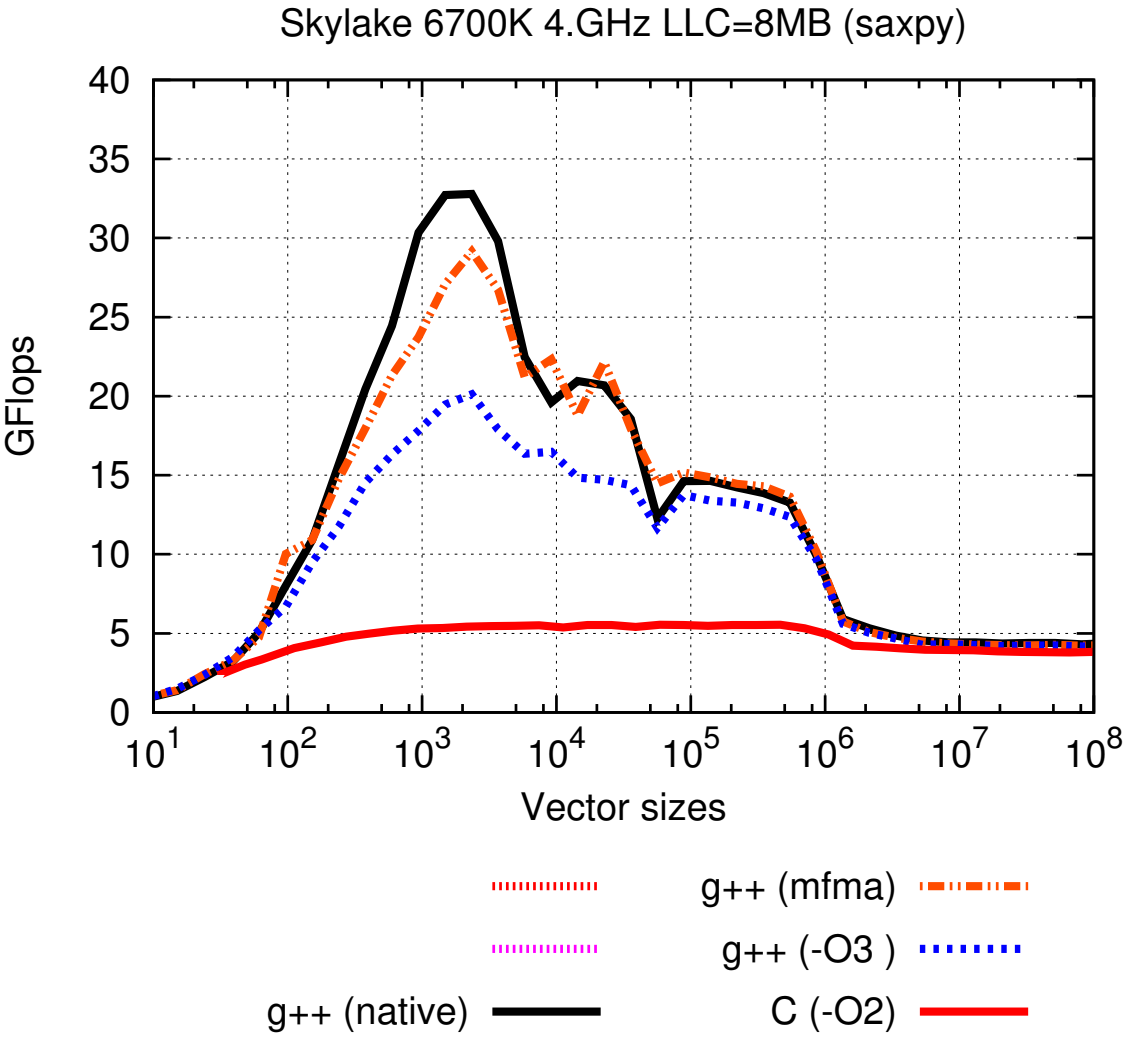
SAXPY SIMD




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1
- -O3 -mfma

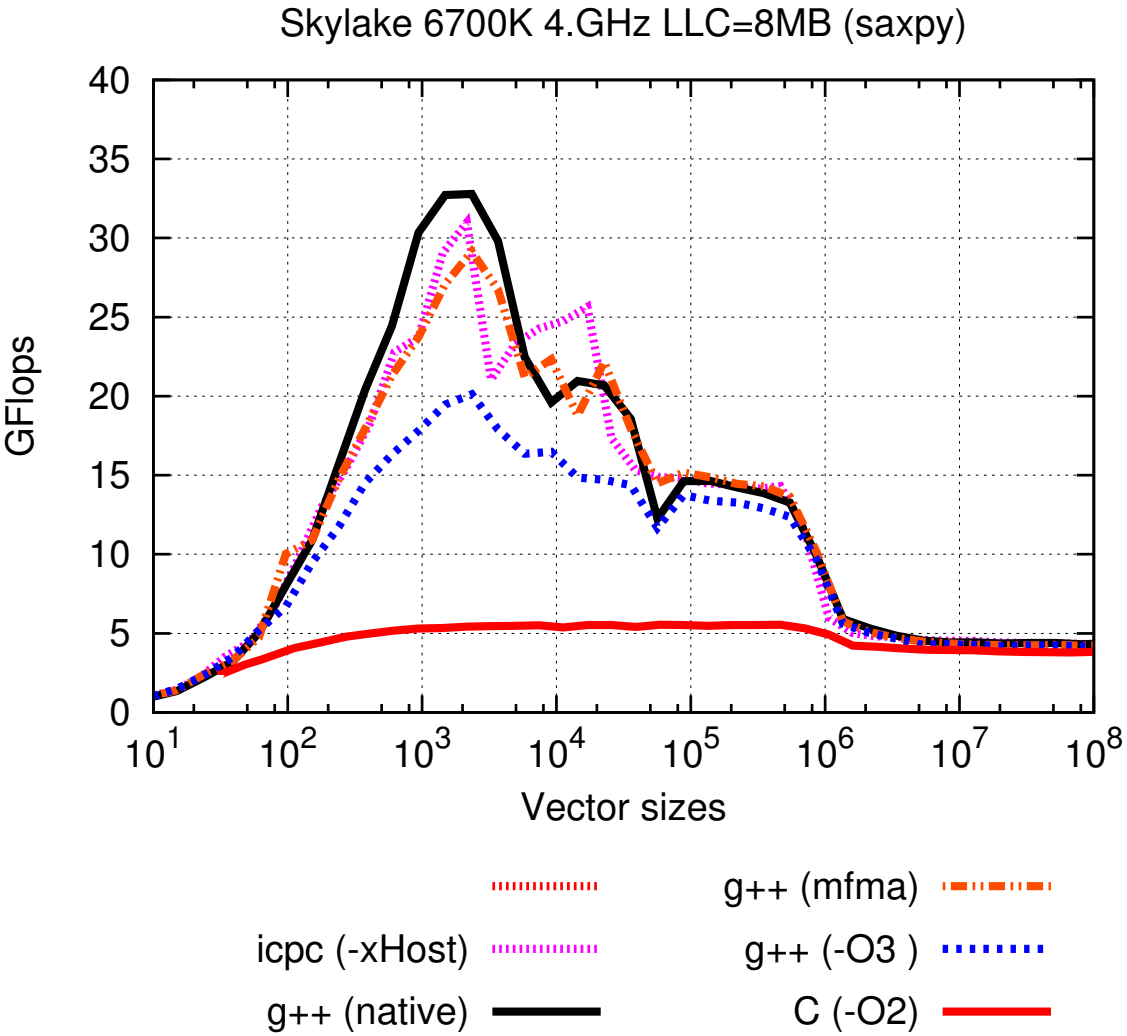
SAXPY SIMD




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- g++6.1
- -O3 -march=native

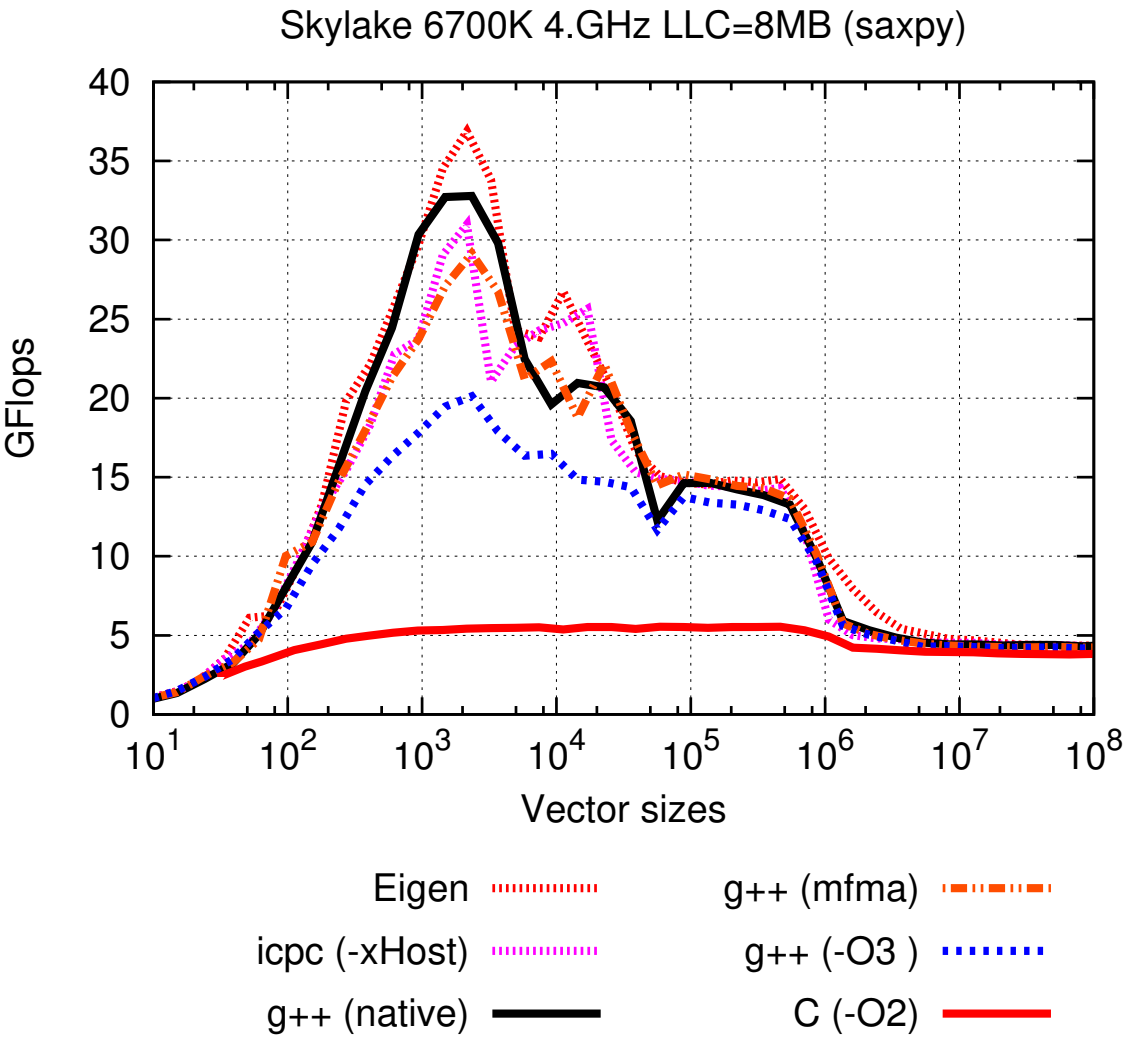
SAXPY SIMD




```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- `icpc17 -XHost`

SAXPY SIMD

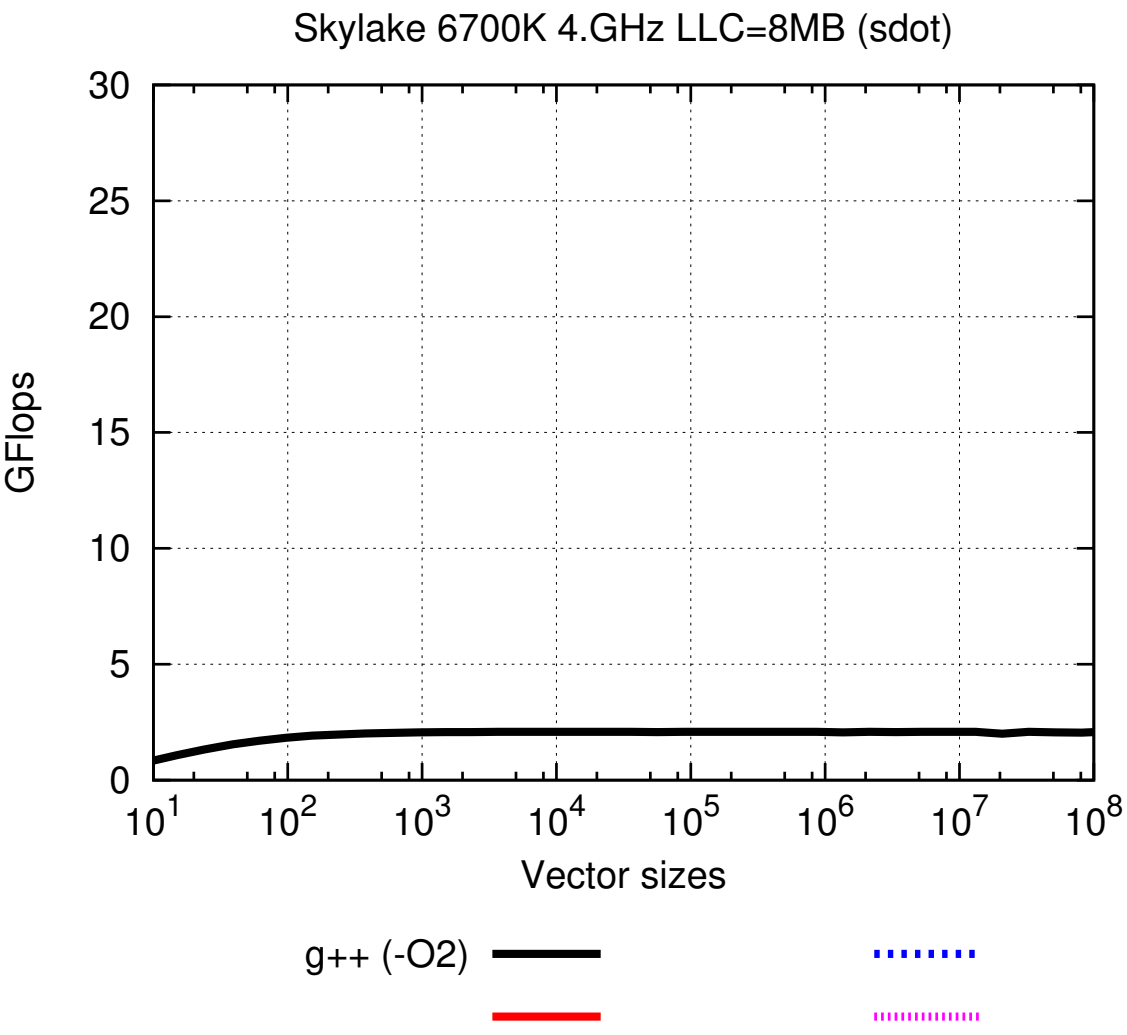



```
for (int i=0;i<N;i++){
   Y[i]+=coef*X[i];
}
```

- Eigen3.3

  ```
  Y+=coef*X;
  ```

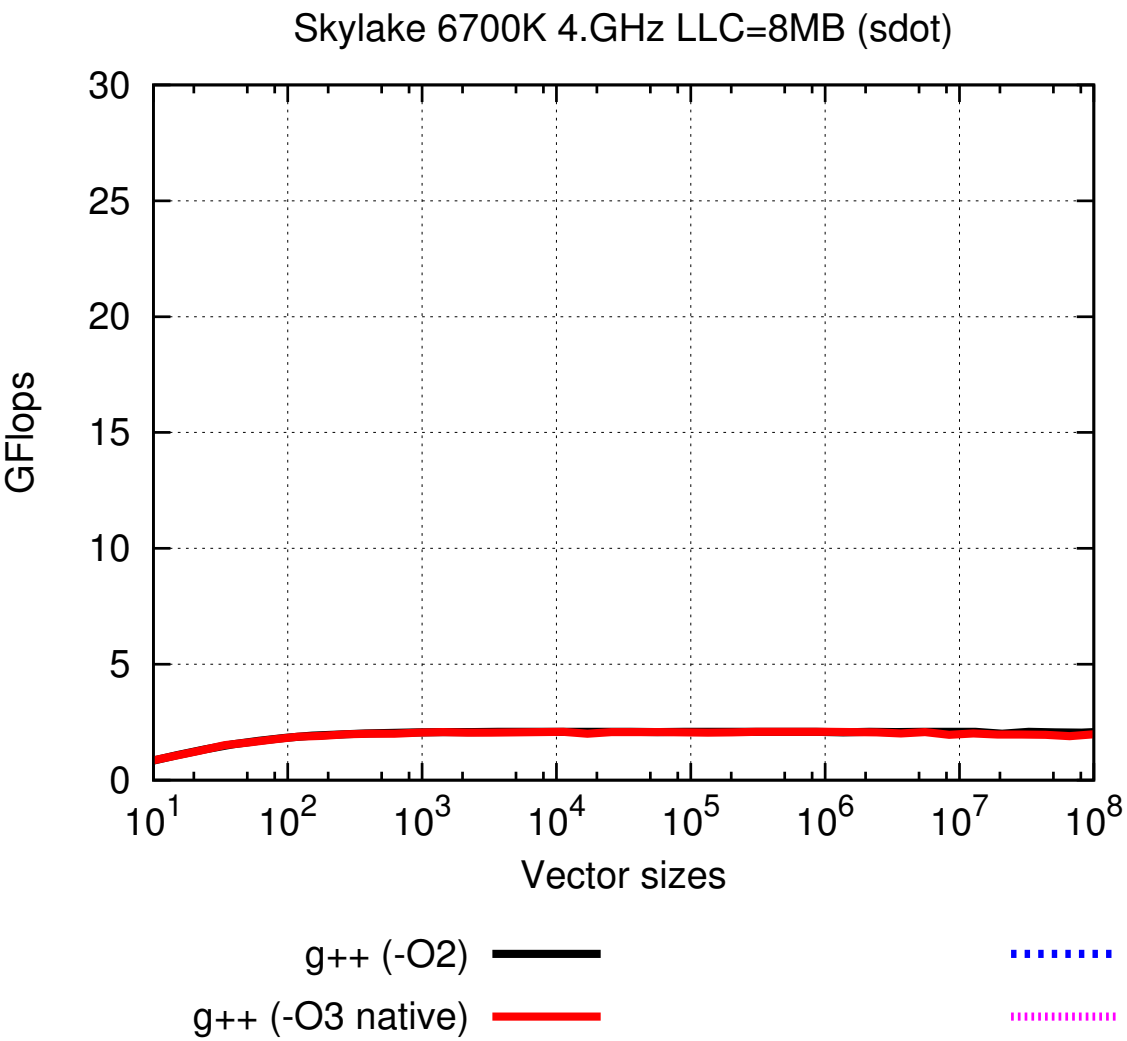
SDOT SIMD




```
for (int i=0;i<N;i++){
   s+=X[i]*Y[i];
}
```

- g++6.1
- -O2

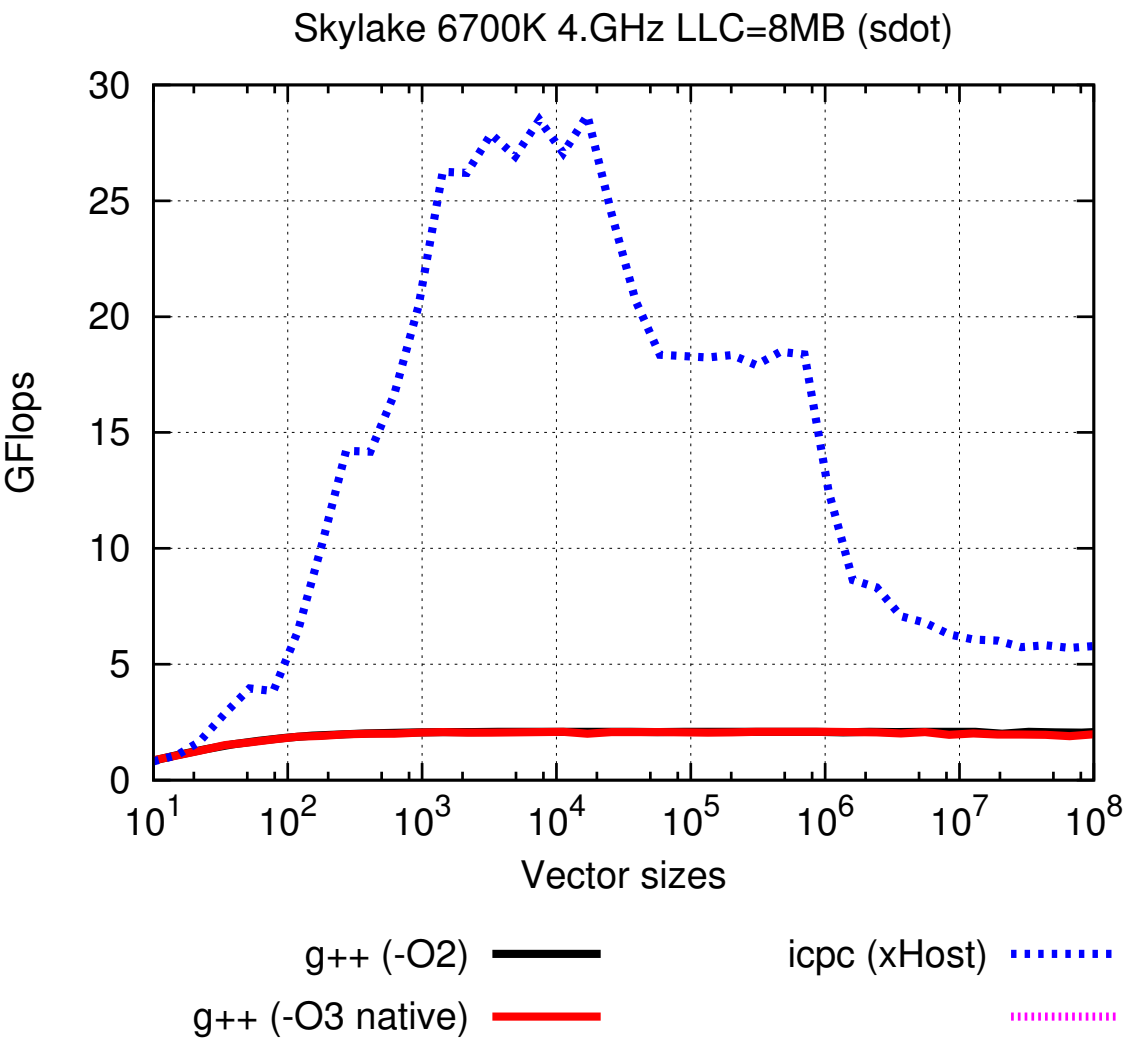
SDOT SIMD




