FAST AND FURIOUS THINGS IN AGRUM/PYAGRUM

CHRISTOPHE GONZALES

LIS – AIX-MARSEILLE UNIVERSITÉ

Outline

Parallelization in Learning algorithms

Outline

- Parallelization in Learning algorithms
- Parallelization in Inferences

Outline

- Parallelization in Learning algorithms
- Parallelization in Inferences
- aGrUM's multithreading facility

BN structure learning : Greedy Hill Climbing

$\mathcal{G}_{best} \leftarrow$ initial graph (empty)
$sc_{best} \leftarrow \text{Score}(\mathcal{G}_{best})$

repeat
  found $\leftarrow$ false
  foreach $\mathcal{G}' \in$ *neighborhood of* $\mathcal{G}_{best}$ **do**
    $sc' \leftarrow \text{Score}(\mathcal{G}')$
    if $sc' > sc_{best}$ **then**
      $\mathcal{G}_{best} \leftarrow \mathcal{G}'$, $sc_{best} \leftarrow sc'$
      found $\leftarrow$ true
until *found* $=$ *false*;

return $\mathcal{G}_{best}$

BN structure learning : Greedy Hill Climbing

$\mathcal{G}_{best} \leftarrow$ initial graph (empty)
$sc_{best} \leftarrow$ Score($\mathcal{G}_{best}$)

repeat
  found $\leftarrow$ false
  foreach $\mathcal{G}' \in$ *neighborhood of* $\mathcal{G}_{best}$ **do**
    $sc' \leftarrow$ Score($\mathcal{G}'$)
    if $sc' > sc_{best}$ **then**
      $\mathcal{G}_{best} \leftarrow \mathcal{G}'$, $sc_{best} \leftarrow sc'$
      found $\leftarrow$ true
until *found* $=$ *false*;

return $\mathcal{G}_{best}$

► 2 parallelization opportunities :

BN structure learning : Greedy Hill Climbing

$\mathcal{G}_{best} \leftarrow$ initial graph (empty)
$sc_{best} \leftarrow \text{Score}(\mathcal{G}_{best})$

repeat
  found $\leftarrow$ false
  foreach $\mathcal{G}' \in$ *neighborhood of* $\mathcal{G}_{best}$ **do**
    $sc' \leftarrow \text{Score}(\mathcal{G}')$
    if $sc' > sc_{best}$ **then**
      $\mathcal{G}_{best} \leftarrow \mathcal{G}'$, $sc_{best} \leftarrow sc'$
      found $\leftarrow$ true
until $found = false$;

return $\mathcal{G}_{best}$

- 2 parallelization opportunities :
  - One thread per graph $\mathcal{G}'$

BN structure learning : Greedy Hill Climbing

$\mathcal{G}_{best} \leftarrow$ initial graph (empty)
$\mathrm{sc}_{best} \leftarrow \mathrm{Score}(\mathcal{G}_{best})$

repeat
  found $\leftarrow$ false
  foreach $\mathcal{G}' \in$ *neighborhood of* $\mathcal{G}_{best}$ **do**
    $\mathrm{sc}' \leftarrow \mathrm{Score}(\mathcal{G}')$
    if $sc' > sc_{best}$ **then**
      $\mathcal{G}_{best} \leftarrow \mathcal{G}'$, $\mathrm{sc}_{best} \leftarrow \mathrm{sc}'$
      found $\leftarrow$ true
until $found = false$;

return $\mathcal{G}_{best}$

- 2 parallelization opportunities :
  - One thread per graph $\mathcal{G}'$
  - Several threads for each score

BN structure learning : Greedy Hill Climbing

$\mathcal{G}_{best} \leftarrow$ initial graph (empty)
$sc_{best} \leftarrow$ Score($\mathcal{G}_{best}$)

repeat
    found $\leftarrow$ false
    foreach $\mathcal{G}' \in$ *neighborhood of* $\mathcal{G}_{best}$ **do**
        $sc' \leftarrow$ Score($\mathcal{G}'$)
        if $sc' > sc_{best}$ **then**
            $\mathcal{G}_{best} \leftarrow \mathcal{G}'$, $sc_{best} \leftarrow sc'$
            found $\leftarrow$ true
until *found* $=$ *false*;

return $\mathcal{G}_{best}$

- 2 parallelization opportunities :
  - One thread per graph $\mathcal{G}'$
  - Several threads for each score

Parallelizing the scores

The BD score

$$\mathrm{Score}_{BD}(X_i|\mathbf{Pa}(X_i),\mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij}+\alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk}+\alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$

Parallelizing the scores

The BD score

$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i), \mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$

-

Parallelizing the scores

The BD score

$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i), \mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$

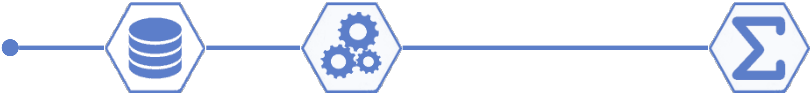
Parallelizing the scores

The BD score

$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i), \mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$

The BD score

$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i), \mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$



Parallelizing the scores

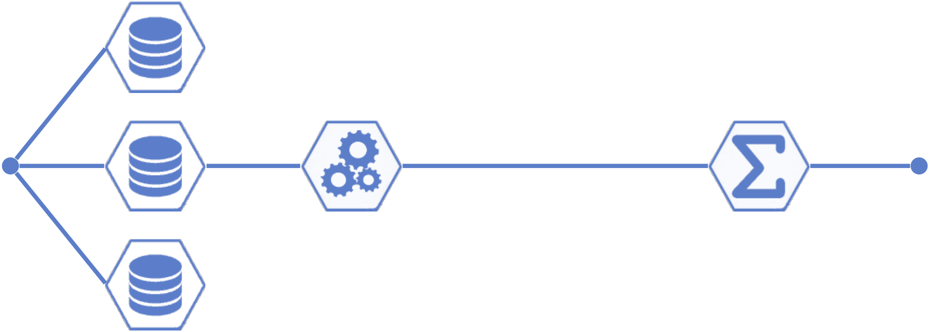
The BD score

$$\mathrm{Score}_{BD}(X_i | \mathbf{Pa}(X_i), \mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$

