FAST AND FURIOUS THINGS IN aGrUM/pyAgrUM

CHRISTOPHE GONZALES

LIS – AIX-MARSEILLE UNIVERSITÉ
Parallelization in Learning algorithms
Parallelization in Learning algorithms

Parallelization in Inferences
Outline

- Parallelization in Learning algorithms
- Parallelization in Inferences
- aGrUM’s multithreading facility
1 \( G_{\text{best}} \leftarrow \text{initial graph (empty)} \)
2 \( \text{sc}_{\text{best}} \leftarrow \text{Score}(G_{\text{best}}) \)
3
4 \textbf{repeat}
5 \hspace{1em} \text{found} \leftarrow \text{false}
6 \hspace{1em} \textbf{foreach} \ G' \in \text{neighborhood of } G_{\text{best}} \hspace{1em} \textbf{do}
7 \hspace{2em} \text{sc}' \leftarrow \text{Score}(G')
8 \hspace{2em} \textbf{if } \text{sc}' > \text{sc}_{\text{best}} \hspace{1em} \textbf{then}
9 \hspace{3em} G_{\text{best}} \leftarrow G', \text{sc}_{\text{best}} \leftarrow \text{sc}'
10 \hspace{2em} \text{found} \leftarrow \text{true}
11 \hspace{1em} \textbf{until} \ \text{found} = \text{false};
12
13 \textbf{return } G_{\text{best}}
BN structure learning: Greedy Hill Climbing

1. $G_{\text{best}} \leftarrow \text{initial graph (empty)}$
2. $sc_{\text{best}} \leftarrow \text{Score}(G_{\text{best}})$

repeat

5. found $\leftarrow$ false
6. foreach $G' \in \text{neighborhood of } G_{\text{best}}$ do
7. \hspace{1em} $sc' \leftarrow \text{Score}(G')$
8. \hspace{2em} if $sc' > sc_{\text{best}}$ then
9. \hspace{3em} $G_{\text{best}} \leftarrow G'$, $sc_{\text{best}} \leftarrow sc'$
10. \hspace{3em} found $\leftarrow$ true

until found $=\ false$;

13. return $G_{\text{best}}$

▶ 2 parallelization opportunities:
BN structure learning: Greedy Hill Climbing

1. $G_{\text{best}} \leftarrow$ initial graph (empty)
2. $sc_{\text{best}} \leftarrow \text{Score}(G_{\text{best}})$
3. repeat
4.   found $\leftarrow$ false
5.   foreach $G'$ in neighborhood of $G_{\text{best}}$ do
6.     $sc' \leftarrow \text{Score}(G')$
7.     if $sc' > sc_{\text{best}}$ then
8.       $G_{\text{best}} \leftarrow G'$, $sc_{\text{best}} \leftarrow sc'$
9.     found $\leftarrow$ true
10. until found = false;
11. return $G_{\text{best}}$

▶ 2 parallelization opportunities:
12.     ► One thread per graph $G'$
1 $G_{best} \leftarrow \text{initial graph (empty)}$
2 $sc_{best} \leftarrow \text{Score}(G_{best})$
3
4 repeat
5      found $\leftarrow$ false
6     foreach $G'$ $\in$ neighborhood of $G_{best}$ do
7            $sc' \leftarrow \text{Score}(G')$
8            if $sc' > sc_{best}$ then
9                $G_{best} \leftarrow G'$, $sc_{best} \leftarrow sc'$
10           found $\leftarrow$ true
11 until found $= \text{false}$;
12 return $G_{best}$

▶ 2 parallelization opportunities:
  
  - One thread per graph $G'$
  - Several threads for each score
BN structure learning: Greedy Hill Climbing

1 \( G_{\text{best}} \leftarrow \) initial graph (empty)
2 \( sc_{\text{best}} \leftarrow \text{Score}(G_{\text{best}}) \)

4 repeat
5 \hspace{1em} \text{found} \leftarrow \text{false}
6 \hspace{1em} \textbf{foreach} \text{ } G' \in \text{neighborhood of } G_{\text{best}} \text{ do}
7 \hspace{2em} \text{sc'} \leftarrow \text{Score}(G')
8 \hspace{2em} \textbf{if} \text{ } sc' > sc_{\text{best}} \text{ then}
9 \hspace{3em} G_{\text{best}} \leftarrow G', \text{ sc}_{\text{best}} \leftarrow \text{sc'}
10 \hspace{3em} \text{found} \leftarrow \text{true}
11 \textbf{until} \text{ } \text{found} = \text{false};
12
13 \textbf{return} \ G_{\text{best}}

\[ 2 \text{ parallelization opportunities:} \]

\[ \text{▷ One thread per graph } G' \]
\[ \text{▷ Several threads for each score} \]
Parallelizing the scores