```
for (int i=0;i<N;i++){
   s+=X[i]*Y[i];
}
```

- g++6.1
- -O3 -march=native

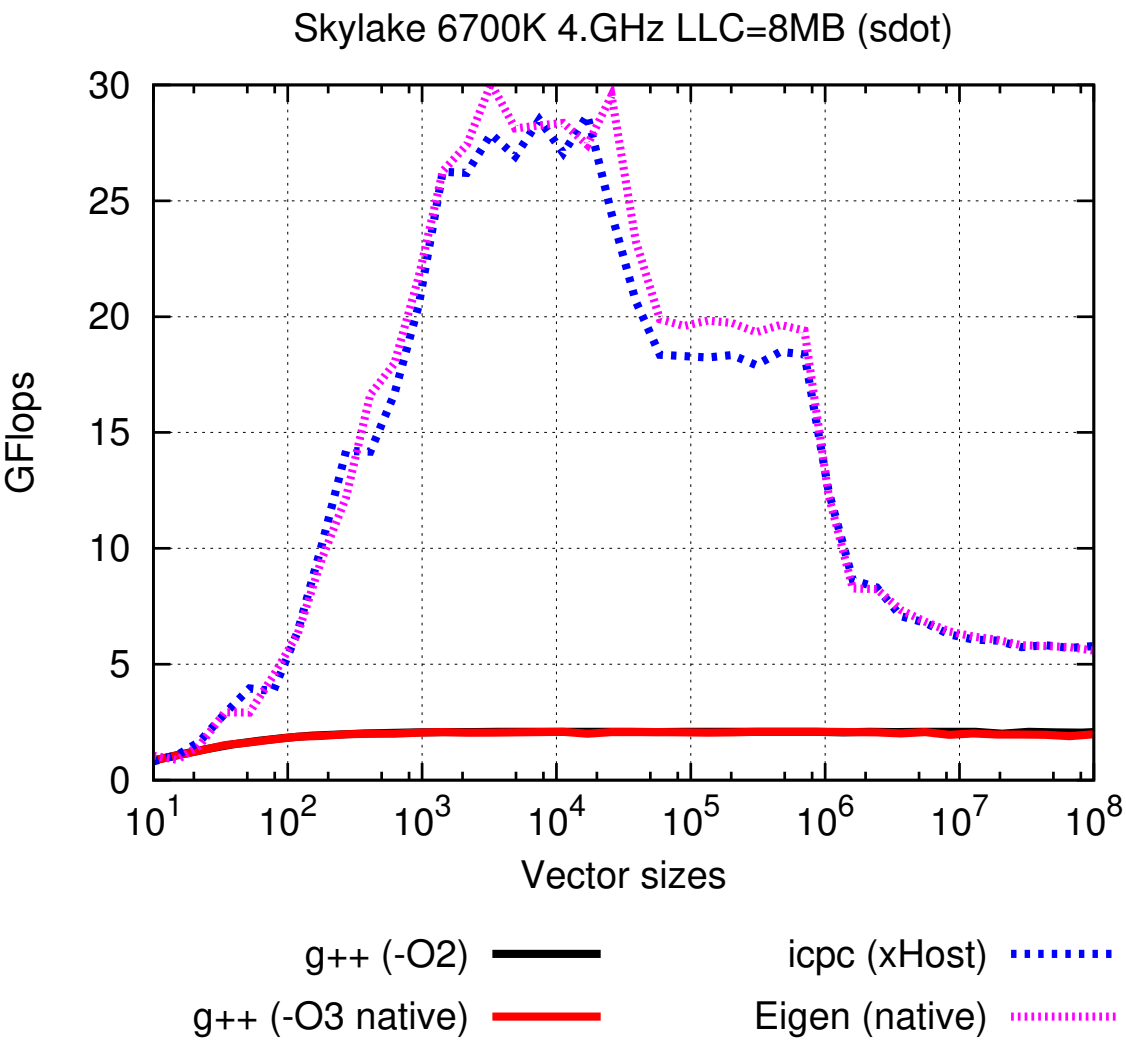
SDOT SIMD




```
for (int i=0;i<N;i++){
   s+=X[i]*Y[i];
}
```

- `icpc17 -XHost`

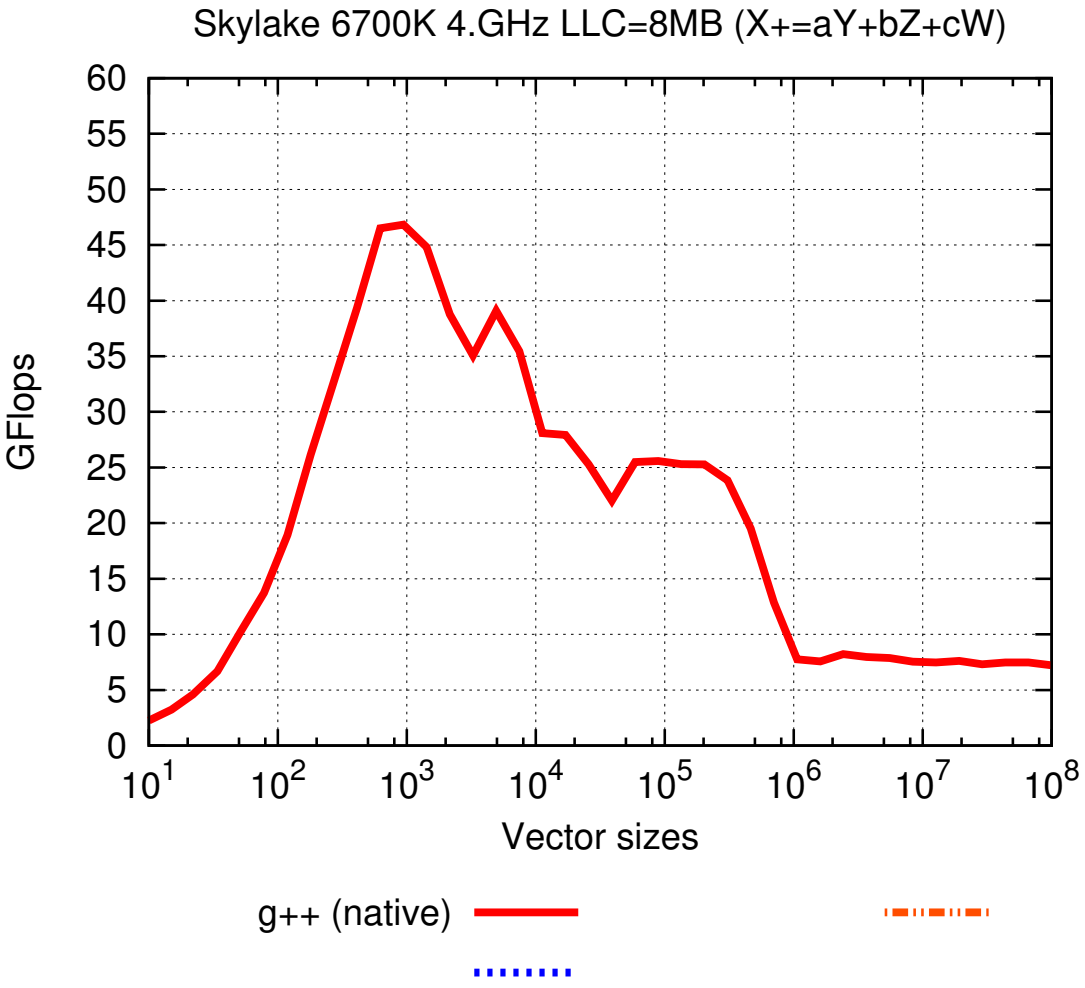
SDOT SIMD




```
for (int i=0;i<N;i++){
   s+=X[i]*Y[i];
}
```

- Eigen3.3
  ```
  s=X.dot(Y);
  ```

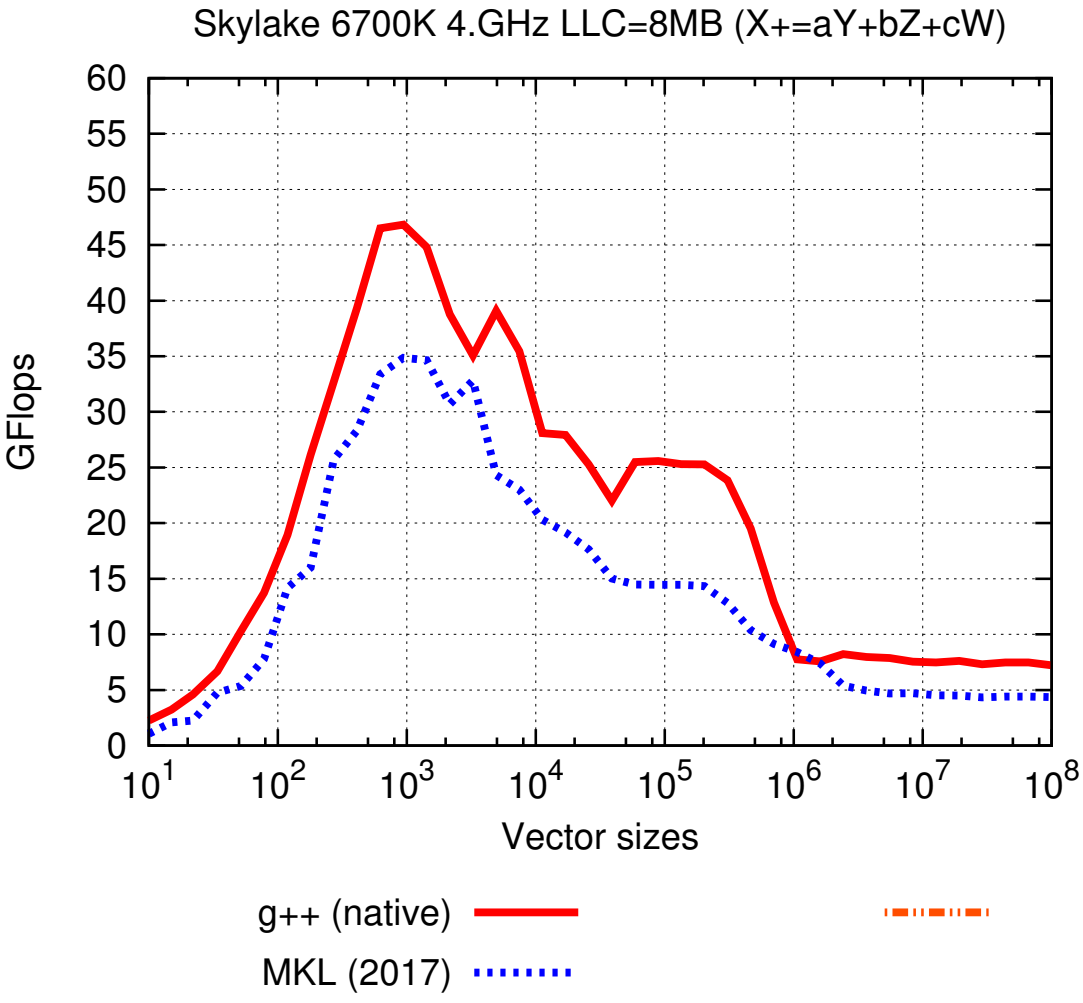
Expression Template with Eigen X+ = aY+bZ+cW




```
for (int i=0;i<N;i++){
   X[i]+=a*Y[i]+b*Z[i]+c*W[i];
}
```

- g++6.1
- -O3 -march=native

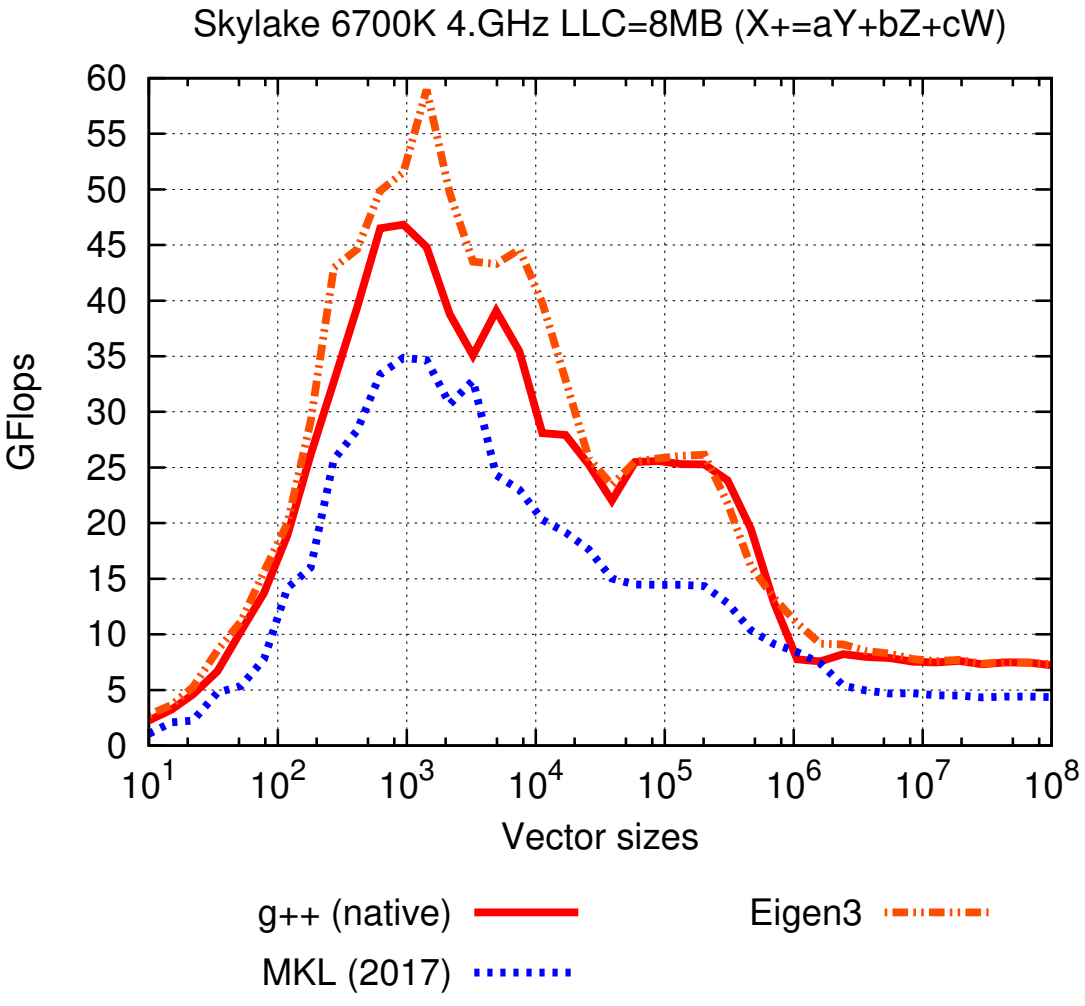
Expression Template with Eigen X+ = aY+bZ+cW




```
for (int i=0;i<N;i++){
    X[i]+=a*Y[i]+b*Z[i]+c*W[i];
}
```

- MKL (2017)

Expression Template with Eigen X+ = aY+bZ+cW