Parallelizing the scores

The BD score

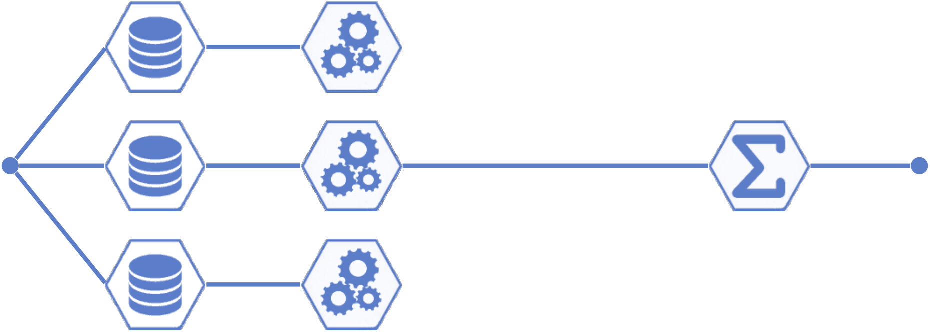
$$\mathrm{Score}_{BD}(X_i|\mathbf{Pa}(X_i),\mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij}+\alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk}+\alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$



Parallelizing the scores

The BD score

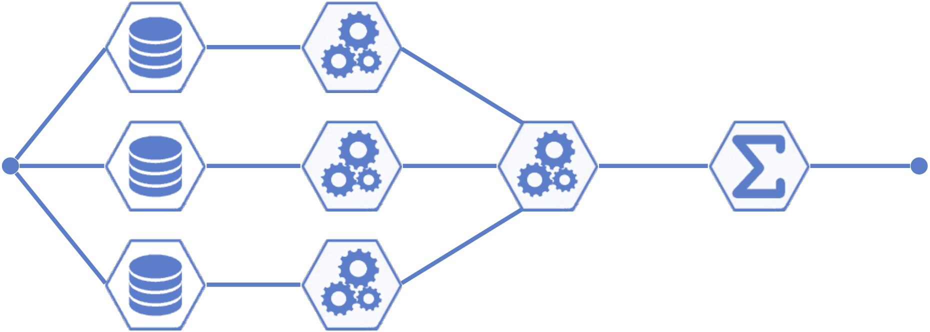
$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i),\mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij}+\alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk}+\alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$



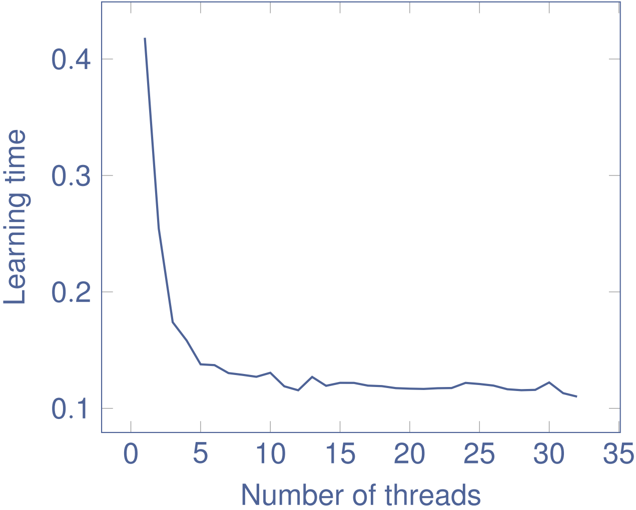
Parallelizing the scores

The BD score

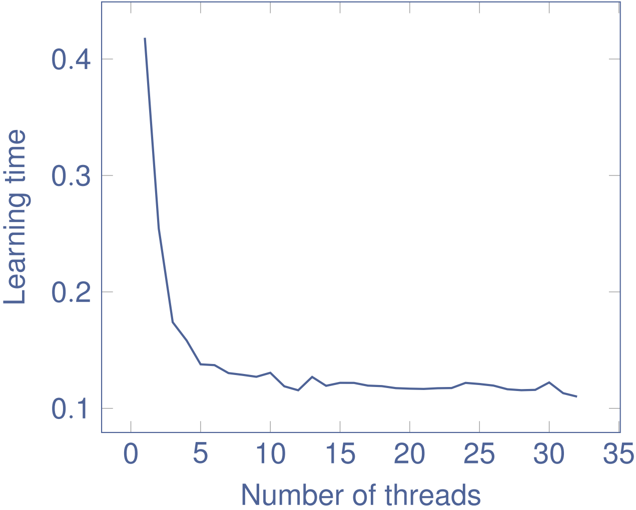
$$\text{Score}_{BD}(X_i|\mathbf{Pa}(X_i),\mathbf{D}) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij}+\alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk}+\alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$



Alarm : learning structure with 10K records

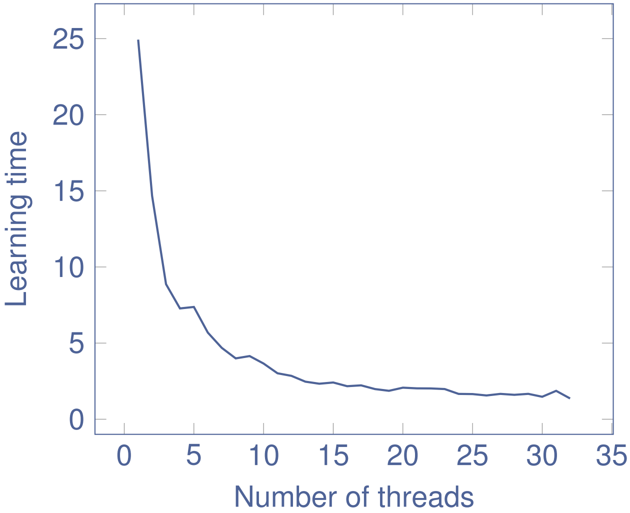


Alarm : learning structure with 10K records



- **aGrUM's rule :** Number of records per thread ≥ 512

Alarm : learning structure with 200K records



Number of threads in pyAgrum : the rationale

aGrUM's rules :

1. pyAgrum manages a ≪ package-wide ≫ number of threads

   $\Longrightarrow$ by default, same number for all multithreaded objects

Number of threads in pyAgrum : the rationale

aGrUM's rules :