```
for (int i=0;i<N;i++){
   X[i]+=a*Y[i]+b*Z[i]+c*W[i];
}
```

- Eigen3.3
  `X+=a*Y+b*Z+c*W`

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).
- Choisir son outil :

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).
- Choisir son outil :
  - assembleur (+ anti-depresseurs)

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).
- Choisir son outil :
  - assembleur (+ anti-depresseurs)
  - intrinsics (+ aspirine)

Résumé

Pour vectoriser efficacement il faut :

- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).
- Choisir son outil :
  - assembleur (+ anti-depresseurs)
  - intrinsics (+ aspirine)
  - compilateur performant (+ boucles triviales)

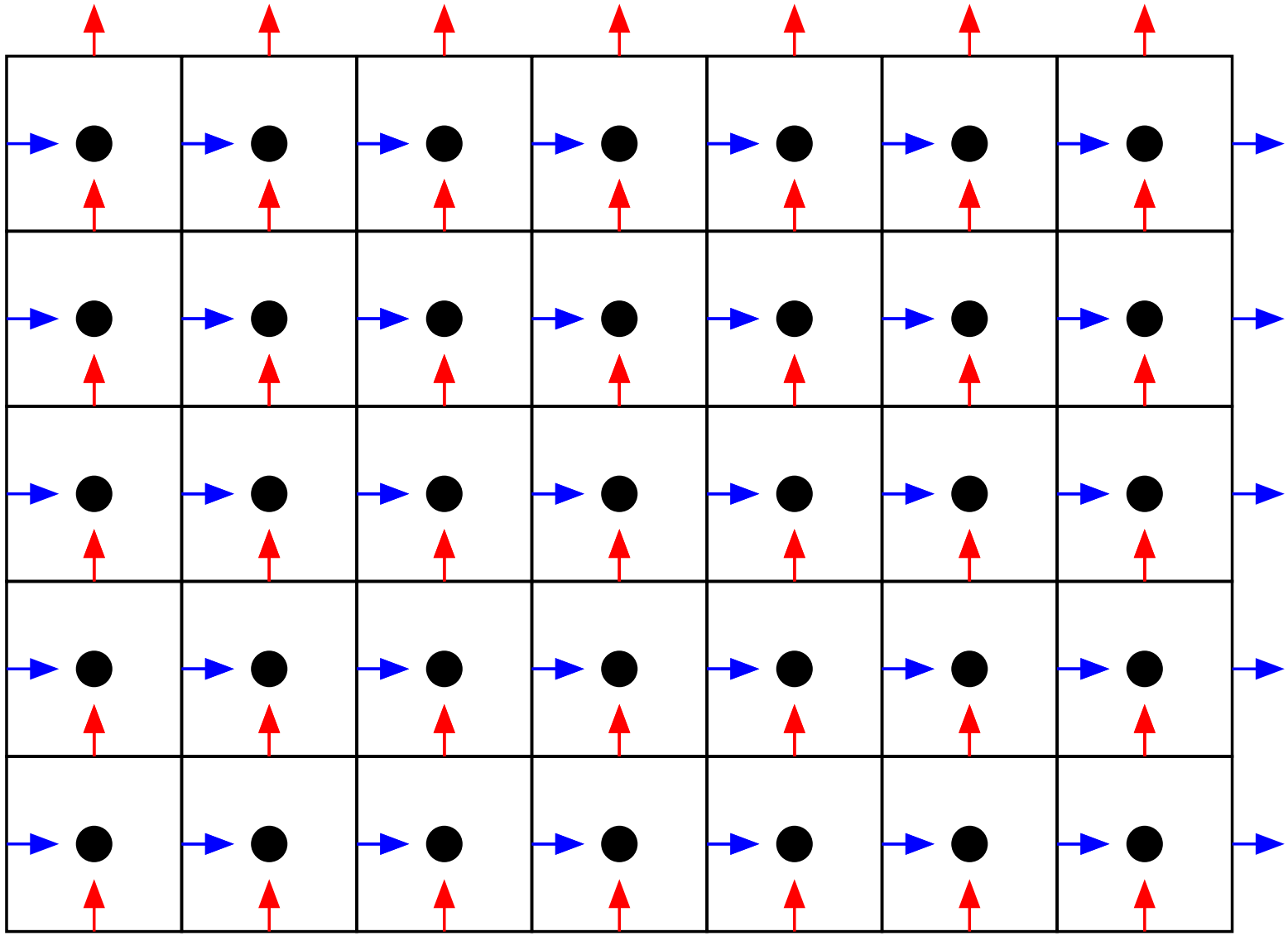
Résumé

Pour vectoriser efficacement il faut :

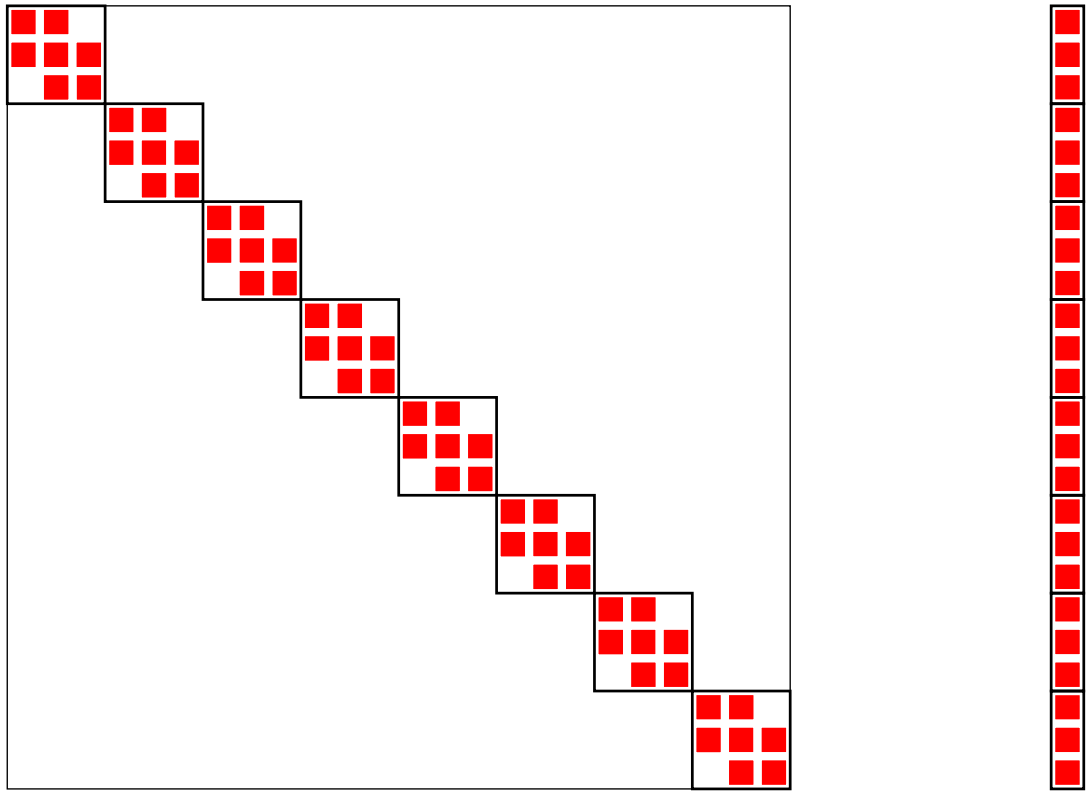
- Trouver du parallélisme SIMD
- Aligner les ballons (données) en face des joueurs (unités SIMD).
- Avoir un bonne localité des données spatiale et temporelle (intensité arithmétique).
- Choisir son outil :
  - assembleur (+ anti-depresseurs)
  - intrinsics (+ aspirine)
  - compilateur performant (+ boucles triviales)
  - bibliothèque adaptée (en C++ Eigen, boost : :simd,..)

Multi-Tridiagonal (ADI)

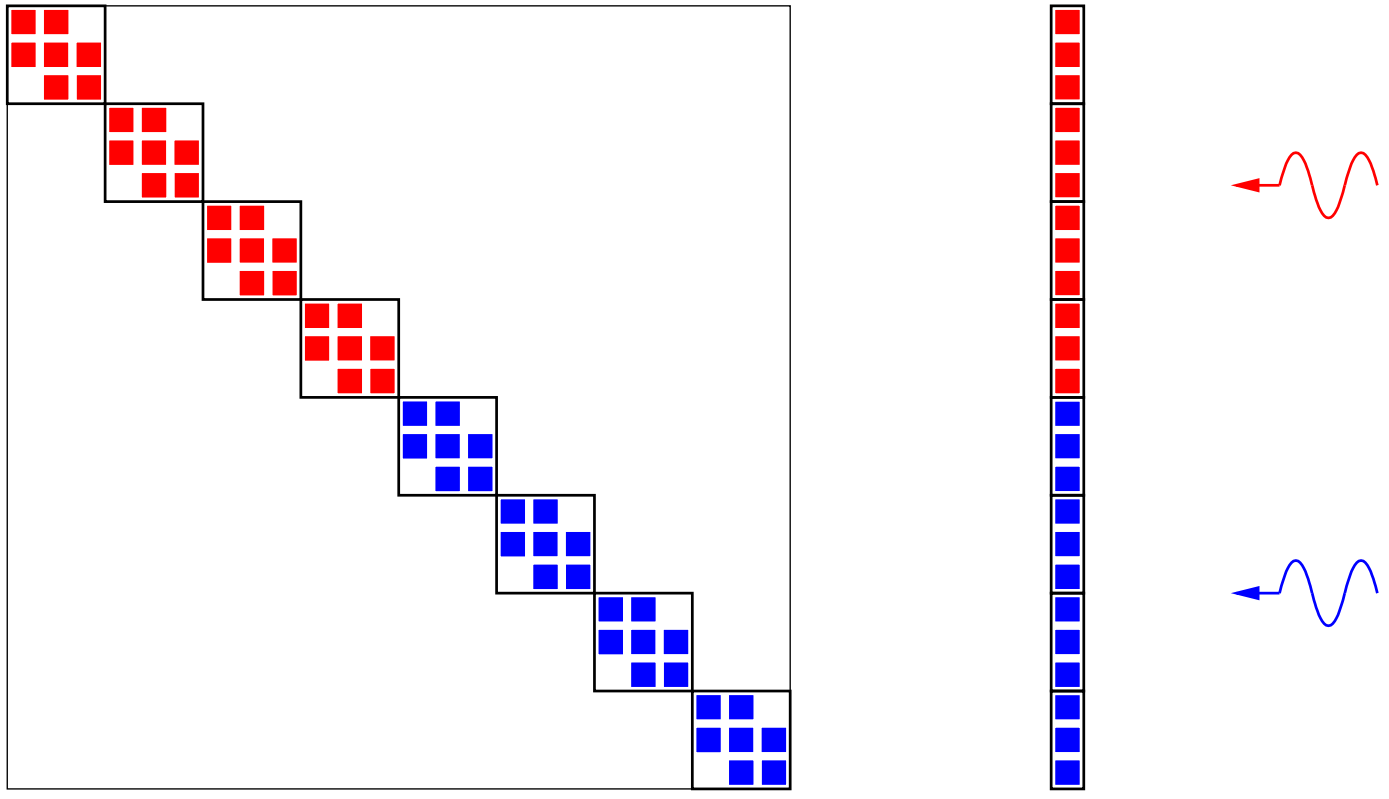
On part d'un code parallèle SMP qui est memory bound.



Matrice Diagonal<TriDiagonal>

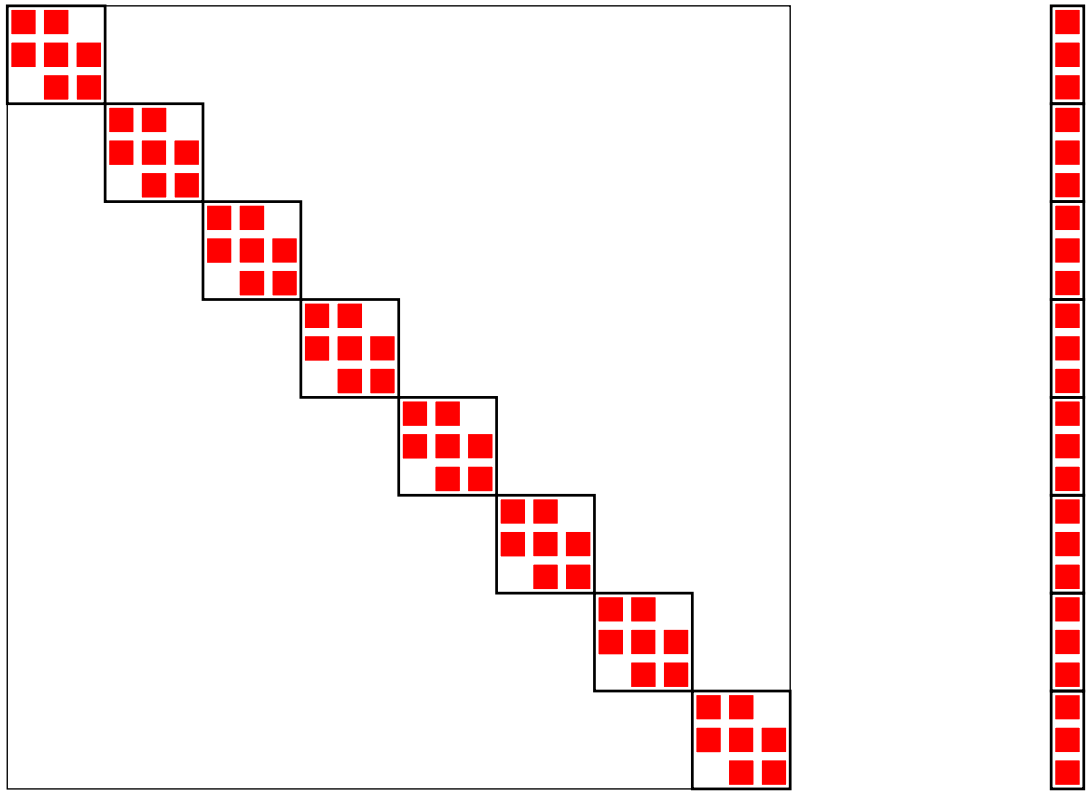


Matrice Diagonal<TriDiagonal>



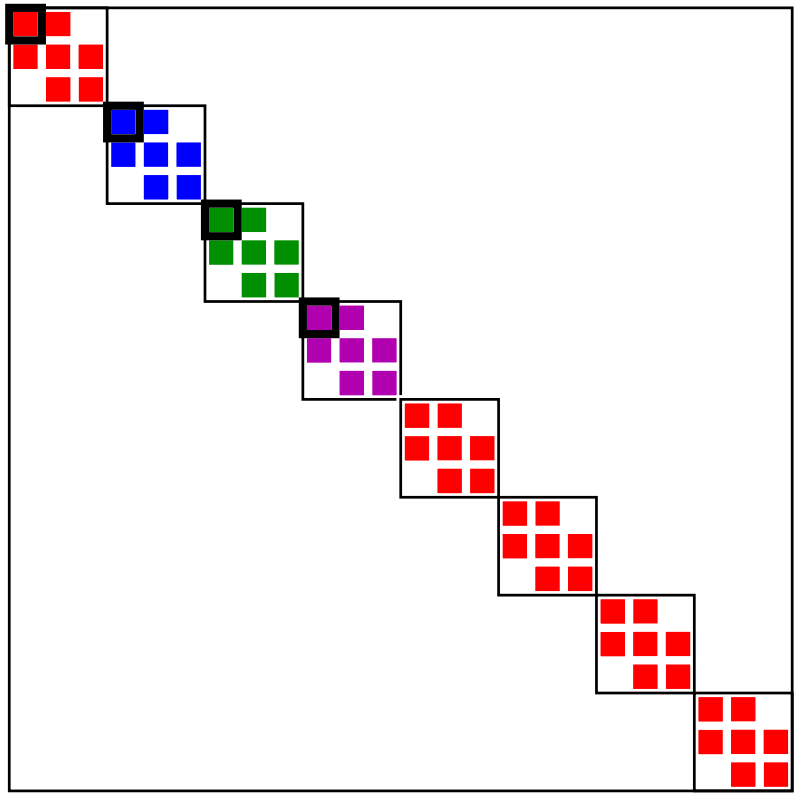
AX=B→Parallélisme à 2 coeurs.

Matrice Diagonal<TriDiagonal>



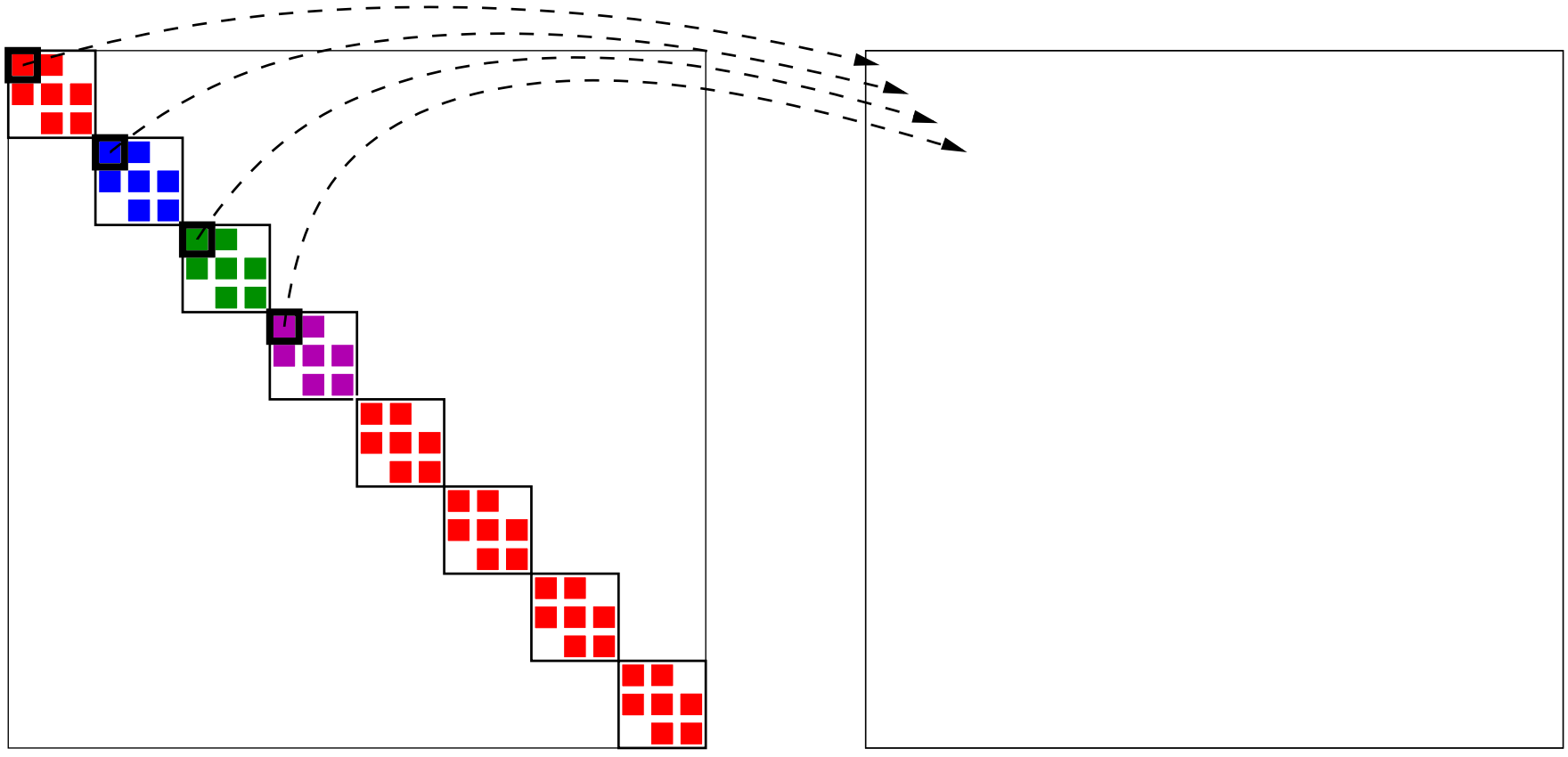
AX=B→Parallélisme SIMD (4 opérations simultanées).

Matrice Diagonal<TriDiagonal>



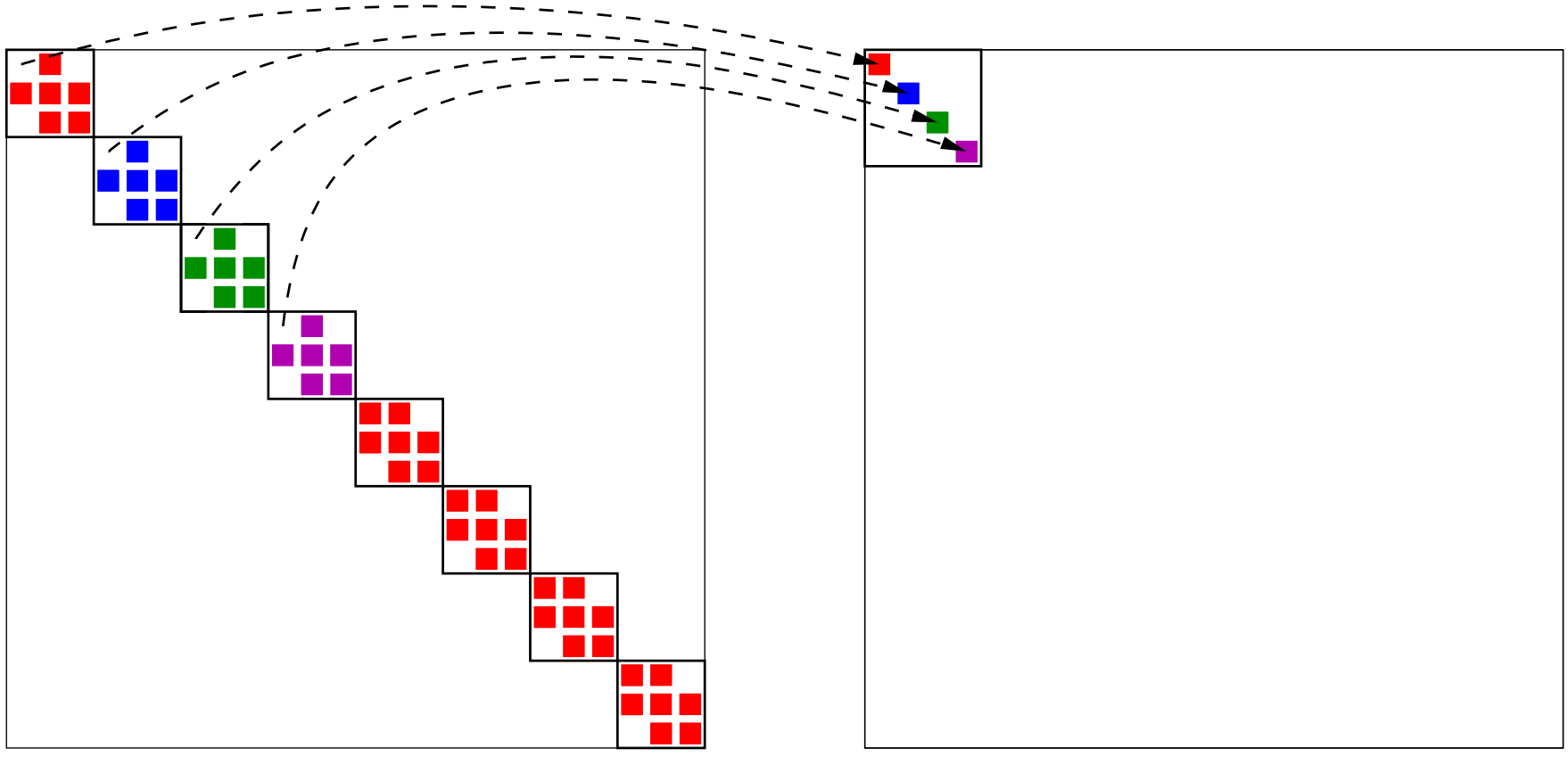
On groupe les blocs diagonaux 4 par 4.