1. pyAgrum manages a « package-wide » number of threads

   $\Longrightarrow$ by default, same number for all multithreaded objects

2. Objects can override this number

pyAgrum's Package-wide API

```
import pyAgrum as gum
```

- **Package-wide functions :**

function	meaning
`gum.getMaxNumberOfThreads`	The number of processors of the computer

```
import pyAgrum as gum
```

- **Package-wide functions :**

function	meaning
`gum.getMaxNumberOfThreads`	The number of processors of the computer

```python
import pyAgrum as gum
```

- **Package-wide functions :**

function	meaning
`gum.getMaxNumberOfThreads`	The number of processors of the computer
`gum.getNumberOfLogicalCores`	The number of processors

```python
import pyAgrum as gum
```

- **Package-wide functions :**

function	meaning
gum.getMaxNumberOfThreads	The number of processors of the computer
gum.getNumberOfLogicalCores	The number of processors
gum.getNumberOfThreads	The number of threads used by default by pyAgrum objects

```python
import pyAgrum as gum
```

- **Package-wide functions :**

function	meaning
`gum.getMaxNumberOfThreads`	The number of processors of the computer
`gum.getNumberOfLogicalCores`	The number of processors
`gum.getNumberOfThreads`	The number of threads used by default by pyAgrum objects
`gum.setNumberOfThreads`	Sets the number of threads used by default

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())
```

Using the pyAgrum-wide API – an example

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())
```

```
Nb procs = 64
```

Using the pyAgrum-wide API – an example

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())

# Default number of threads for all pyAgrum objects
print("Nb used =", gum.getNumberOfThreads())
```

```
Nb procs = 64
```

Using the pyAgrum-wide API – an example

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())

# Default number of threads for all pyAgrum objects
print("Nb used =", gum.getNumberOfThreads())
```

```
Nb procs = 64
Nb used = 64
```

Using the pyAgrum-wide API – an example

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())

# Default number of threads for all pyAgrum objects
print("Nb used =", gum.getNumberOfThreads())

# Changing this default number
gum.setNumberOfThreads(10)
```

```
Nb procs = 64
Nb used = 64
```

Using the pyAgrum-wide API – an example

```python
import pyAgrum as gum

print("Nb procs =", gum.getMaxNumberOfThreads())

# Default number of threads for all pyAgrum objects
print("Nb used =", gum.getNumberOfThreads())

# Changing this default number
gum.setNumberOfThreads(10)

# Default number of threads for all pyAgrum objects
print("Nb used =", gum.getNumberOfThreads())
```

```
Nb procs = 64
Nb used = 64
Nb used = 10
```

Overriding the pyAgrum multithreading setting

- **Object overriding methods :**

- **Object overriding methods :**

function	meaning
`obj.getNumberOfThreads`	returns the current number of threads used by `obj`

- **Object overriding methods :**

function	meaning
`obj.getNumberOfThreads`	returns the current number of threads used by `obj`
`obj.setNumberOfThreads`	changes the number of threads used by `obj`

Overriding the pyAgrum multithreading setting

- **Object overriding methods :**

function	meaning
`obj.getNumberOfThreads`	returns the current number of threads used by `obj`
`obj.setNumberOfThreads`	changes the number of threads used by `obj`
`obj.isGumNumberOfThreadsOverriden`	indicates whether `obj` uses its own number or that of pyAgrum

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

```
pyAgrum threads = 64
Learner threads = 64
Learner override = False
```

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# changing the number of threads only for the learner
learner.setNumberOfThreads(10)
print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

```
pyAgrum threads = 64
Learner threads = 64
Learner override = False
```

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# changing the number of threads only for the learner
learner.setNumberOfThreads(10)
print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

```
pyAgrum threads = 64
Learner threads = 64
Learner override = False
```

```
pyAgrum threads = 64
Learner threads = 10
Learner override = True
```

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# changing the number of threads only for the learner
learner.setNumberOfThreads(10)
print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# making the learner use the package-wide number again
learner.setNumberOfThreads(0) # 0 = package-wide
print("Learner threads =", learner.getNumberOfThreads(),
      " Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

```
pyAgrum threads = 64
Learner threads = 64
Learner override = False
```

```
pyAgrum threads = 64
Learner threads = 10
Learner override = True
```

Using the object API – an example

```python
import pyAgrum as gum
learner = gum.BNLearner("data/alarm.csv")

print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# changing the number of threads only for the learner
learner.setNumberOfThreads(10)
print("pyAgrum threads =", gum.getNumberOfThreads())
print("Learner threads =", learner.getNumberOfThreads())
print("Learner override =",
      learner.isGumNumberOfThreadsOverriden())

# making the learner use the package-wide number again
learner.setNumberOfThreads(0) # 0 = package-wide
print("Learner threads =", learner.getNumberOfThreads(),
      " Learner override =",
      learner.isGumNumberOfThreadsOverriden())
```