Matrice Diagonal<TriDiagonal>



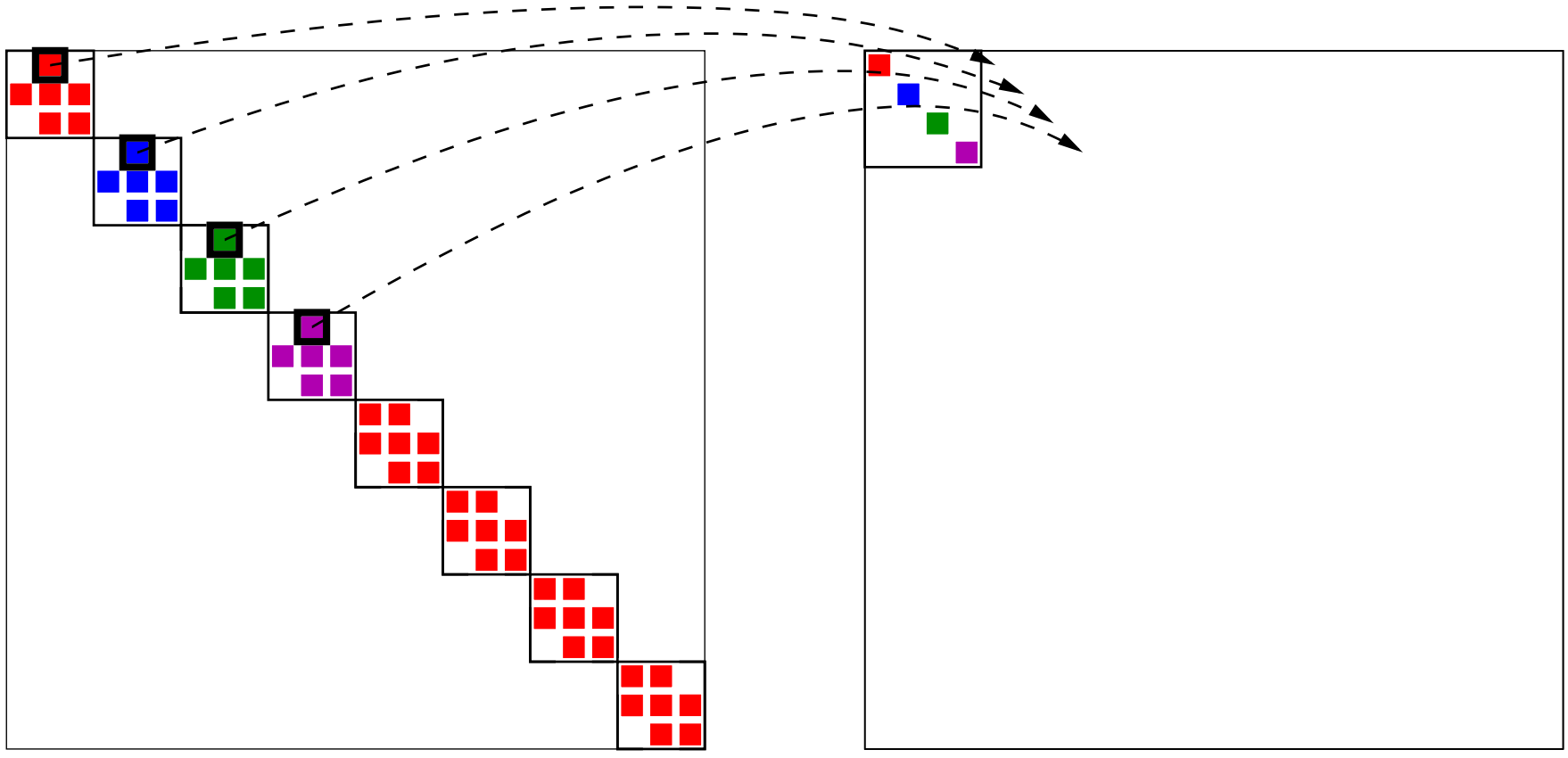
On groupe les blocs diagonaux 4 par 4.

Matrice Diagonal<TriDiagonal>



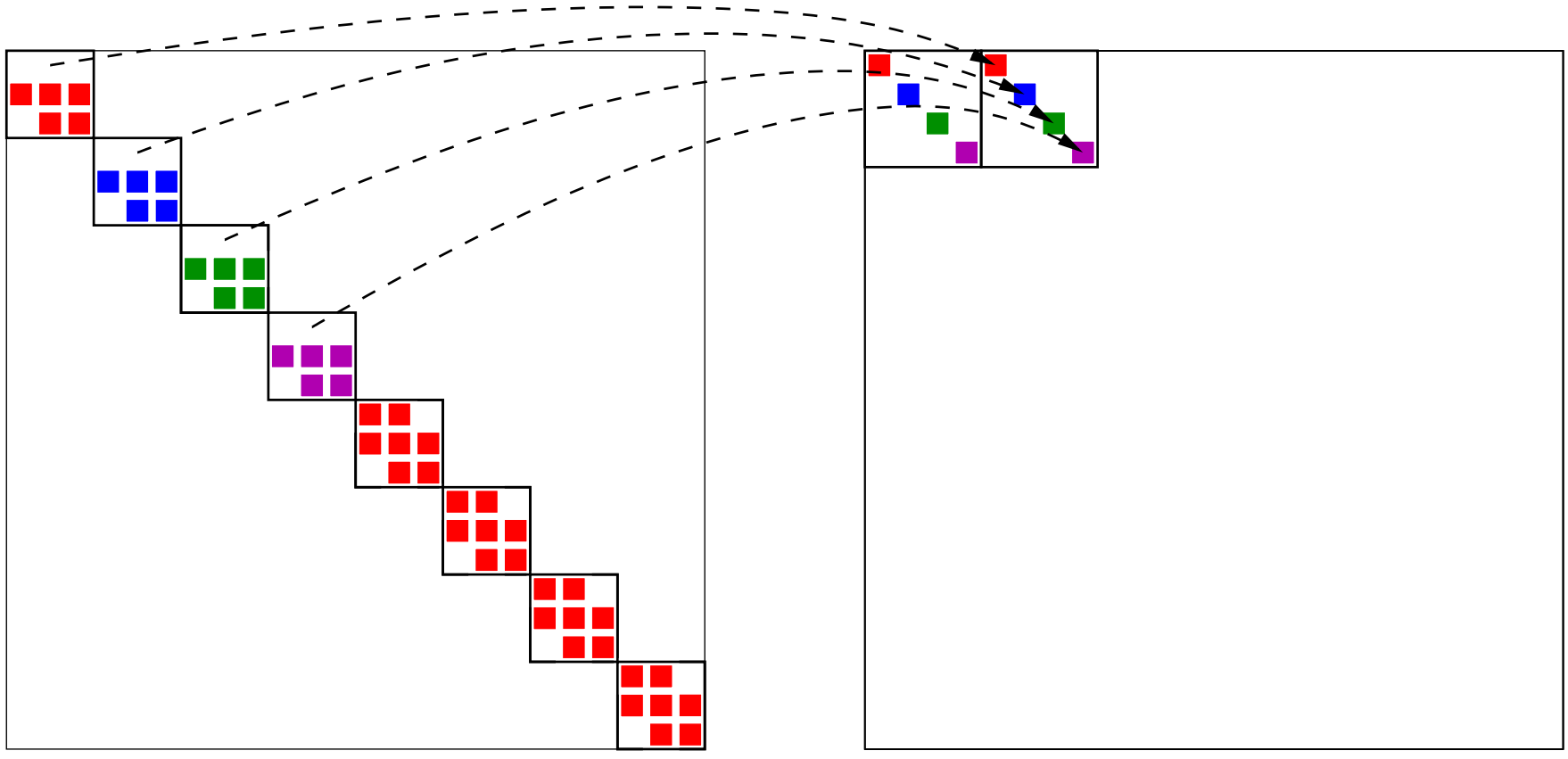
On groupe les blocs diagonaux 4 par 4.

Matrice Diagonal<TriDiagonal>



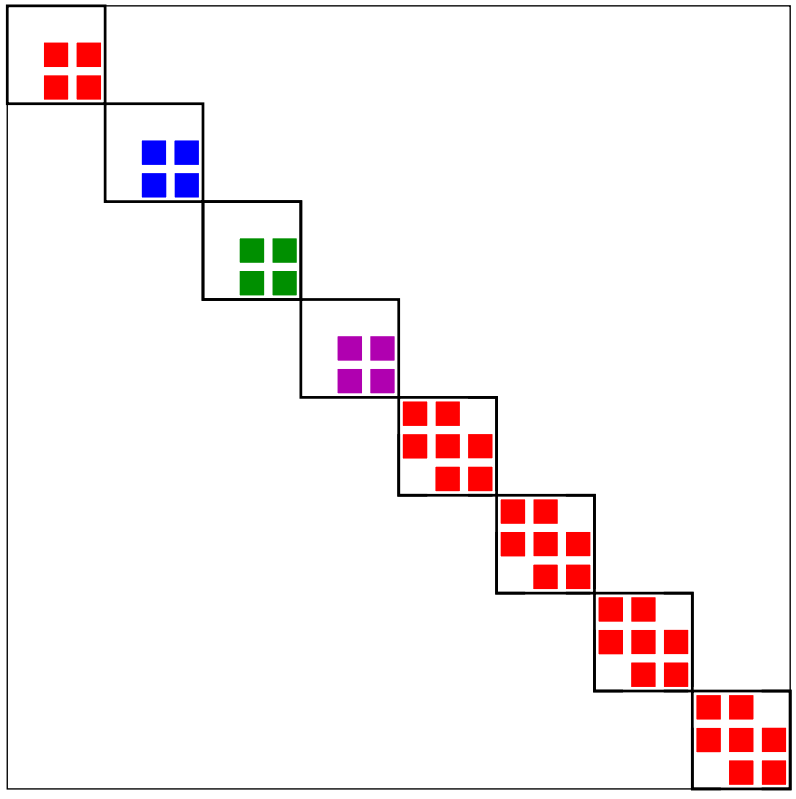
On groupe les blocs diagonaux 4 par 4.

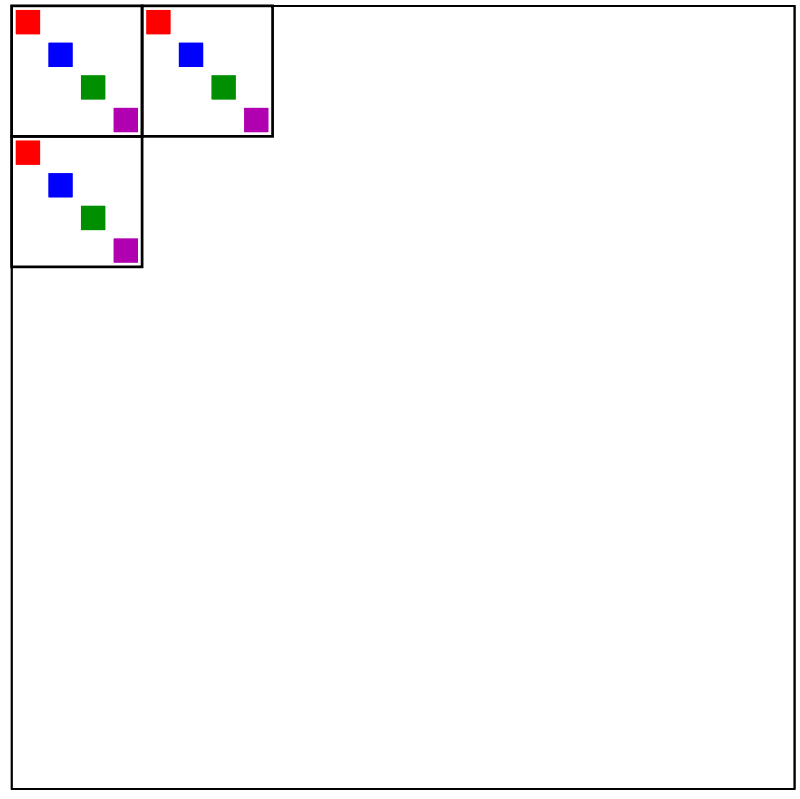
Matrice Diagonal<TriDiagonal>



On groupe les blocs diagonaux 4 par 4.

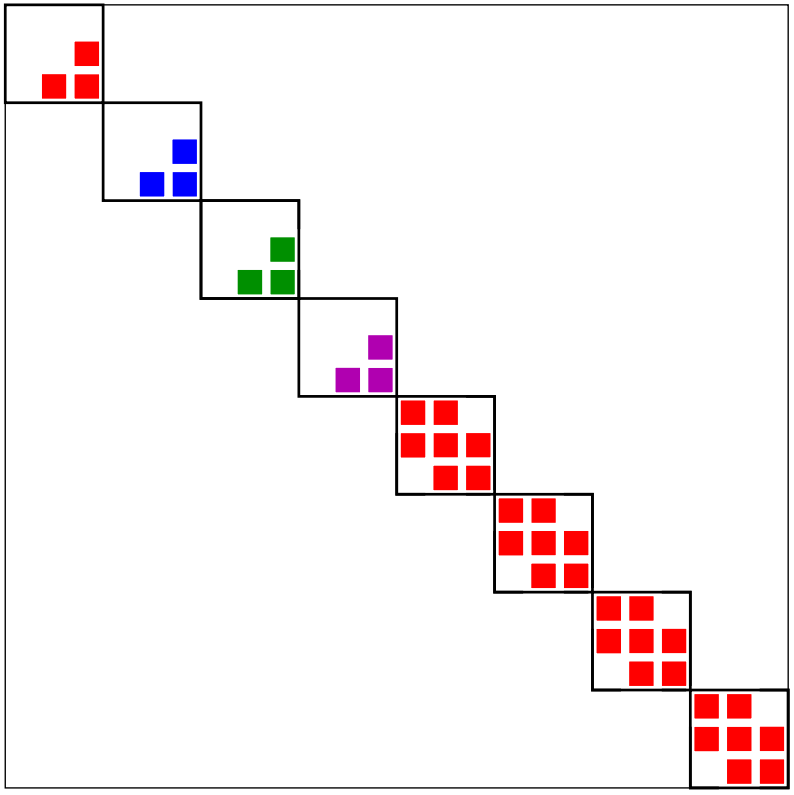
Matrice Diagonal<TriDiagonal>

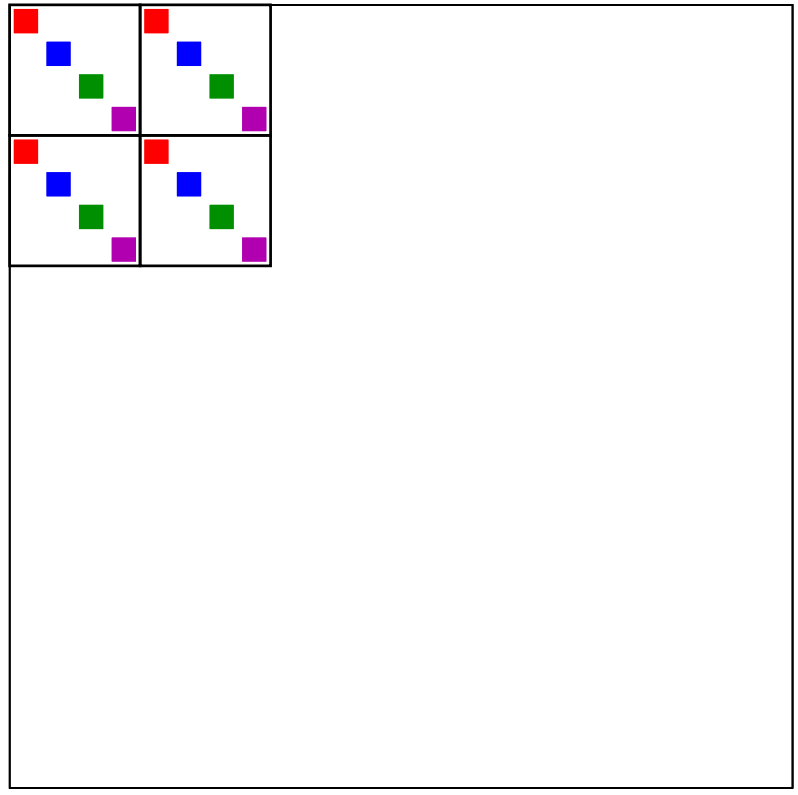




On groupe les blocs diagonaux 4 par 4.

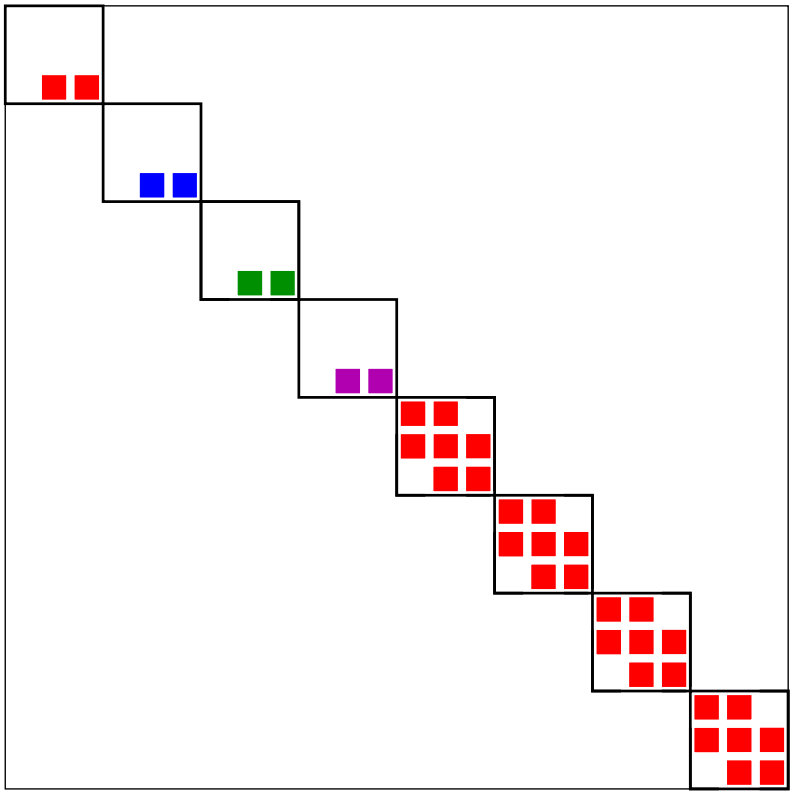
Matrice Diagonal<TriDiagonal>

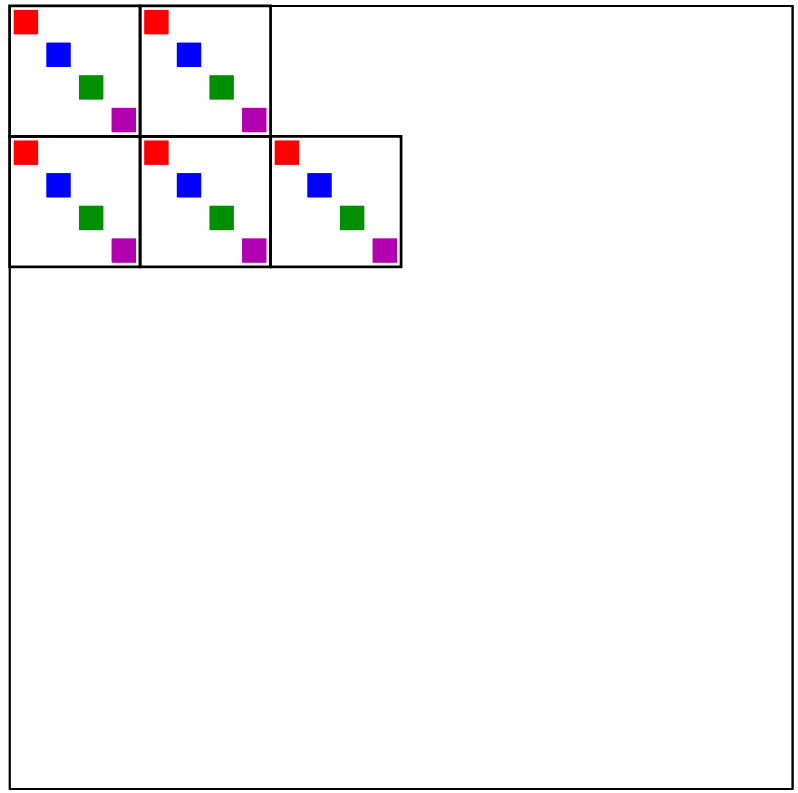




On groupe les blocs diagonaux 4 par 4.

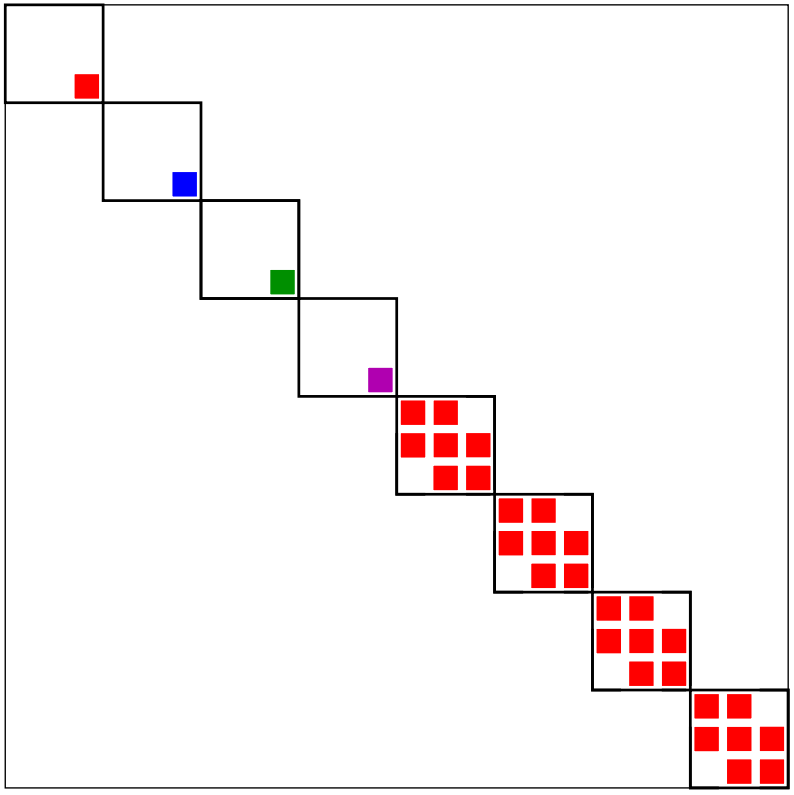
Matrice Diagonal<TriDiagonal>

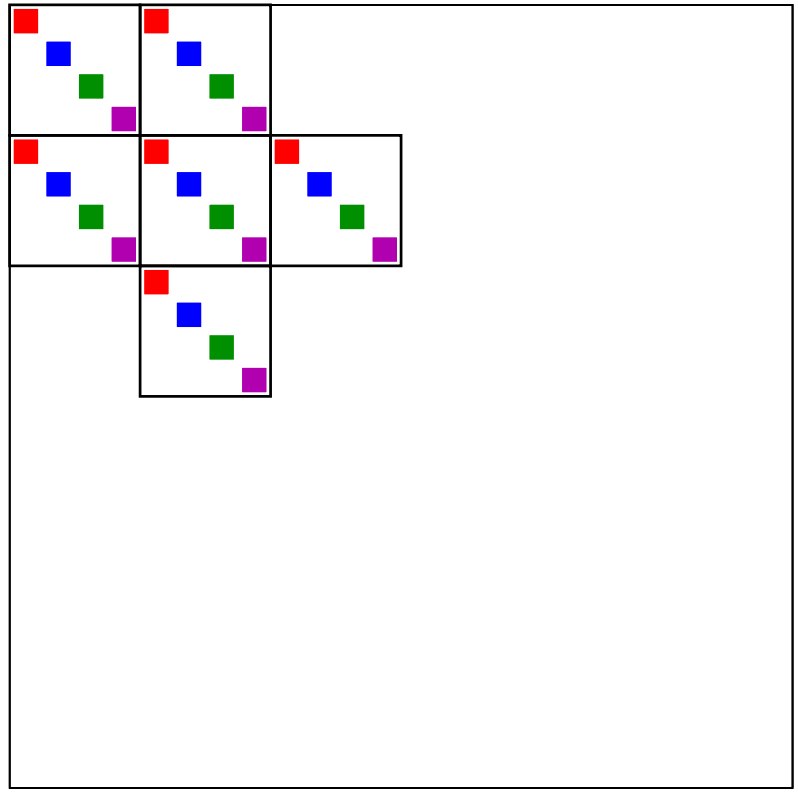




On groupe les blocs diagonaux 4 par 4.

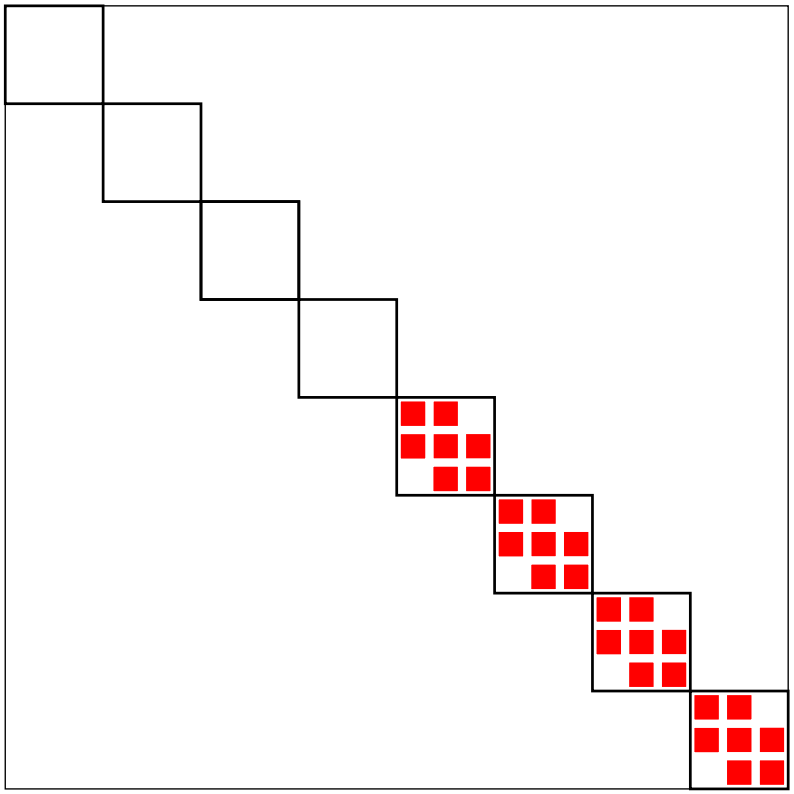
Matrice Diagonal<TriDiagonal>

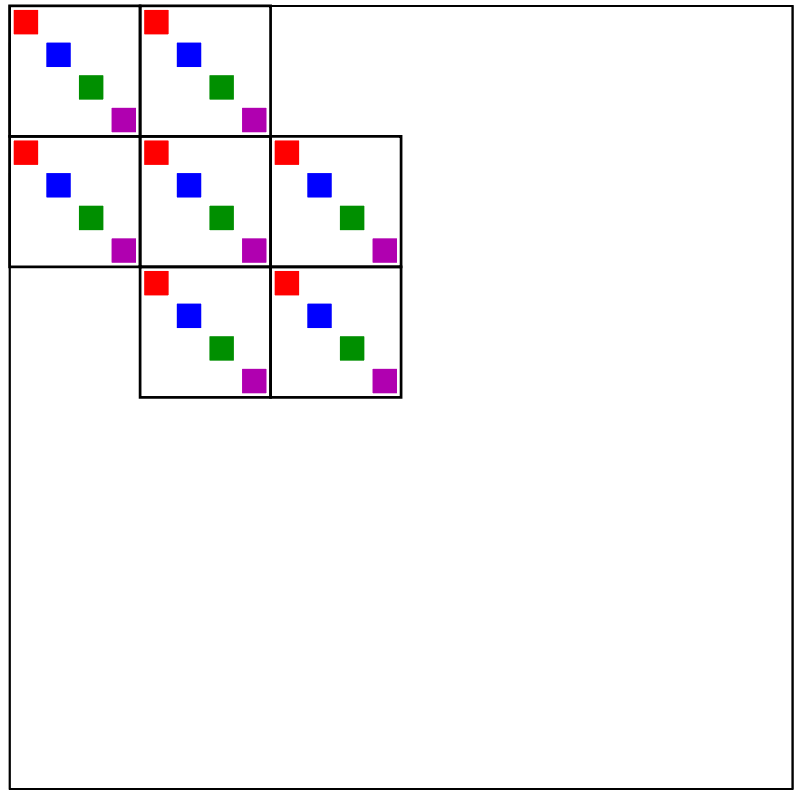




On groupe les blocs diagonaux 4 par 4.

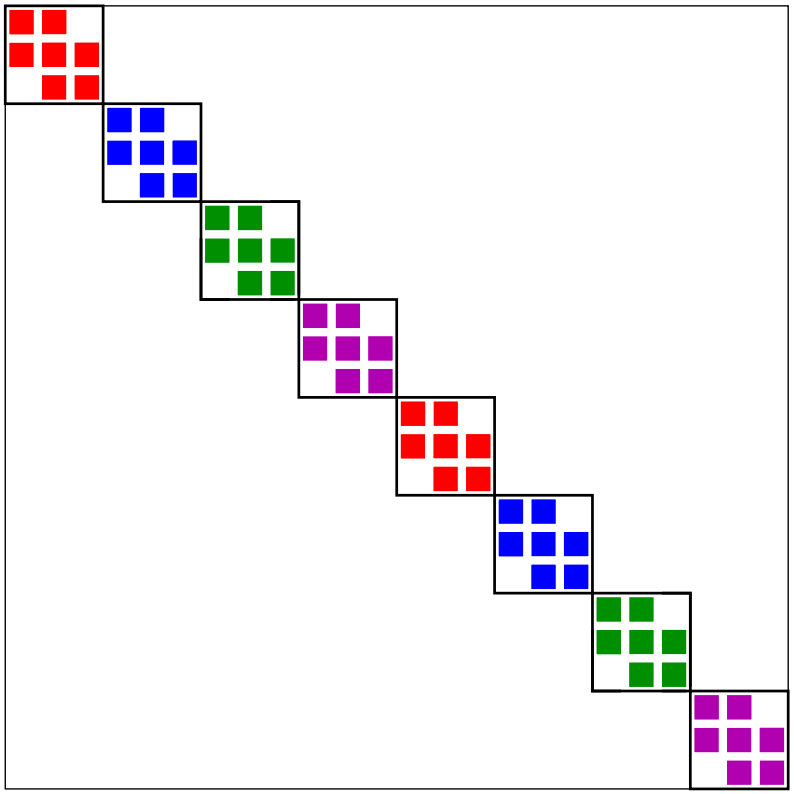
Matrice Diagonal<TriDiagonal>

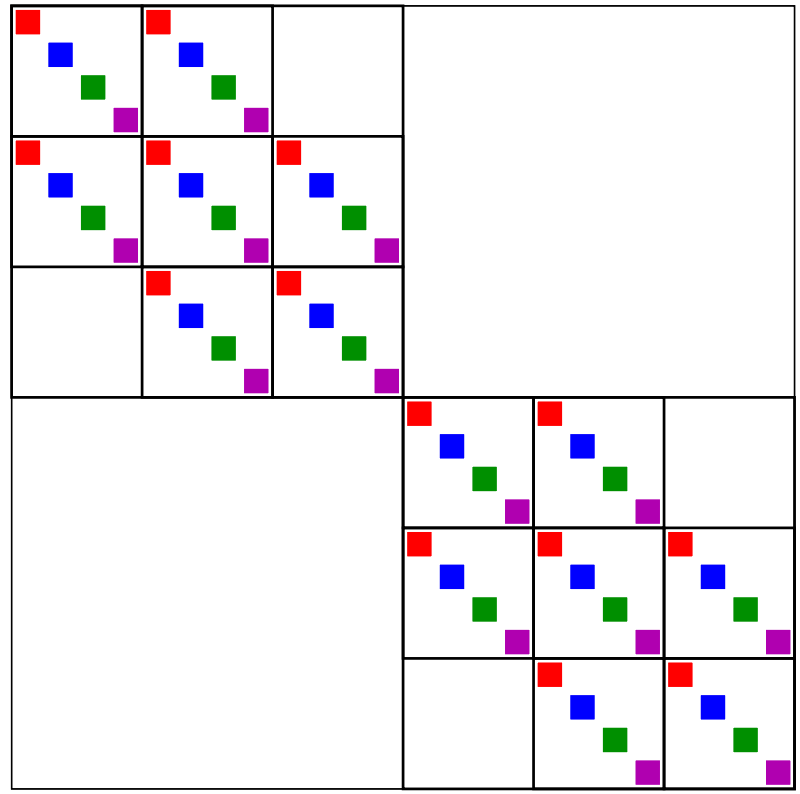




On groupe les blocs diagonaux 4 par 4.

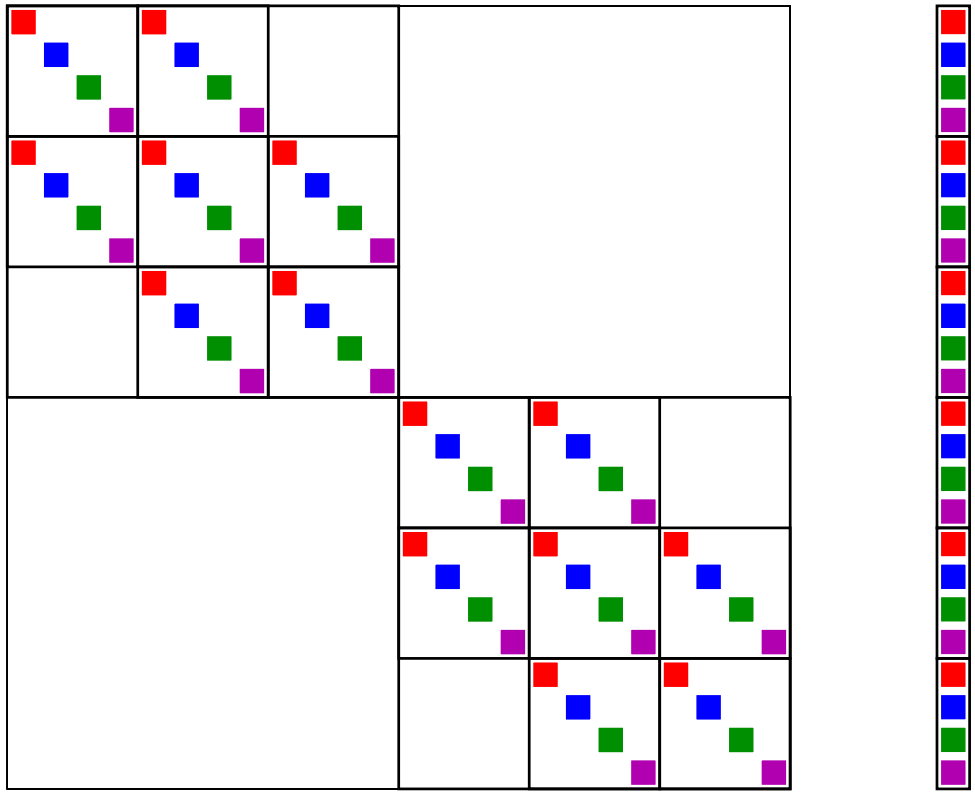
Matrice Diagonal<TriDiagonal>





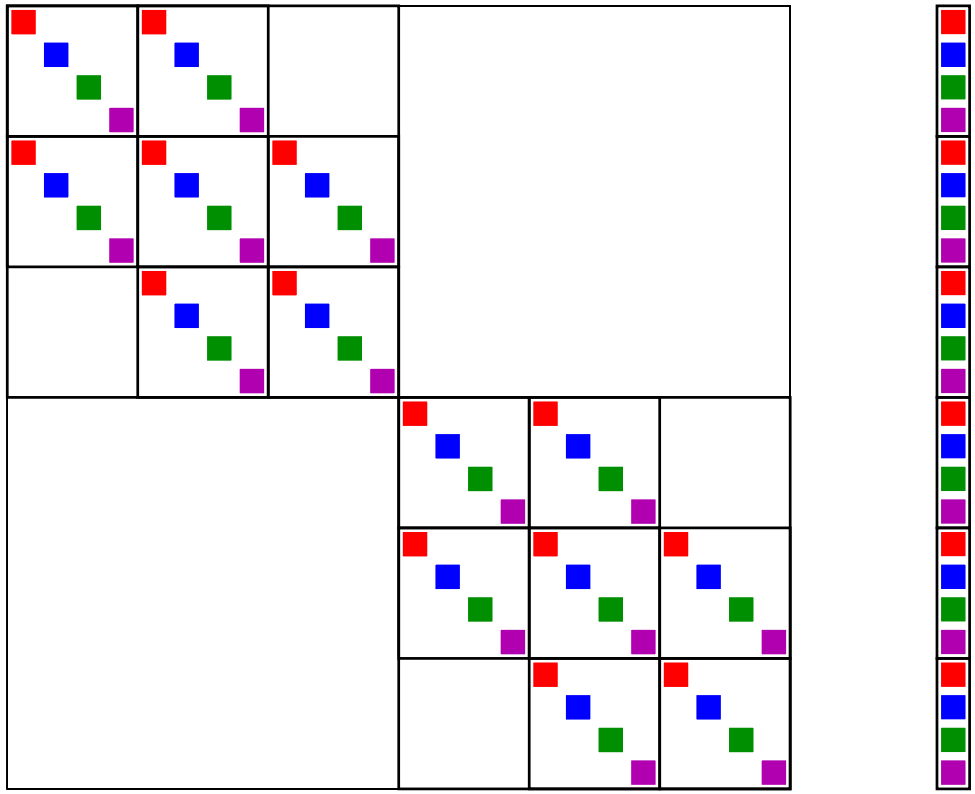
On groupe les blocs diagonaux 4 par 4.

Matrice Diagonal<TriDiagonal>



Diag<TridaDiag>→Diag< Tridiag< Diag > >

Matrice Diagonal<TriDiagonal>



`X[i][j]` $\mathrm{i} \in [0, \mathrm{N_x}[, \mathrm{j} \in [0, \mathrm{N_y}[$

$\rightarrow$ `X[i][j][k]` $\mathrm{i} \in [0, \mathrm{N_x}/4[, \mathrm{j} \in [0, \mathrm{N_y}[, \mathrm{k} \in [0, 4[$

Système Diagonal<TriDiagonal>

```
for (int i=0 ; i<Nx ; i++){//boucle parallelisable
  typedef Legolas::MultiVector<RT,1> V1D;
  const V1D & D(D2D[i]),U(U2D[i]),L(L2D[i]),B(B2D[i]);
  V1D & X(X2D_[i]),S(S_);

  RT s(D[0]);
  RT sm1=1.f/s;

  X[0]=B[0]*sm1;
  //Descente
  for (int j=1 ; j < Ny ; j++ ){
    S[j]=U[j-1]*sm1;
    s=D[j]-L[j]*S[j];
    X[j]=B[j]-L[j]*X[j-1];
    sm1=1.f/s;
    X[j]*=sm1;
  }
  //Remontee
  for (int j=(Ny-2) ; j >= 0 ; j-- ){
    X[j]-=S[j+1]*X[j+1];
  }
}
```

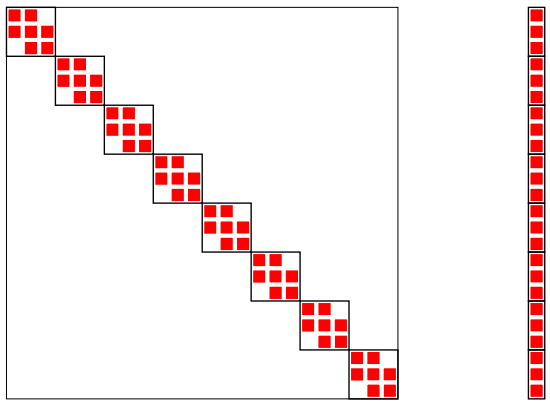
Système Diagonal<TriDiagonal<V4f> >

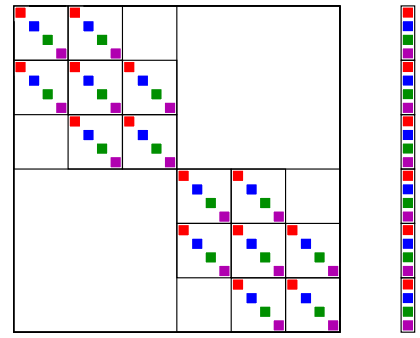
```
for (int b=0 ; b<Nx/4 ; b++){//boucle parallelisable
  //Interface vers des vecteurs qui renvoient des V4f
  ConstEigenWrap<RT> D(DI[b]),U(UI[b]),L(LI[b]),B(BI[b]);
  EigenWrap<RT> X(XI_[b]),S(S_);

  V4f s(D[0]);
  V4f sm1=s.inverse();

  X[0]=B[0]*sm1;
  //Descente
  for (int j=1 ; j < Ny ; j++ ){
    S[j]=U[j-1]*sm1;
    s=D[j]-L[j]*S[j];
    X[j]=B[j]-L[j]*X[j-1];
    sm1=s.inverse();
    X[j]*=sm1;
  }
  //Remontee
  for (int j=(Ny-2) ; j >= 0 ; j-- ){
    X[j]-=S[j+1]*X[j+1];
  }
}
```

Eigen Wrapper

```
template<>
struct EigenWrap<float>{
  static const int ps=4;

  typedef Legolas::MultiVector<float,1> V1D;

  typedef Eigen::Array<float, ps, 1> V4f;
  typedef Eigen::Map<V4f,Eigen::Aligned>  V4fView;
  typedef Eigen::Map<const V4f,Eigen::Aligned>  ↩
      ConstV4fView;