```
pyAgrum threads = 64
Learner threads = 64
Learner override = False
```

```
pyAgrum threads = 64
Learner threads = 10
Learner override = True
```

```
Learner threads = 64  Learner override = False
```

Different situations :

- Performing just one learning :
  use pyAgrum number of threads

Toward an optimal choice of the number of threads

Different situations :

- Performing just one learning :
  use pyAgrum number of threads
- Performing a large amount of learning experiments :
  use 1 thread per BNlearner
  perform experiments in parallel

Toward an optimal choice of the number of threads

Different situations :

- Performing just one learning :
  use pyAgrum number of threads
- Performing a large amount of learning experiments :
  use 1 thread per BNlearner
  perform experiments in parallel
- Performing a small amount K of learnings :
  let nb processors $\approx A \times B$, with K multiple of A
  perform A experiments in parallel
  use B thread per BNlearner

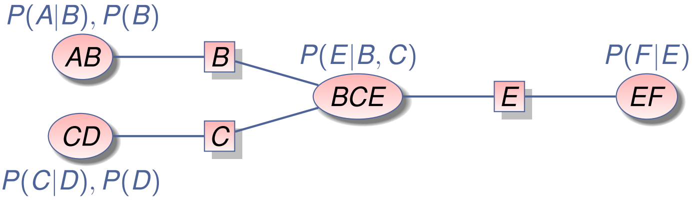
Next multithreading steps. . .

- Allow Databases to be column-wise instead of row-wise

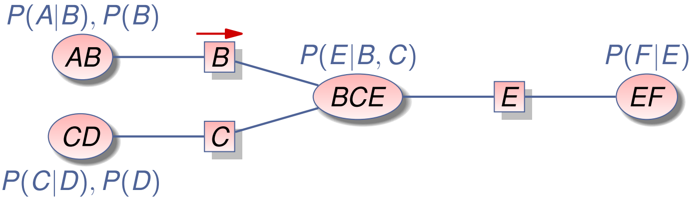
  $\Longrightarrow$ improved cacheline use

Parallelization in Inferences

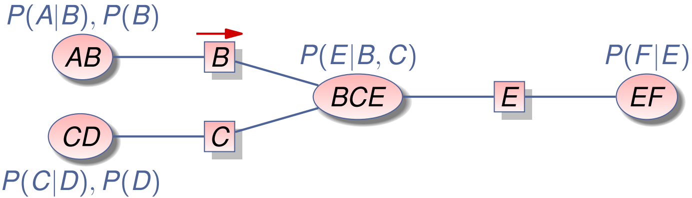
New exact inference architecture (1/2)



New exact inference architecture (1/2)

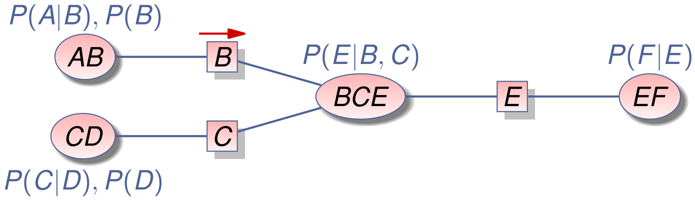


New exact inference architecture (1/2)



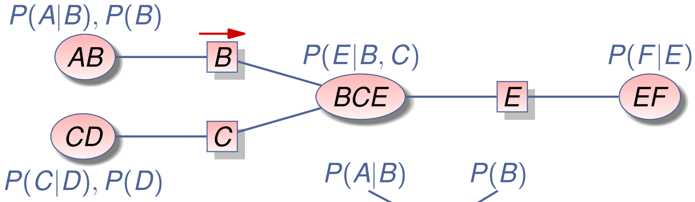
1. $P(A,B) = P(A|B) \times P(B)$

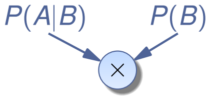
New exact inference architecture (1/2)



1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$

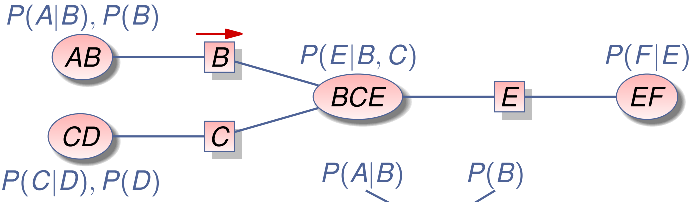
New exact inference architecture (1/2)

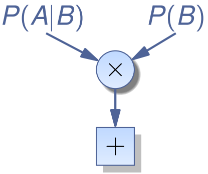




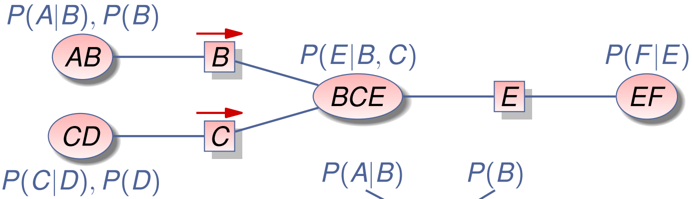
1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$

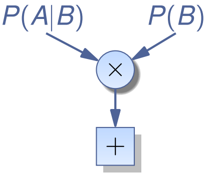
New exact inference architecture (1/2)





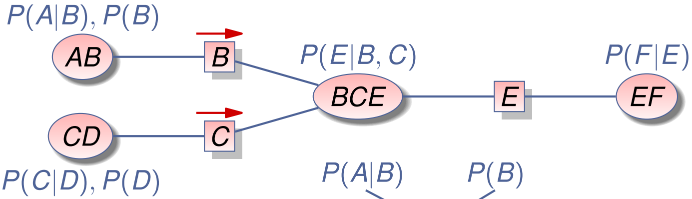
1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$

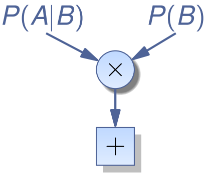




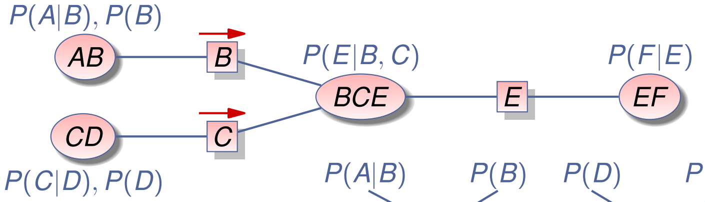
1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$

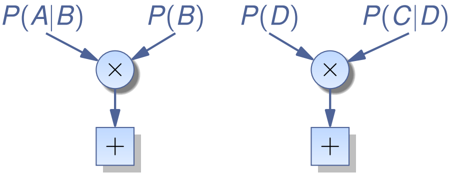
New exact inference architecture (1/2)





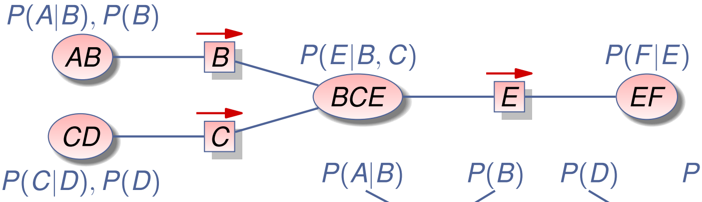
1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$
3. $P(C,D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C,D)$

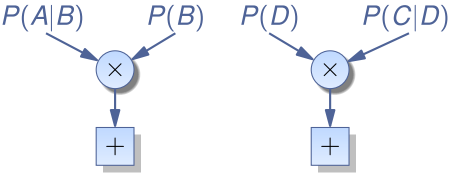




1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$
3. $P(C,D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C,D)$

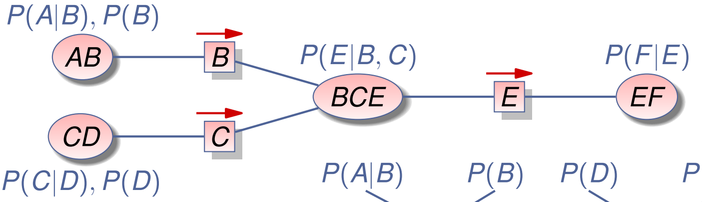
New exact inference architecture (1/2)

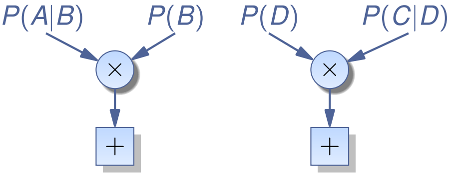




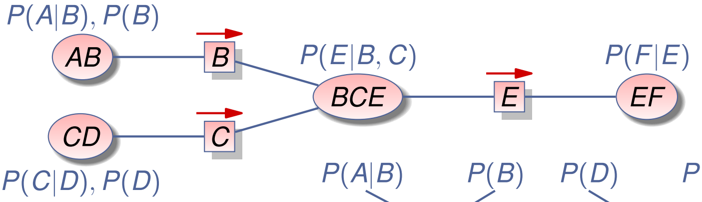
1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$
3. $P(C, D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C, D)$

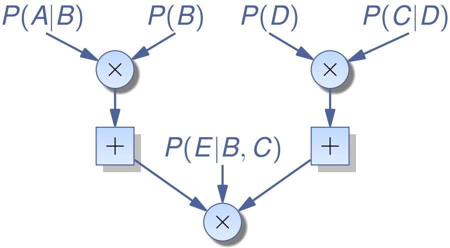
New exact inference architecture (1/2)





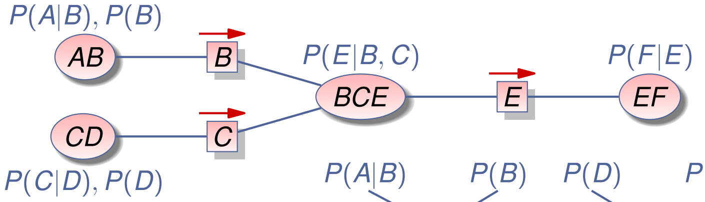
1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$
3. $P(C, D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C, D)$
5. $P(B, C, E) = P(E|B, C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B, C, E)$

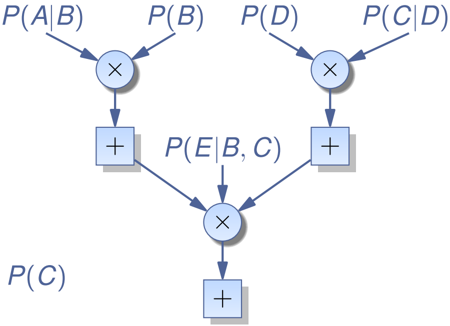




1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$
3. $P(C,D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C,D)$
5. $P(B,C,E) = P(E|B,C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B,C,E)$

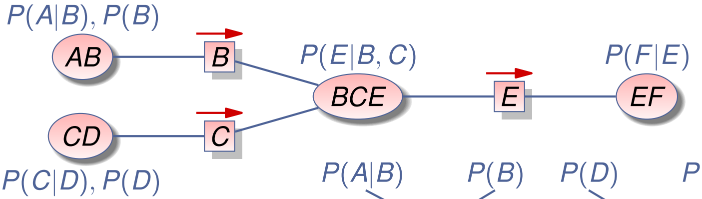
New exact inference architecture (1/2)

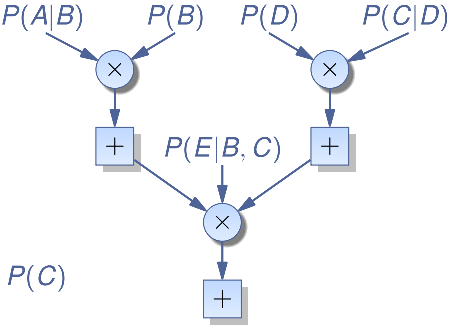




1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$
3. $P(C, D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C, D)$
5. $P(B, C, E) = P(E|B, C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B, C, E)$

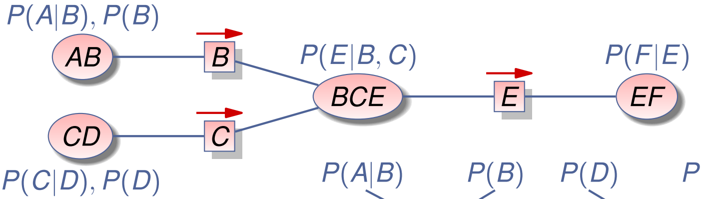
New exact inference architecture (1/2)

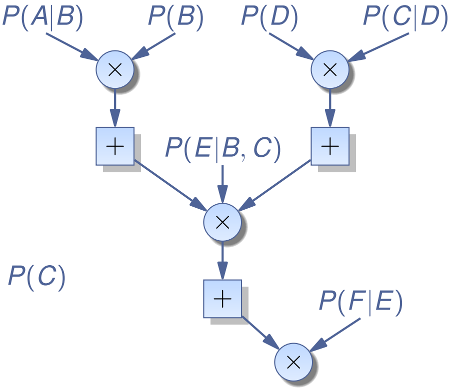




1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$
3. $P(C,D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C,D)$
5. $P(B,C,E) = P(E|B,C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B,C,E)$
7. $P(E,F) = P(F|E) \times P(E)$
8. $P(F) = \sum_E P(E,F)$

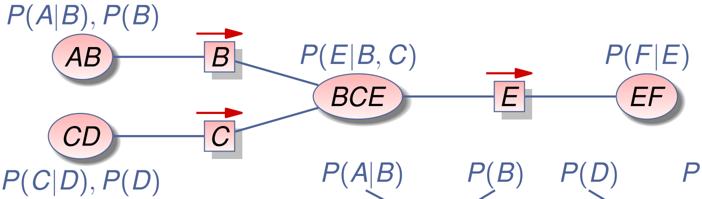
New exact inference architecture (1/2)

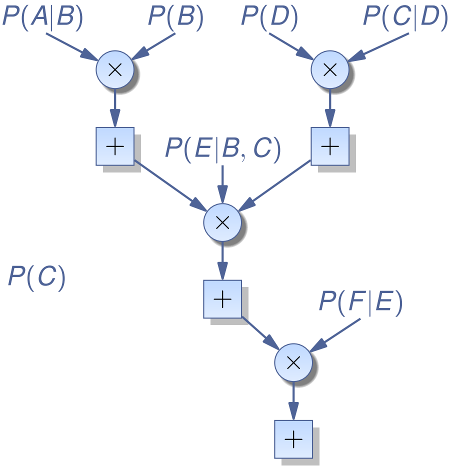




1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$
3. $P(C, D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C, D)$
5. $P(B, C, E) = P(E|B, C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B, C, E)$
7. $P(E, F) = P(F|E) \times P(E)$
8. $P(F) = \sum_E P(E, F)$

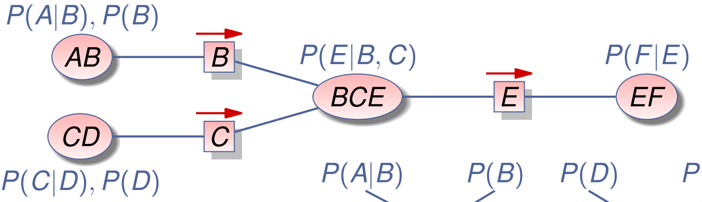
New exact inference architecture (1/2)

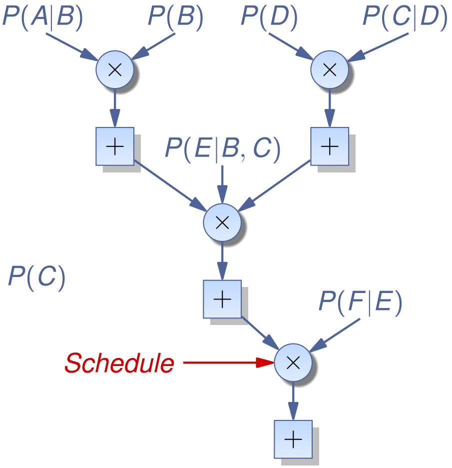




1. $P(A, B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A, B)$
3. $P(C, D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C, D)$
5. $P(B, C, E) = P(E|B, C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B, C, E)$
7. $P(E, F) = P(F|E) \times P(E)$
8. $P(F) = \sum_E P(E, F)$

New exact inference architecture (1/2)





1. $P(A,B) = P(A|B) \times P(B)$
2. $P(B) = \sum_A P(A,B)$
3. $P(C,D) = P(C|D) \times P(D)$
4. $P(C) = \sum_D P(C,D)$
5. $P(B,C,E) = P(E|B,C) \times P(B) \times P(C)$
6. $P(E) = \sum_{B,C} P(B,C,E)$
7. $P(E,F) = P(F|E) \times P(E)$
8. $P(F) = \sum_E P(E,F)$

New exact inference architecture (2/2)

1. Create a junction tree

New exact inference architecture (2/2)

1. Create a junction tree
2. Create a schedule from the JT

New exact inference architecture (2/2)

1. Create a junction tree
2. Create a schedule from the JT
3. Execute the schedule

New exact inference architecture (2/2)

1. Create a junction tree
2. Create a schedule from the JT
3. Execute the schedule

- **Used by LazyPropagation and Shafer-Shenoy**

New exact inference architecture (2/2)

1. Create a junction tree
2. Create a schedule from the JT
3. Execute the schedule

- **Used by LazyPropagation and Shafer-Shenoy**
- **2 schedulers :**
  - Sequential scheduler
  - Parallel scheduler

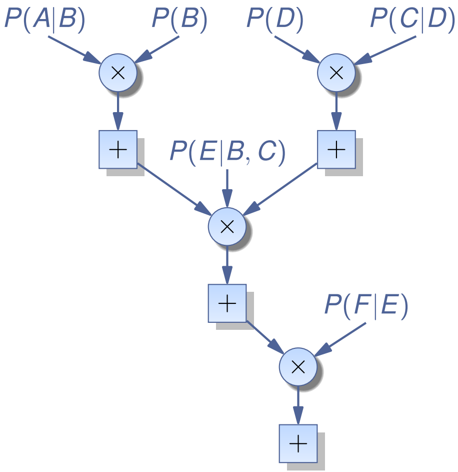
New exact inference architecture (2/2)

1. Create a junction tree
2. Create a schedule from the JT
3. Execute the schedule

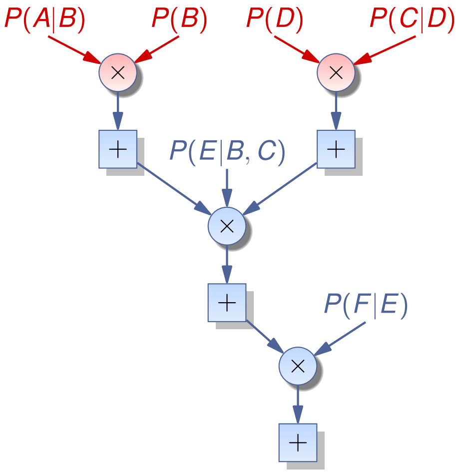
- **Used by LazyPropagation and Shafer-Shenoy**