  V1D & v1D_;
  EigenWrap( V1D & v1D):v1D_(v1D){}

  inline V4fView operator[] (int i){ return &v1D_[i*ps] ;}
  inline ConstV4fView operator[] (int i) const { return ↩
      &v1D_[i*ps] ;}
};
```

Système Diagonal<TriDiagonal<V4f> >



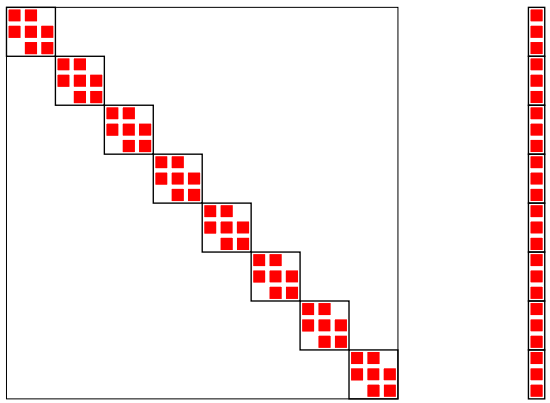


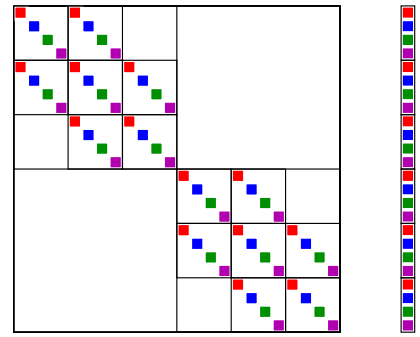
Intel Xeon E6520 2.4 GHz, 2×4 cores

- Nombre de blocs tridiagonaux : $N_x = 600$
- Taille des blocs tridiagonaux : $N_y = 800$

Implémentation	Durée du calcul (s)	GFLOPS	Speed-Up
Séquentielle	0.88	0.60	1.0

Système Diagonal<TriDiagonal<V4f> >



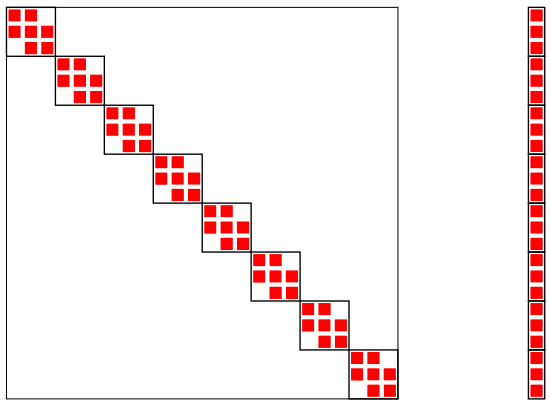


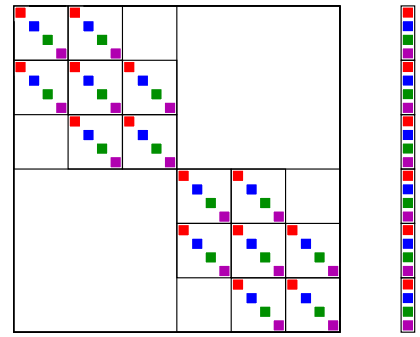
Intel Xeon E6520 2.4 GHz, 2×4 cores

- Nombre de blocs tridiagonaux : $N_x = 600$
- Taille des blocs tridiagonaux : $N_y = 800$

Implémentation	Durée du calcul (s)	GFLOPS	Speed-Up
Séquentielle	0.88	0.60	1.0
Parallèle (TBB)	0.13	4.2	×6.97

Système Diagonal<TriDiagonal<V4f> >



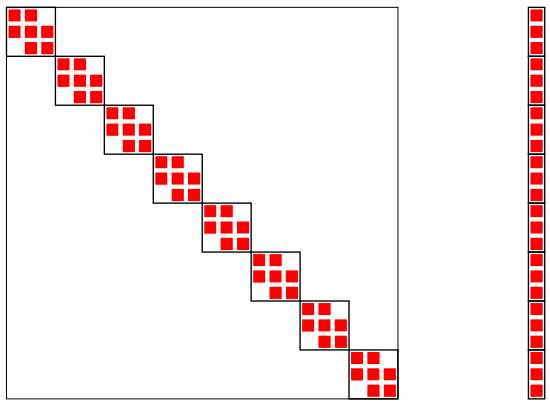


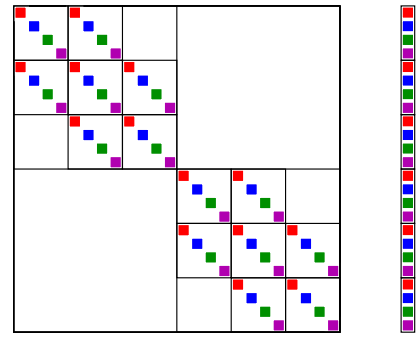
Intel Xeon E6520 2.4 GHz, 2×4 cores

- Nombre de blocs tridiagonaux : $N_x = 600$
- Taille des blocs tridiagonaux : $N_y = 800$

Implémentation	Durée du calcul (s)	GFLOPS	Speed-Up
Séquentielle	0.88	0.60	1.0
Parallèle (TBB)	0.13	4.2	×6.97
SIMD (Eigen)	0.20	2.6	×4.31

Système Diagonal<TriDiagonal<V4f> >



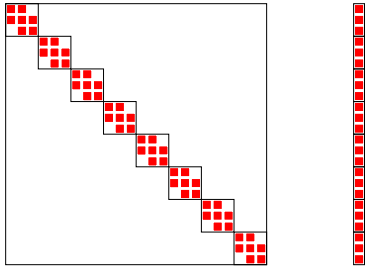


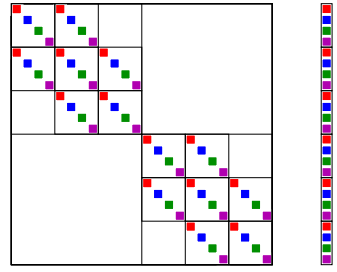
Intel Xeon E6520 2.4 GHz, 2×4 cores

- Nombre de blocs tridiagonaux : $N_x = 600$
- Taille des blocs tridiagonaux : $N_y = 800$

Implémentation	Durée du calcul (s)	GFLOPS	Speed-Up
Séquentielle	0.88	0.60	1.0
Parallèle (TBB)	0.13	4.2	×6.97
SIMD (Eigen)	0.20	2.6	×4.31
TBB+Eigen	0.04	13.3	×22.1

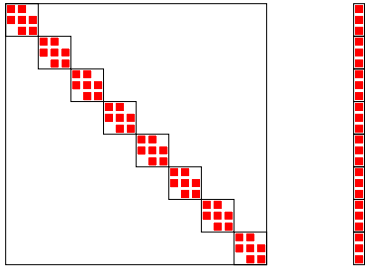
Système Diagonal<TriDiagonal<V4f> > (I6700K)

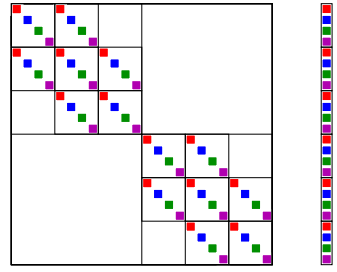




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB				
SSE4				
AVX2				
TBB+SSE4				
TBB+AVX2				
OpenCL (GTX670)				

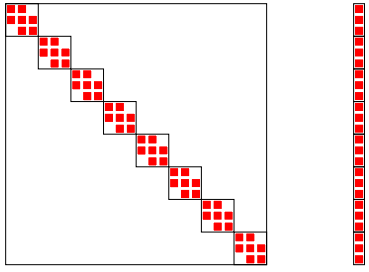
Système Diagonal<TriDiagonal<V4f> > (I6700K)

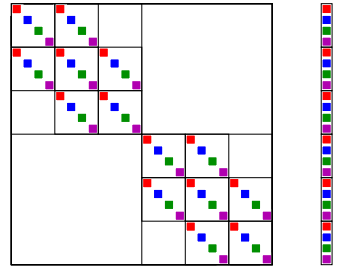




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4				
AVX2				
TBB+SSE4				
TBB+AVX2				
OpenCL (GTX670)				

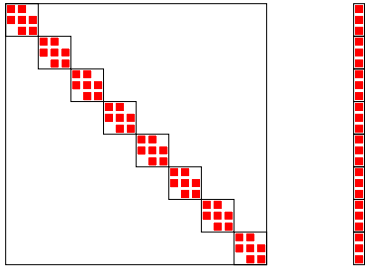
Système Diagonal<TriDiagonal<V4f> > (I6700K)

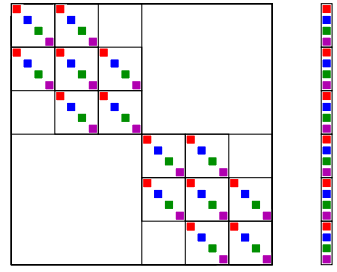




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4	7.5	7.4	7.4	7.5
AVX2				
TBB+SSE4				
TBB+AVX2				
OpenCL (GTX670)				

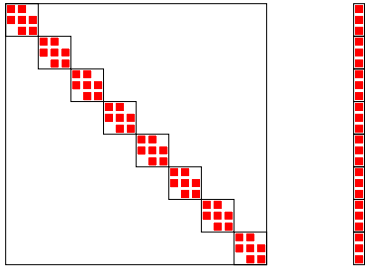
Système Diagonal<TriDiagonal<V4f> > (I6700K)

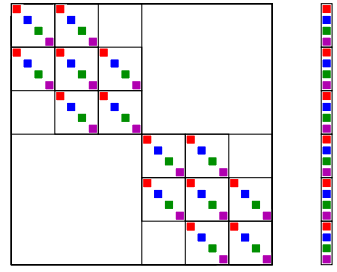




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4	7.5	7.4	7.4	7.5
AVX2	8.8	8.9	9.3	11
TBB+SSE4				
TBB+AVX2				
OpenCL (GTX670)				

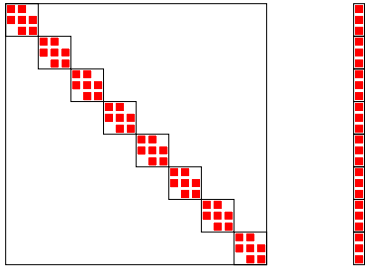
Système Diagonal<TriDiagonal<V4f> > (I6700K)

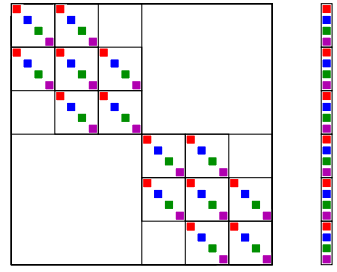




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4	7.5	7.4	7.4	7.5
AVX2	8.8	8.9	9.3	11
TBB+SSE4	11.5	11.4	11.4	43
TBB+AVX2				
OpenCL (GTX670)				

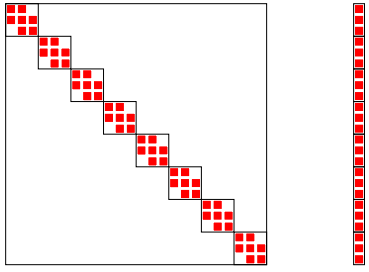
Système Diagonal<TriDiagonal<V4f> > (I6700K)

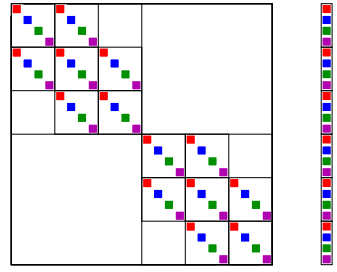




GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4	7.5	7.4	7.4	7.5
AVX2	8.8	8.9	9.3	11
TBB+SSE4	11.5	11.4	11.4	43
TBB+AVX2	11.5	11.4	11.7	44
OpenCL (GTX670)				

Système Diagonal<TriDiagonal<V4f> > (I6700K)





GFLOPS	512^3	256^3	128^3	64^3
Séquentielle	1.8	1.7	1.7	1.7
TBB	7.7	7.4	6.8	11
SSE4	7.5	7.4	7.4	7.5
AVX2	8.8	8.9	9.3	11
TBB+SSE4	11.5	11.4	11.4	43
TBB+AVX2	11.5	11.4	11.7	44
OpenCL (GTX670)	39.4	35.4	35.5	27

C++14 Legolas++

```
struct TriDiagonalSolver{
  template <class ARRAY_2D>
  void operator()(int begin, int end,
      ARRAY_2D D2D, ARRAY_2D U2D, ARRAY_2D L2D,
                  ARRAY_2D B2D, ARRAY_2D X2D) const {
    typename ARRAY_2D::Element S(X2D[0].shape());
    typename ARRAY_2D::ScalarType one(1.0),s,sm1;

    for (int i=begin ; i<end ; i++){
      auto D=D2D[i]; auto U=U2D[i]; auto L=L2D[i];
      auto B=B2D[i]; auto X=X2D[i];
      s=D[0];
      sm1=one/s;
      const int size=X.size();
      X[0]=B[0]*sm1;
      for(int j=1; j<size; j++){//Descente
        S[j]=U[j-1]*sm1;
        s=D[j]-L[j]*S[j];
        X[j]=B[j]-L[j]*X[j-1];
        sm1=one/s;
        X[j]*=sm1;
      }
      for (int j=(size-2);j>=0 ; j--){//Remontee
        X[j]-=S[j+1]*X[j+1];
      }
    }
  }
};
```

C++14 Legolas++

```
int main () {
  size_t nx=800;
  size_t ny=200;
  typedef Legolas::Array<float,2> V2D;

  V2D u2D(nx,ny);  V2D l2D(nx,ny);  V2D d2D(nx,ny);
  V2D X(nx,ny);  V2D Y(nx,ny);

  Legolas::map(TriDiagonalSolver(),d2D,u2D,l2D,Y,X);
}
```

```
CXX = g++  -O3 -std=c++14  -ltbb -funroll-loops -msse4.1
```

```
1.18 GFlops
```

C++14 Legolas++

```
int main () {
  size_t nx=800;
  size_t ny=200;
  typedef Legolas::Array<float,2,4,2> V2D;

  V2D u2D(nx,ny);  V2D l2D(nx,ny);  V2D d2D(nx,ny);
  V2D X(nx,ny);  V2D Y(nx,ny);

  Legolas::map(TriDiagonalSolver(),d2D,u2D,l2D,Y,X);
}
```

```
CXX = g++  -O3 -std=c++14  -ltbb -funroll-loops -msse4.1
```

```
4.56 GFlops
```

C++14 Legolas++

```
int main () {
  size_t nx=800;
  size_t ny=200;
  typedef Legolas::Array<float,2,4,2> V2D;

  V2D u2D(nx,ny);  V2D l2D(nx,ny);  V2D d2D(nx,ny);
  V2D X(nx,ny);  V2D Y(nx,ny);

  Legolas::parmap(TriDiagonalSolver(),d2D,u2D,l2D,Y,X);
}

CXX = g++  -O3 -std=c++14  -ltbb -funroll-loops -msse4.1

17 GFlops
```

C++14 Legolas++

```
int main () {
  size_t nx=800;
  size_t ny=200;
  typedef Legolas::Array<float,2,8,2> V2D;

  V2D u2D(nx,ny);  V2D l2D(nx,ny);  V2D d2D(nx,ny);
  V2D X(nx,ny);  V2D Y(nx,ny);

  Legolas::parmap(TriDiagonalSolver(),d2D,u2D,l2D,Y,X);
}

CXX = g++  -O3 -std=c++14  -ltbb -funroll-loops -msse4.1

17 GFlops
```

C++14 Array map

```
template <class ALGO, typename... ARRAYS>
void scalar_map(ALGO algo, ARRAYS... rest){
  const int thisSize=getArraysSize(rest...);
  algo(0,thisSize,rest...);
}
```

C++14 Array getPackedView

```
typedef Eigen::Array<SCALAR_TYPE,PACK_SIZE,1> ↩
    PackedScalarType;
typedef Legolas::Array<PackedScalarType,LEVEL,1,1> ↩
    PackedArrayView;

const PackedArrayView getPackedView( void ) const {
 ArrayShape<1,1,LEVEL> ↩
     packedShape=this->shape_.getPackedShape();
 PackedScalarType * ↩
     vectorPtr=reinterpret_cast<PackedScalarType*>(this->dataPtr_);
 return PackedArrayView(packedShape,vectorPtr);
}
```

C++14 Array vecMap

```
template <class ALGO, typename... ARRAYS>
void vecMap(ALGO algo, ARRAYS... rest){
 const int thisSize=getArraysSize(rest...);
 const int thisPackedSize=getPackedArraysSize(rest...);
 const int packSize=getPackSize(rest...);

 algo(0,thisPackedSize,rest.getPackedView()...);
 // If rest.. is equal to a1,a2,..,an then
 // rest.getPackedView()... is equivalent to
 //  ↩
    a1.getPackedView(),a2.getPackedView(),...,an.getPackedView()
 algo(thisPackedSize*packSize,thisSize,rest...);
}
```

C++14 Array parMap

```
template <class ALGO, typename... ARRAYS>
void parallel_ranged_map(int begin, int end, ALGO algo, ↩
    ARRAYS... rest){
  tbb::parallel_for(tbb::blocked_range<int>(begin,end),
        [=](tbb::blocked_range<int> r){
       algo(r.begin(),r.end(),rest...);}
            ,tbb::auto_partitioner());
}
```

- Le C++14 facilite la programmation générique.

C++14

- Le C++14 facilite la programmation générique.
- Legolas++ utilise une combinaison entre les tbb et Eigen.

C++14

- Le C++14 facilite la programmation générique.
- Legolas++ utilise une combinaison entre les tbb et Eigen.
- Ecole d'été CEA-EFD-INRIA (15-19 juin 2015)

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  $\rightarrow$task based programming.

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  →task based programming.
- Vectorisation :

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  $\rightarrow$task based programming.
- Vectorisation :
  - Réorganisation des données.

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  →task based programming.
- Vectorisation :
  - Réorganisation des données.
  - Vectorisation portable avec de bons outils
    →bibliothèques génériques.

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  →task based programming.
- Vectorisation :
  - Réorganisation des données.
  - Vectorisation portable avec de bons outils
    →bibliothèques génériques.
  - Un facteur 10 possible.

Conclusion : la performance machine augmente mais...

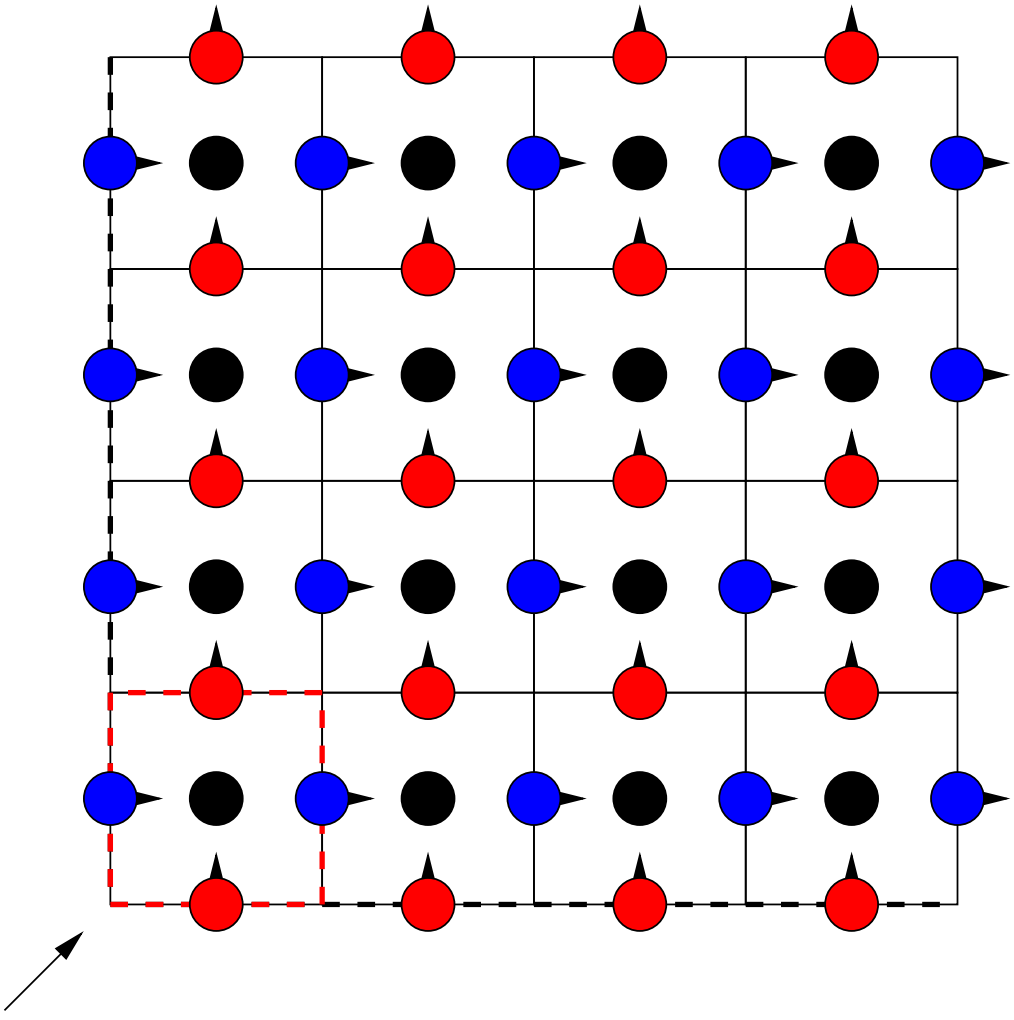
- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  →task based programming.
- Vectorisation :
  - Réorganisation des données.
  - Vectorisation portable avec de bons outils
    →bibliothèques génériques.
  - Un facteur 10 possible.
  - Convergence avec les GPU (OpenCL).

Conclusion : la performance machine augmente mais...

- Memory Wall (Précision mixte, recalculs, maths..).
- Programmation hybride (MPI+MT)
  →task based programming.
- Vectorisation :
  - Réorganisation des données.
  - Vectorisation portable avec de bons outils
    →bibliothèques génériques.
  - Un facteur 10 possible.
  - Convergence avec les GPU (OpenCL).
- Le HPC est sorti des grands centres de calculs...

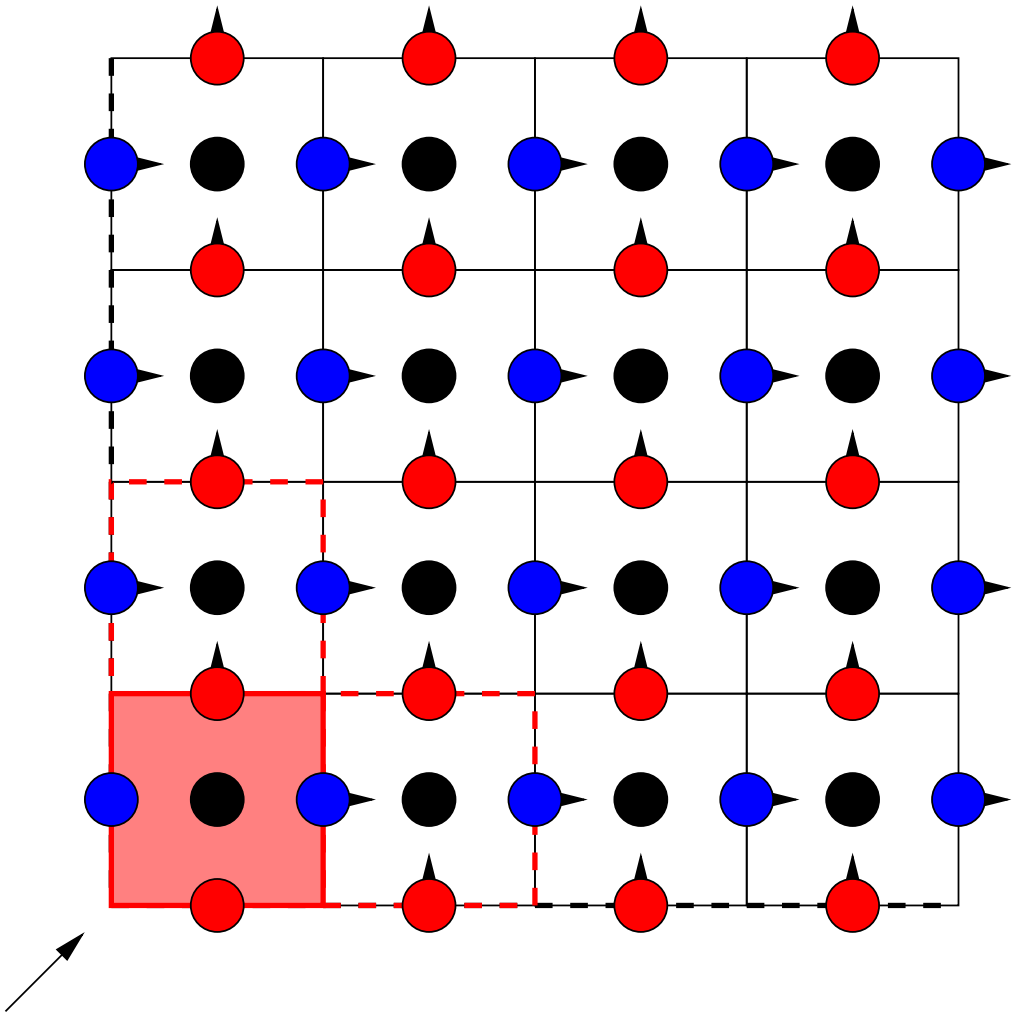
3. Task, Dags, Runtime

Balayage 2D : Sweep2D



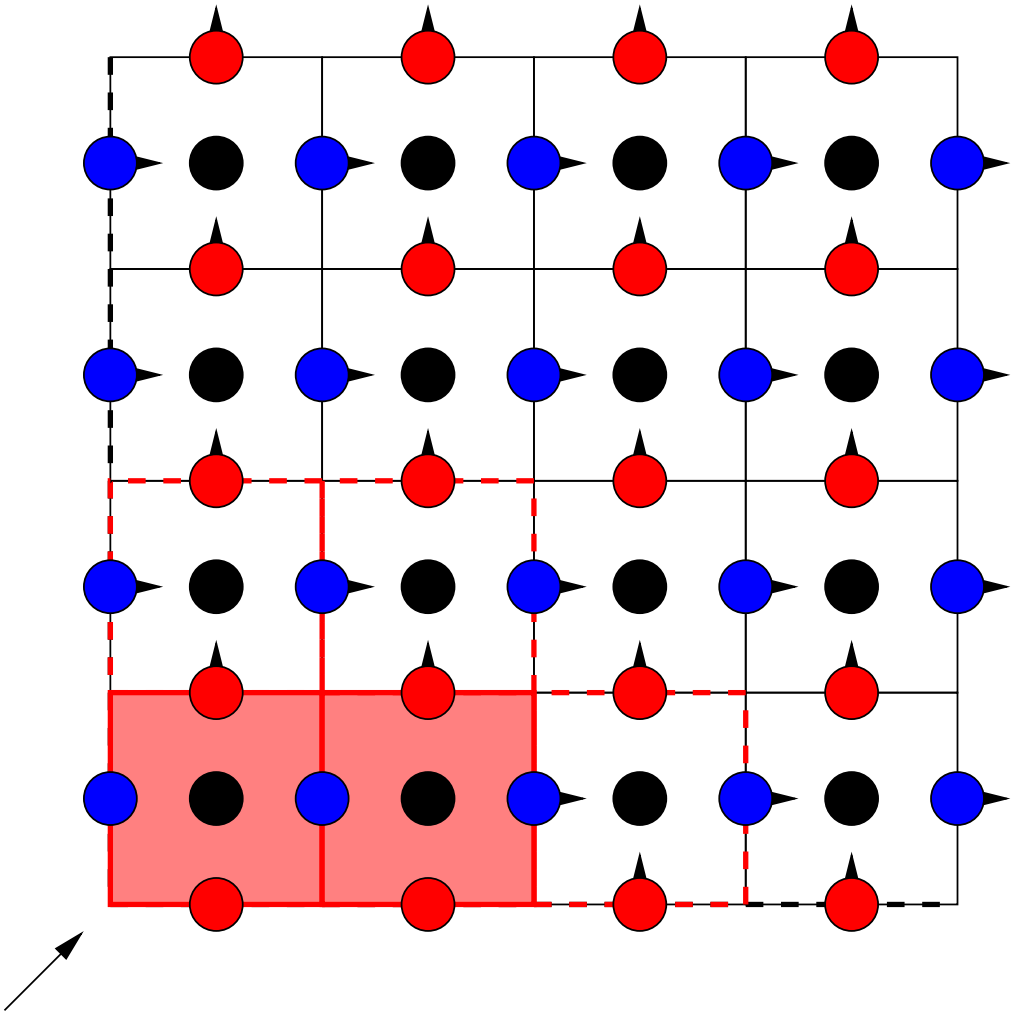
Noyau de calcul pour la résolution de l'équation de Boltzmann qui régit le transport des particules neutres (neutron, photons,..). Ce noyau est utilisé pour le calcul du flux de neutron dans les cœurs de centrales.

Balayage 2D : Sweep2D



Noyau de calcul pour la résolution de l'équation de Boltzmann qui régit le transport des particules neutres (neutron, photons,..). Ce noyau est utilisé pour le calcul du flux de neutron dans les cœurs de centrales.

Balayage 2D : Sweep2D



Noyau de calcul pour la résolution de l'équation de Boltzmann qui régit le transport des particules neutres (neutron, photons,..). Ce noyau est utilisé pour le calcul du flux de neutron dans les cœurs de centrales.

- Implémentation // ?

Implémentation C++ : Grid2D

```
template <class T>
struct Grid2D{
  int nx_;
  int ny_;
  std::vector<T> data_;

  Grid2D(int nx, int ↩
      ny):nx_(nx),ny_(ny),data_(nx*ny){}

  const T & operator()(int row, int col) const{
    return data_[row+nx_*col];
  }

  T & operator()(int row, int col){
    return data_[row+nx_*col];
  }
};
```

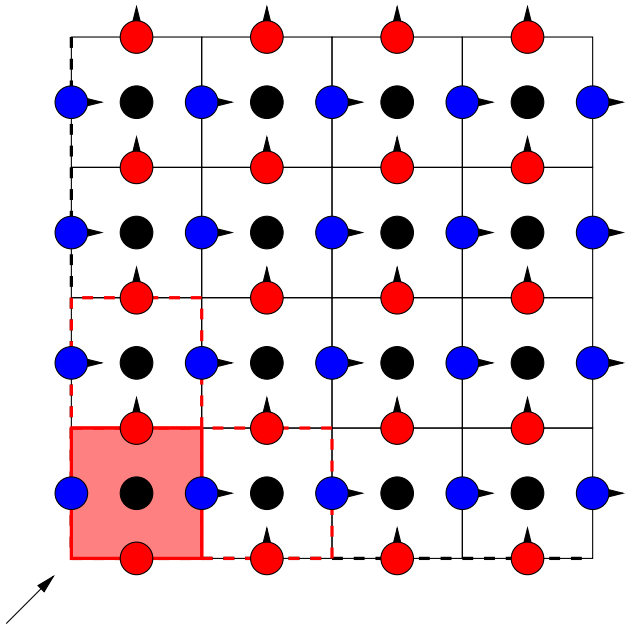
Implémentation C++ : Mesh2D

```
#include "Cell.hxx"
#include "Grid2D.hxx"

class Mesh2D : public Grid2D<Cell>{
public:
  Mesh2D(int nx, int ny):Grid2D<Cell>(nx,ny){

    Grid2D<Cell> & cells=*this;

    for (int i=0 ; i<nx ; i++){
      for (int j=0 ; j<ny ; j++){

        cells(i,j).value_=1.0;

      }
    }
  }
};
```

Implémentation C++ : Mesh2DSweepFunctor

```
class Mesh2DSweepFunctor {
 Mesh2D & mesh2D_;
 int f_;//frontIndex
 void
 operator()(const tbb::blocked_range<int> & r) ↩
     const {
  for (int i=r.begin() ; i!=r.end() ; i++){
    const int j = f_-i;
    mesh2D_(i,j).value_=2.0;
    fonctionLongue();
  }
 }
 ...
};
```

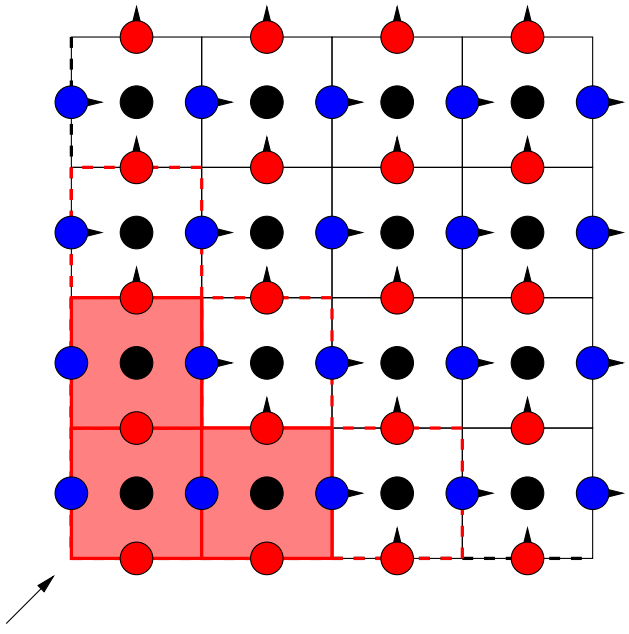
Sweep2D : Séquence de tbb::parallel_for



```
typedef ↩
    tbb::blocked_range<int> BR;

int nFront=nx+ny-1;

for (int f=0; f<nFront ;f++){
 int xMin=std::max(0,f-ny+1);
 int xMax=std::min(f,nx-1);

 Mesh2DSweepFunctor ↩
     sf(&mesh2D,f);

 parallel_for(BR(xMin,xMax-1),sf);
}
```

f=0

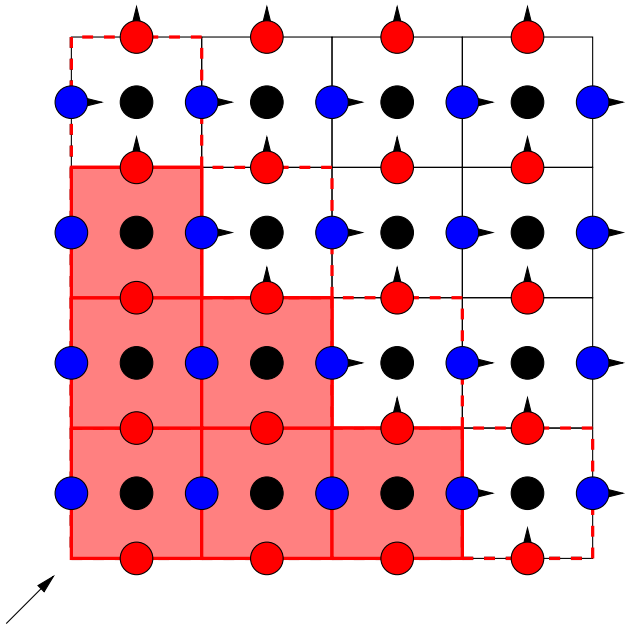
Sweep2D : Séquence de tbb::parallel_for



```
typedef ↩
    tbb::blocked_range<int> BR;

int nFront=nx+ny-1;

for (int f=0; f<nFront ;f++){
 int xMin=std::max(0,f-ny+1);
 int xMax=std::min(f,nx-1);

 Mesh2DSweepFunctor ↩
     sf(&mesh2D,f);

 parallel_for(BR(xMin,xMax-1),sf);
}
```

f=1

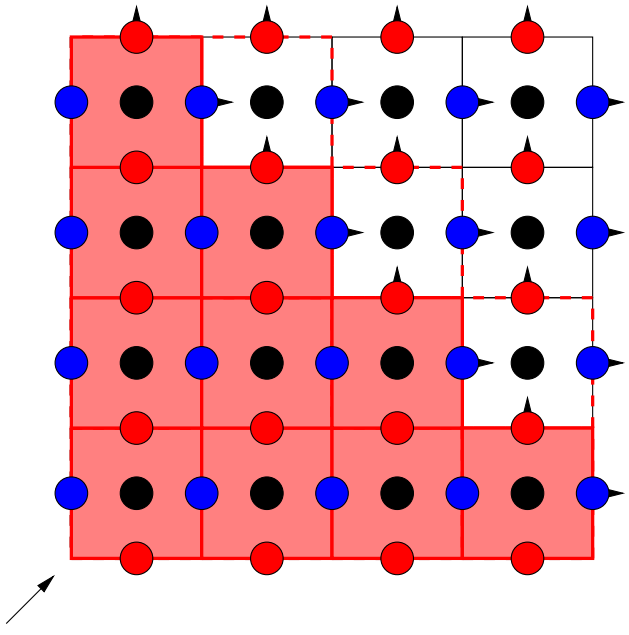
Sweep2D : Séquence de tbb::parallel_for



f=2

```
typedef ↩
    tbb::blocked_range<int> BR;

int nFront=nx+ny-1;

for (int f=0; f<nFront ;f++){
 int xMin=std::max(0,f-ny+1);
 int xMax=std::min(f,nx-1);

 Mesh2DSweepFunctor ↩
     sf(&mesh2D,f);

 parallel_for(BR(xMin,xMax-1),sf);
}
```

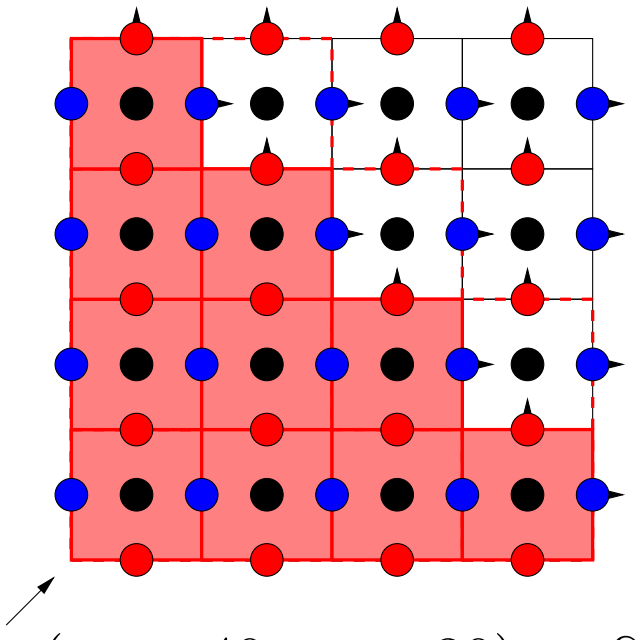
Sweep2D : Séquence de tbb::parallel_for



```
typedef ↩
    tbb::blocked_range<int> BR;

int nFront=nx+ny-1;

for (int f=0; f<nFront ;f++){
 int xMin=std::max(0,f-ny);
 int xMax=std::min(f,nx-1);

 Mesh2DSweepFunctor ↩
     sf(&mesh2D,f);

 parallel_for(BR(xMin,xMax-1),sf);
}
```

f=3

Sweep2D : Speed-Up théorique



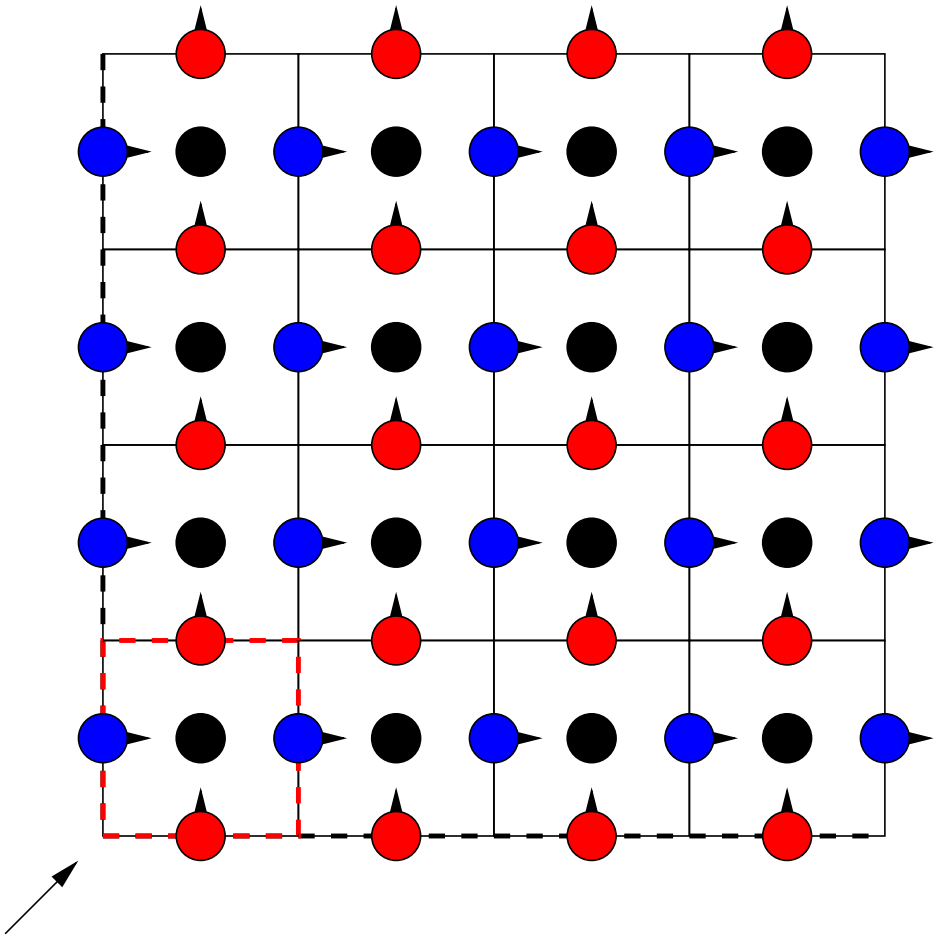
- Nb d'étape pour faire les coins : $2n_p$
- Nb de cellules parallèles : $n_x n_y - n_p^2$
- Nb d'étapes parallèles : $2n_p + (n_x n_y - n_p^2)/n_p$
- Nb d'étapes séquentielles : $n_x n_y$
- Speed-Up théorique : $n_p(\frac{n_x n_y}{n_x n_y + n_p^2})$

$(n_x = 40, n_y = 30) \rightarrow SpU_2 = 1.993, SpU_4 = 3.95, SpU_8 = 7.59$

Performances $n_x = 40, n_y = 30$ Z600 (8 cœurs)

N_p	Temps(s)	$(< t^2 > - < t >^2)^{1/2}$	SpeedUp	$n_p(\frac{n_x n_y}{n_x n_y + n_p^2})$
1	62.3596	0.004	1	1
2	35.6645	0.022	1.748	1.993
4	19.3053	0.034	3.229	3.595
8	9.98104	0.056	6.247	7.594
16	10.1714	0.117	6.13	13.18

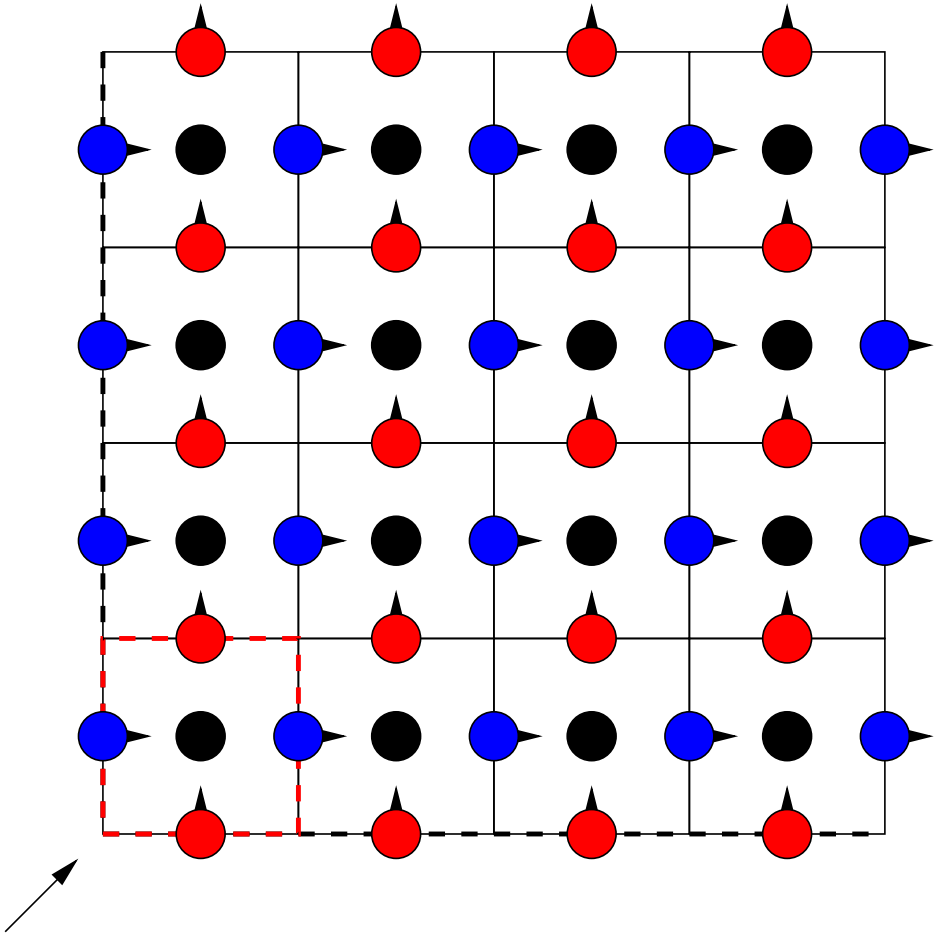
Balayage 2D : Expression des dépendances de tâche



L'algorithme précédent était surcontraint. Cette fois on ne veut exprimer que des dépendances intrinsèques entre les tâches (cellules).

En 2D chaque cellule a au maximum 2 prédécesseurs. l'attribut `count_` rajouté à la classe `Cell` dénombre ces dépendance.

Balayage 2D : struct Cell