- **2 schedulers :**
  - Sequential scheduler
  - Parallel scheduler
- **1 Rule :** Use the sequential scheduler if and only if :

  1 thread or nb elementary operations $< 10^6$

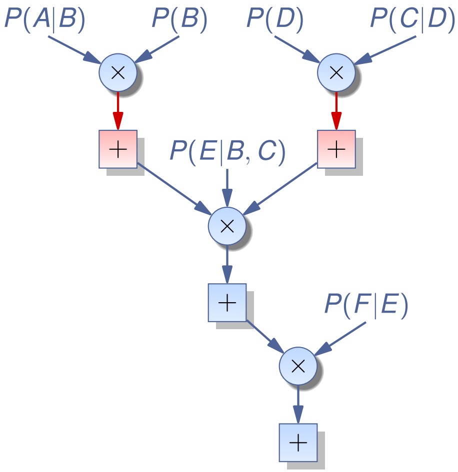
Parallel scheduler – an example



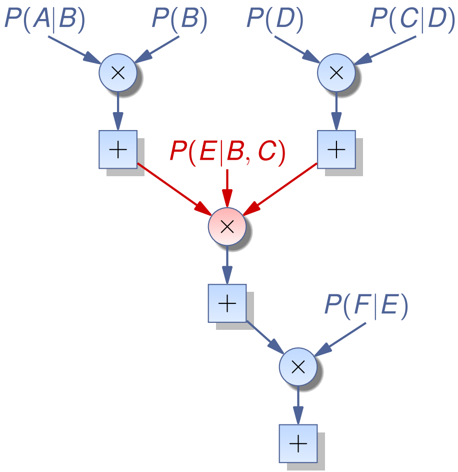
Parallel scheduler – an example



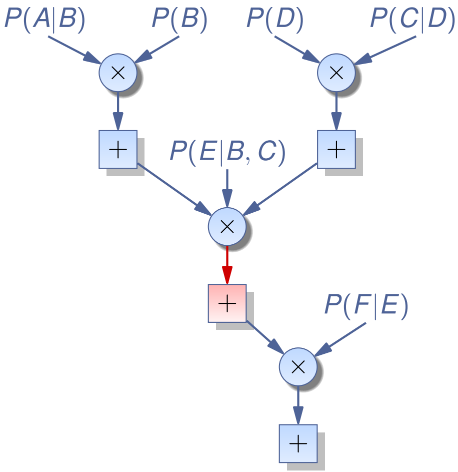
Parallel scheduler – an example



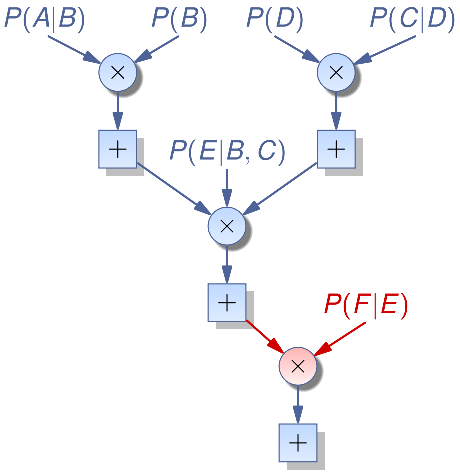
Parallel scheduler – an example



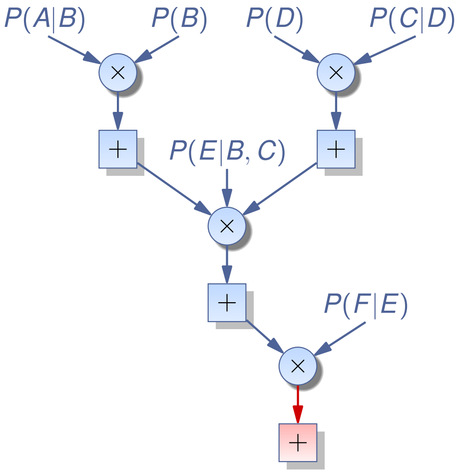
Parallel scheduler – an example



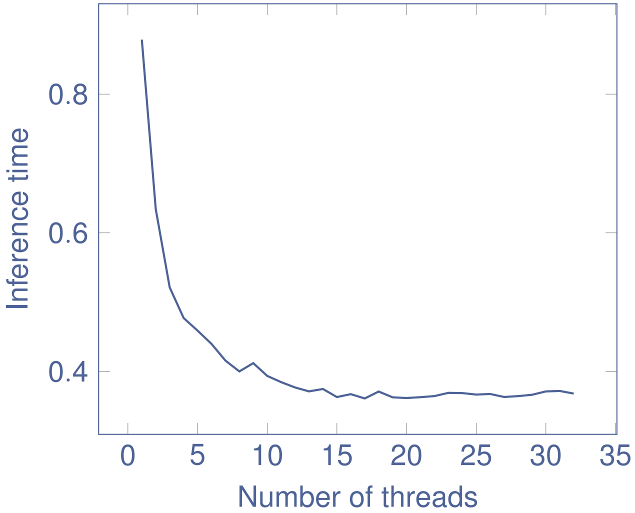
Parallel scheduler – an example



Parallel scheduler – an example



LazyPropagation's inferences on Munin4



Toward an optimal use of schedulers (1/2)

- **Synchronization mechanisms :**
  - Mutexes / locks
  - Condition variables
  - Atomics

Toward an optimal use of schedulers (1/2)

- **Synchronization mechanisms :**
  - Mutexes / locks
  - Condition variables
  - Atomics

Toward an optimal use of schedulers (1/2)

- **Synchronization mechanisms :**
  - Mutexes / locks
  - Condition variables
  - Atomics

  . . . but there is still an overhead.

Toward an optimal use of schedulers (1/2)

- **Synchronization mechanisms :**
  - Mutexes / locks
  - Condition variables
  - Atomics

  . . . but there is still an overhead.

- **Experiments on large BNs :**
  - Major time reduction from 1 to 4-6 threads
  - More limited gain above 6 threads

Toward an optimal use of schedulers (1/2)

- **Synchronization mechanisms :**
  - Mutexes / locks
  - Condition variables
  - Atomics

  . . . but there is still an overhead.

- **Experiments on large BNs :**
  - Major time reduction from 1 to 4-6 threads
  - More limited gain above 6 threads
  - Explanation : clique sizes imbalanced

Different situations for large BNs :

- Performing just one inference :

  parallellize with pyAgrum number of threads

Toward an optimal use of schedulers (2/2)

Different situations for large BNs :

- Performing just one inference :

  parallellize with pyAgrum number of threads

- Performing a large amount of inference experiments :

  use 1 thread per LazyPropagation instance

  perform experiments in parallel

Toward an optimal use of schedulers (2/2)

Different situations for large BNs :

- Performing just one inference :
  parallellize with pyAgrum number of threads
- Performing a large amount of inference experiments :
  use 1 thread per LazyPropagation instance
  perform experiments in parallel
- Performing a small amount K of inferences :
  perform $K/4$ experiments in parallel
  use 4 threads per LazyPropagation instance

Objects contained into schedules :

- ScheduleMultiDim : abstraction of potentials

Objects contained into schedules :

- ScheduleMultiDim : abstraction of potentials
- ScheduleBinaryCombination : $T \otimes T \mapsto T$
- ScheduleProjection : $T \Downarrow v \mapsto T$

Schedules and their operators

Objects contained into schedules :

- ScheduleMultiDim : abstraction of potentials
- ScheduleBinaryCombination : $T \otimes T \mapsto T$
- ScheduleProjection : $T \Downarrow v \mapsto T$
- ScheduleDeletion : remove a ScheduleMultiDim from memory