```
struct Cell{
  double value_;
  tbb::atomic<int> count_;

  Cell( void ):value_(0.0),
               count_(){
  }
};
```

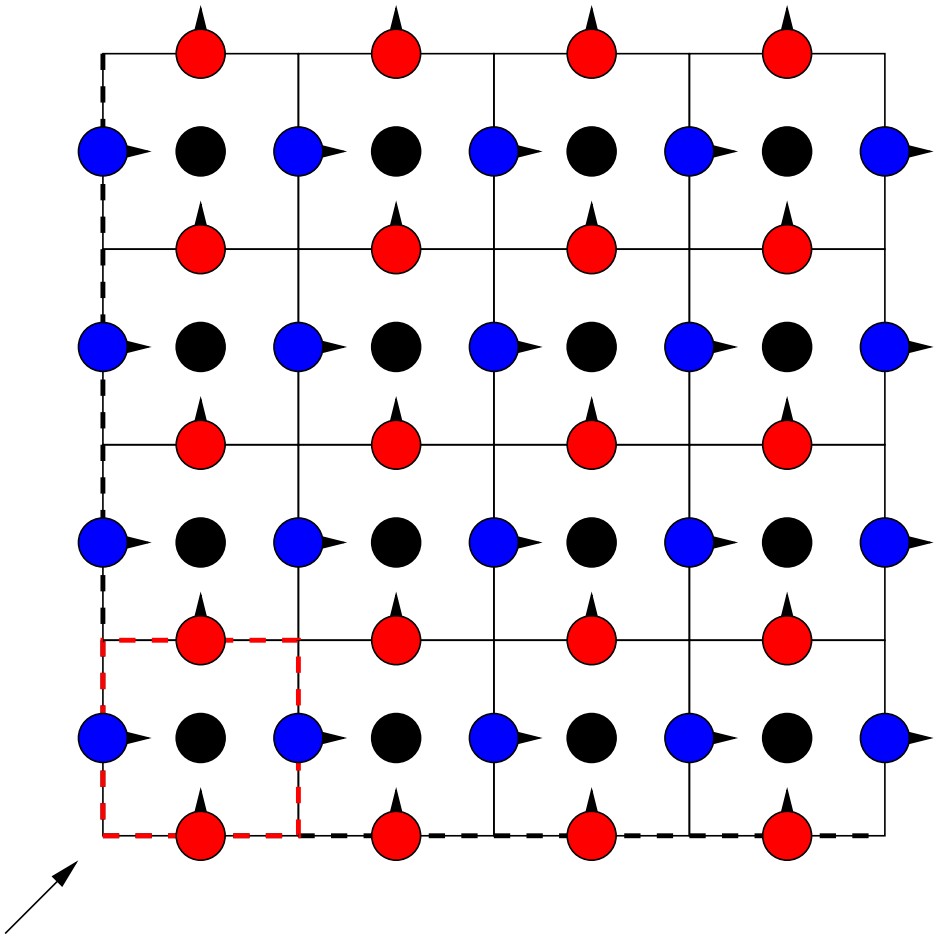
Implémentation C++ : Mesh2D

```
class Mesh2D : public Grid2D<Cell>{
public:
  Mesh2D(int nx, int ny):Grid2D<Cell>(nx,ny){

    Grid2D<Cell> & cells=*this;

    for (int i=0 ; i<nx ; i++){
      for (int j=0 ; j<ny ; j++){

        cells(i,j).value_=1.0;

      }
    }
  }
};
```

Implémentation C++ : Mesh2D

```
class Mesh2D : public Grid2D<Cell>{
public:
  Mesh2D(int nx, int ny):Grid2D<Cell>(nx,ny){

    Grid2D<Cell> & cells=*this;

    for (int i=0 ; i<nx ; i++){
      for (int j=0 ; j<ny ; j++){

        cells(i,j).value_=1.0;
        cells(i,j).count_=((i>0)+(j>0));
      }
    }
  }
};
```

Coordonnées de tâche : `struct CIJ`



```
struct CIJ{
  int i_;
  int j_;

  CIJ(int i, int j):i_(i),
                    j_(j){}
};
```

tbb::parallel_do

```
//Task List
std::vector<CIJ> todo;
todo.push_back( CIJ(0,0) );

//Functor
Mesh2DSweepFunctor sweep(mesh2D);

//Perform the task list in parallel
tbb::parallel_do(todo.begin(),todo.end(),sweep);
}
```

```
class Mesh2DSweepFunctor {
 Mesh2D & mesh2D_;
 typedef tbb::parallel_do_feeder<CIJ> Feeder;
 //Helper function that update the task list
 void updateNewCell(int i, int j, Feeder & ↩
     feeder) const{
  Cell & rightCell=mesh2D_(i,j);
  if (0 ==  --rightCell.count_) ↩
      feeder.add(CIJ(i,j));
 }
 typedef CIJ argument_type;
 //main functor function
 void operator()(CIJ cij, Feeder & feeder ) ↩
     const {
  const int i=cij.i_; const int j=cij.j_;
  fonctionLongue();

  if ((i+1)< mesh2D_.nx_) ↩
      updateNewCell(i+1,j,feeder);
  if ((j+1)< mesh2D_.ny_) ↩
      updateNewCell(i,j+1,feeder);
 }
```

Performances $n_x = 40, n_y = 30$ Z600 (8 cœurs)

`parallel_do`

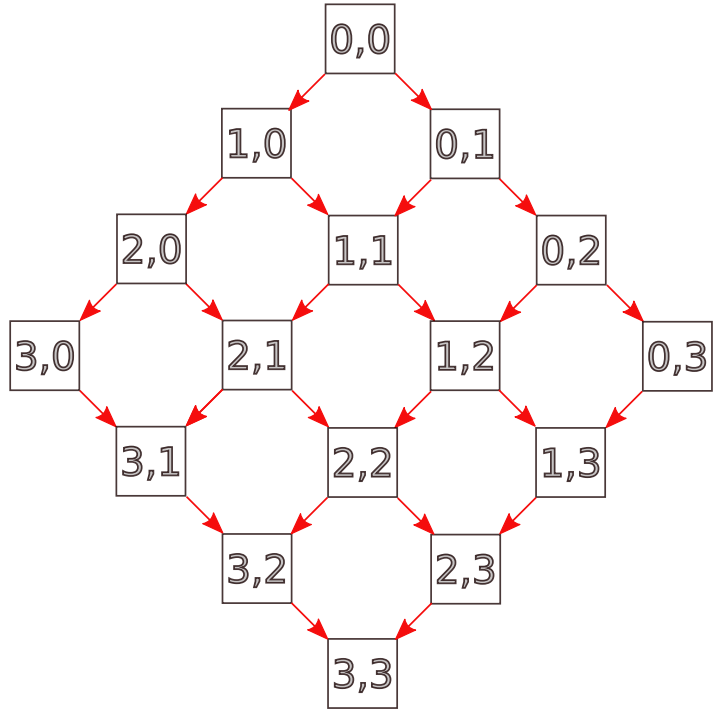
N_p	$\min(\{t_i\})$	SpeedUp	$n_p(\frac{n_x n_y}{n_x n_y + n_p^2})$
2	34.06	1.783	1.993
4	17.56	3.551	3.942
8	9.43	6.613	7.594
16	9.60	6.594	13.18

`parallel_for (rappel)`

N_p	$\min(t)$	SpeedUp	$n_p(\frac{n_x n_y}{n_x n_y + n_p^2})$
1	62.35	1	1
2	32.28	1.93	1.993
4	17.23	3.61	33.94
8	9.953	6.26	7.594
16	10.19	6.11	13.18

Directed Acyclic Graphs

3,0	3,1	3,2	3,3
2,0	2,1	2,2	2,3
1,0	1,1	1,2	1,3
0,0	0,1	0,2	0,3

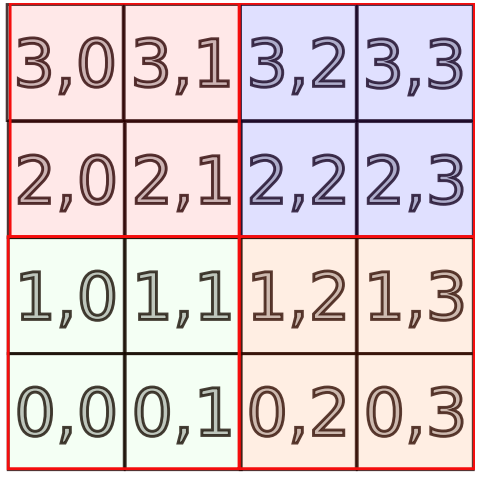


Runtime (+Y)

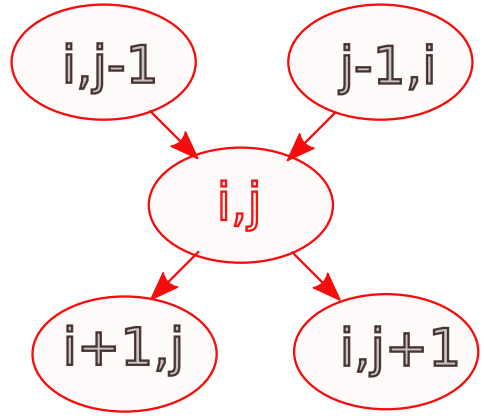
Cluster	100,1000 nodes	Distr. Mem.	RunTime (ParSec,...)
Node	1,2,4,8 cpus	Shared Mem.	
CPU	2,16 cores	LLC	
Core	simd units	L1/L2	
SIMD	128,512 bits ops	Registers	asm,intrinsics...

Runtime : ParSec

task
+



+



→ Génération automatique du code MPI+Multithread !

ParSec Sweep Performances

- Vectorisation des tâches via Eigen

3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep Performances

- Vectorisation des tâches via Eigen
- ParSec pour générer MPI+Threads

3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep Performances

- Vectorisation des tâches via Eigen
- ParSec pour générer MPI+Threads
- 3 niveaux de parallélismes.

3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep Performances

- Vectorisation des tâches via Eigen
- ParSec pour générer MPI+Threads
- 3 niveaux de parallélismes.
- 6.1 Tflop/s on 768 cores $\rightarrow$ 33.9% peak !

3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep Performances

- Vectorisation des tâches via Eigen
- ParSec pour générer MPI+Threads
- 3 niveaux de parallélismes.
- 6.1 Tflop/s on 768 cores $\rightarrow$ 33.9% peak !
- Big case : 26-group PWR with 1.02×10^{12} DoFs using 1526 cores

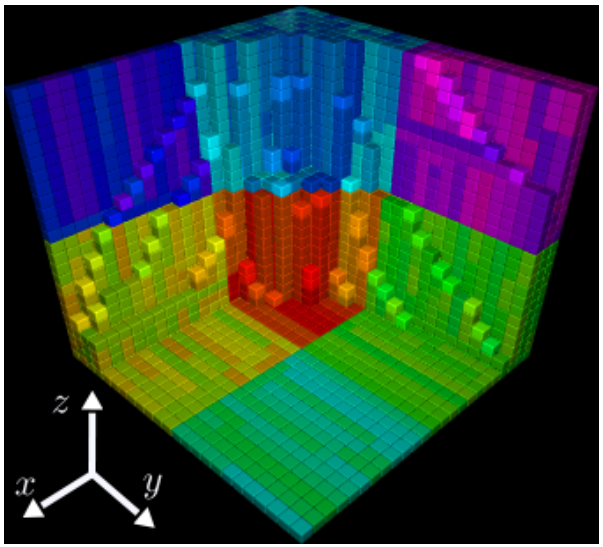
3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep Performances

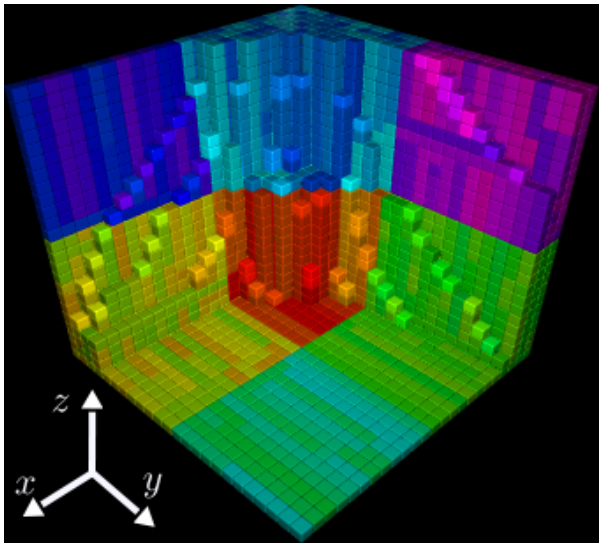
- Vectorisation des tâches via Eigen
- ParSec pour générer MPI+Threads
- 3 niveaux de parallélismes.
- 6.1 Tflop/s on 768 cores $\rightarrow$ 33.9% peak !
- Big case : 26-group PWR with 1.02×10^{12} DoFs using 1526 cores
- Runtime : StarPU (dynamic),HPX,...

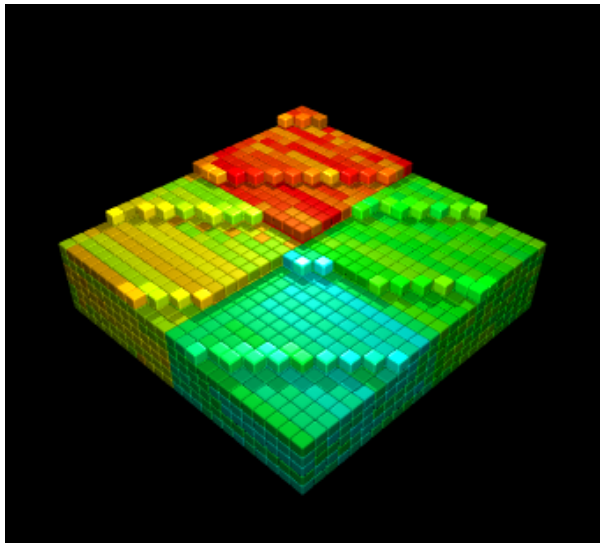
3D Cartesian Transport Sweep for Massively Parallel Architectures with PaRSEC
Salli Moustafa ; Mathieu Faverge ; Laurent Plagne ; Pierre Ramet
Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International

ParSec Sweep : Animation

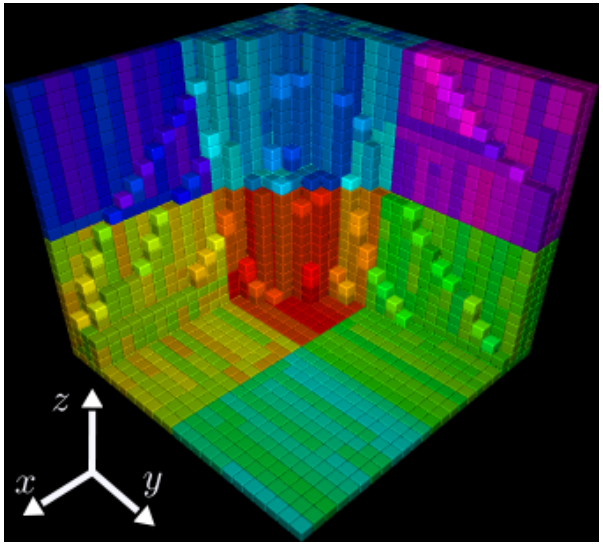


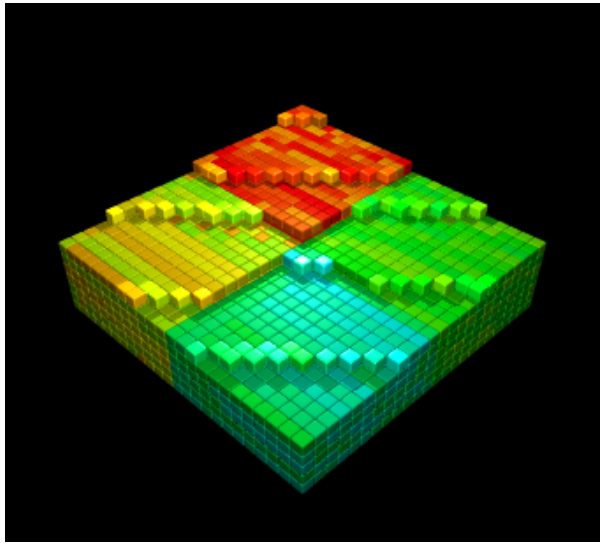
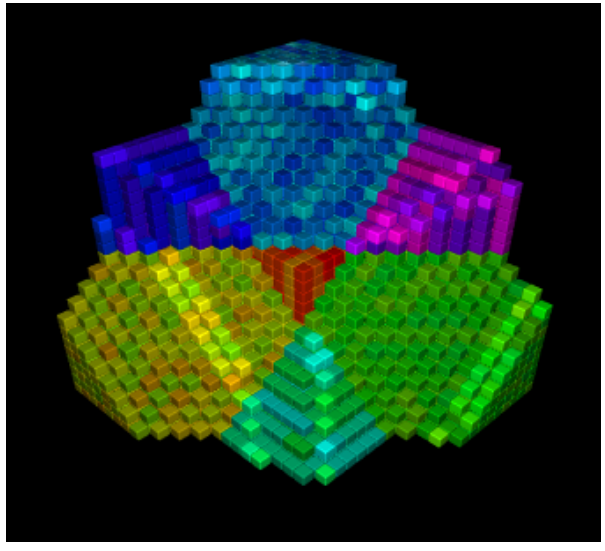
ParSec Sweep : Animation





ParSec Sweep : Animation



4. Résumé et tendances

Résumé (final)

- Explosion des performances (futur ?)

Résumé (final)

- Explosion des performances (futur ?)
- Forte baisse de l'efficacité des codes ($< 1\%$)

Résumé (final)

- Explosion des performances (futur ?)
- Forte baisse de l'efficacité des codes ($< 1\%$)
- 3 niveaux de //isme : Distributed/Shared/SIMD

Résumé (final)

- Explosion des performances (futur ?)
- Forte baisse de l'efficacité des codes ($< 1\%$)
- 3 niveaux de //isme : Distributed/Shared/SIMD
- //isme hétérogène : CPU, GPU,...(TPU ?)

Résumé (final)

- Explosion des performances (futur ?)
- Forte baisse de l'efficacité des codes ($< 1\%$)
- 3 niveaux de //isme : Distributed/Shared/SIMD
- //isme hétérogène : CPU, GPU,...(TPU ?)
- Mélange de paradigmes de programmation
  - MPI,MT,Stream Computing,eDSLs,Task based,Runtime

Résumé (final)

- Explosion des performances (futur ?)
- Forte baisse de l'efficacité des codes ($< 1\%$)
- 3 niveaux de //isme : Distributed/Shared/SIMD
- //isme hétérogène : CPU, GPU,...(TPU ?)
- Mélange de paradigmes de programmation
  - MPI,MT,Stream Computing,eDSLs,Task based,Runtime
- Enjeu dominant : Mémoire/Données !
  - Débit,Latence,Placement

Tendances ?

- Les méthodes évoluent :

Tendances ?

- Les méthodes évoluent :
  - constness et //isme

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - $\rightarrow$ programmation fonctionnelle.

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - → programmation fonctionnelle.
- Dag et Runtime

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - → programmation fonctionnelle.
- Dag et Runtime
- Cache programmable

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - → programmation fonctionnelle.
- Dag et Runtime
- Cache programmable
- Unités spécialisées (render script)

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - → programmation fonctionnelle.
- Dag et Runtime
- Cache programmable
- Unités spécialisées (render script)
- Dynamique/Statique → JIT

Tendances ?

- Les méthodes évoluent :
  - constness et //isme
  - fonctions d'ordre supérieur
  - lambda
  - → programmation fonctionnelle.
- Dag et Runtime
- Cache programmable
- Unités spécialisées (render script)
- Dynamique/Statique → JIT
- Les algorithmes évoluent !

Merci pour votre attention !

Vos questions sont les bienvenues ;)