- ScheduleStorage : store a ScheduleMultiDim into a container

Schedules and their operators

Objects contained into schedules :

- ScheduleMultiDim : abstraction of potentials
- ScheduleBinaryCombination : $T \otimes T \mapsto T$
- ScheduleProjection : $T \Downarrow v \mapsto T$
- ScheduleDeletion : remove a ScheduleMultiDim from memory
- ScheduleStorage : store a ScheduleMultiDim into a container

$\Longrightarrow$ very general-purpose

Next multithreading steps. . .

- Expose `numberOfOperations` and `setMaxMemory` to pyAgrum

Next multithreading steps. . .

- Expose `numberOfOperations` and `setMaxMemory` to pyAgrum
- Reduce the overhead of using schedulers

Next multithreading steps. . .

- Expose `numberOfOperations` and `setMaxMemory` to pyAgrum
- Reduce the overhead of using schedulers
- Add schedulers parallelizing both operators and operations

  $\Longrightarrow$ requires splitting Potential operators computations

Next multithreading steps. . .

- Expose `numberOfOperations` and `setMaxMemory` to pyAgrum
- Reduce the overhead of using schedulers
- Add schedulers parallelizing both operators and operations

  $\Longrightarrow$ requires splitting Potential operators computations
- Add a scheduler exploiting GPU

  $\Longrightarrow$ requires ScheduleOperator for changing the order of variables in potentials aGrUM's multithreading facility

- Multithreaded objects support both openMP and STL threads

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used except if :
  - either the user compiled aGrUM with `--threads=stl` option
  - or the compiler does not support openMP

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used except if :
  - either the user compiled aGrUM with `--threads=stl` option
  - or the compiler does not support openMP
- Parallelism achieved by using `ThreadExecutor` instances

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used except if :
  - either the user compiled aGrUM with `--threads=stl` option
  - or the compiler does not support openMP
- Parallelism achieved by using `ThreadExecutor` instances
- Advantages :
  - Multithreaded objects are agnostic

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used except if :
  - either the user compiled aGrUM with `--threads=stl` option
  - or the compiler does not support openMP
- Parallelism achieved by using `ThreadExecutor` instances
- Advantages :
  - Multithreaded objects are agnostic
  - Exceptions can be catched

openMP vs. STL threads

- Multithreaded objects support both openMP and STL threads
- By default, openMP is used except if :
  - either the user compiled aGrUM with `--threads=stl` option
  - or the compiler does not support openMP
- Parallelism achieved by using `ThreadExecutor` instances
- Advantages :
  - Multithreaded objects are agnostic
  - Exceptions can be catched
  - When one thread : no overhead

ThreadExecutors – an example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads) -> void {
  std::cout << "thread #" << this_thread << std::endl;
};

try {
  gum::ThreadExecutor::execute(5, func);
} catch(...) {
  std::cout << "Exception catched" << std::endl;
}
```

ThreadExecutors – an example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads) -> void {
  std::cout << "thread #" << this_thread << std::endl;
};

try {
  gum::ThreadExecutor::execute(5, func);
} catch(...) {
  std::cout << "Exception catched" << std::endl;
}
```

```
thread #0
thread #4
thread #3
thread #thread #2
1
```

ThreadExecutors – an example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads) -> void {
  std::cout << "thread #" << this_thread << std::endl;
};

try {
  gum::ThreadExecutor::execute(5, func);
} catch(...) {
  std::cout << "Exception catched" << std::endl;
}
```

```
thread #0
thread #4
thread #3
thread #thread #2
1
```

$\Longrightarrow$ Exceptions can be catched in Python !

ThreadExecutors – another example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads,
               int nb,
               const std::string& str) -> void {
  std::cout << str << nb << " #"
            << this_thread << std::endl;
};

gum::ThreadExecutor::execute(5, func, 8, "thread ");
```

ThreadExecutors – another example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads,
               int nb,
               const std::string& str) -> void {
  std::cout << str << nb << " #"
            << this_thread << std::endl;
};

gum::ThreadExecutor::execute(5, func, 8, "thread ");
```

```
thread 8 #thread 0thread 8 #4thread 8 #2
8
thread 8 #3
 #1
```

ThreadExecutors – another example

```
auto func = [](const std::size_t this_thread,
               const std::size_t nb_threads,
               int nb,
               const std::string& str) -> void {
  std::cout << str << nb << " #"
            << this_thread << std::endl;
};

gum::ThreadExecutor::execute(5, func, 8, "thread ");
```

```
thread 8 #thread 0thread 8 #4thread 8 #2
8
thread 8 #3
 #1
```

⟹ Functions can have as many parameters as wished

Only constraint : first 2 params : `this_thread` and `nb_threads`

Conclusion

- Parallelism speeds-up learning and inference computations

Conclusion

- Parallelism speeds-up learning and inference computations
- Many things to do yet for inferences
  - in particular, check `const` objects...

Conclusion

- Parallelism speeds-up learning and inference computations
- Many things to do yet for inferences
  - in particular, check `const` objects…
  - Reduce schedules' creations overhead

Conclusion

- Parallelism speeds-up learning and inference computations
- Many things to do yet for inferences
  - in particular, check `const` objects...
  - Reduce schedules' creations overhead
  - inferences over GPU