The BD score

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

The BD score

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

**The BD score**

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

The BD score

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

The BD score

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

The BD score

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

The BD score

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

The BD score

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

**The BD score**

$$\text{Score}_{BD}(X_i|\text{Pa}(X_i), D) = \prod_{j=1}^{q_i} \frac{\Gamma(\alpha_{ij})}{\Gamma(N_{ij} + \alpha_{ij})} \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + \alpha_{ijk})}{\Gamma(\alpha_{ijk})}$$
Alarm: learning structure with 10K records

Number of threads

Learning time

Number of threads

aGrUM's rule: Number of records per thread ≥ 512
Alberto: learning structure with 10K records

![Graph showing learning time vs. number of threads]

- **aGrUM's rule**: Number of records per thread \( \geq 512 \)
Alarm: learning structure with 200K records

![Graph showing the relationship between the number of threads and learning time. The learning time decreases significantly as the number of threads increases, reaching a plateau around 30 threads.](image-url)
aGrUM’s rules:

1. pyAgrum manages a «package-wide» number of threads
   → 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
   ➔ by default, same number for all multithreaded objects

2. Objects can override this number
import pyAgrum as gum

### Package-wide functions:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gum.getMaxNumberOfThreads</code></td>
<td>The number of processors of the computer</td>
</tr>
</tbody>
</table>
import pyAgrum as gum

**Package-wide functions:**

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gum.getMaxNumberOfThreads</td>
<td>The number of processors of the computer</td>
</tr>
</tbody>
</table>
import pyAgrum as gum

### Package-wide functions:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gum.getMaxNumberOfThreads</td>
<td>The number of processors of the computer</td>
</tr>
<tr>
<td>gum.getNumberOfLogicalCores</td>
<td>The number of processors</td>
</tr>
</tbody>
</table>
import `pyAgrum` as `gum`

### Package-wide functions:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gum.getMaxNumberOfThreads</code></td>
<td>The number of processors of the computer</td>
</tr>
<tr>
<td><code>gum.getNumberOfLogicalCores</code></td>
<td>The number of processors</td>
</tr>
<tr>
<td><code>gum.getNumberofThreads</code></td>
<td>The number of threads used by default by <code>pyAgrum</code> objects</td>
</tr>
</tbody>
</table>
```python
import pyAgrum as gum
```

### Package-wide functions:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>gum.getMaxNumberOfThreads</code></td>
<td>The number of processors of the computer</td>
</tr>
<tr>
<td><code>gum.getNumberOfLogicalCores</code></td>
<td>The number of processors</td>
</tr>
<tr>
<td><code>gum.getNumberOfThreads</code></td>
<td>The number of threads used by default by pyAgrum objects</td>
</tr>
<tr>
<td><code>gum.setNumberOfThreads</code></td>
<td>Sets the number of threads used by default</td>
</tr>
</tbody>
</table>
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())
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
```
import pyAgrum as gum

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

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

Nb procs = 64
import pyAgrum as gum

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

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

default_nb_threads = 10

gum.setNumberOfThreads(default_nb_threads)

# 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
```
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
setNumberOfThreads(10)

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

- **Object overriding methods:**
  - `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.
### Object overriding methods:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.getNumberThreads</code></td>
<td>returns the current number of threads used by <code>obj</code></td>
</tr>
</tbody>
</table>
Overriding the pyAgrum multithreading setting

### Object overriding methods:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj.getNumberOfThreads</td>
<td>returns the current number of threads used by obj</td>
</tr>
<tr>
<td>obj.setNumberOfThreads</td>
<td>changes the number of threads used by obj</td>
</tr>
</tbody>
</table>
### Object overriding methods:

<table>
<thead>
<tr>
<th>function</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj.getNumberOfThreads</code></td>
<td>returns the current number of threads used by <code>obj</code></td>
</tr>
<tr>
<td><code>obj.setNumberOfThreads</code></td>
<td>changes the number of threads used by <code>obj</code></td>
</tr>
<tr>
<td><code>obj.isGumNumberOfThreadsOverridden</code></td>
<td>indicates whether <code>obj</code> uses its own number or that of pyAgrum</td>
</tr>
</tbody>
</table>
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)
print("Learner threads =", learner.getNumberOfThreads(), " 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())

# 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
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)
print("Learner threads =", learner.getNumberOfThreads(),
    " 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())

# 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
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
```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

- Performing a large amount of learning experiments:
  use 1 thread per BNlearner
  perform experiments in parallel

- Performing a small amount of learnings:
  let nb processors ≈ A × B,
  with K multiple of A
  perform A experiments in parallel
  use B thread per BNlearner
**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
  → improved cacheline use
Parallelization in Inferences
New exact inference architecture (1/2)

\[ P(A|B), P(B) \]

\[ P(C|D), P(D) \]

\[ P(A|B) \times P(B) + P(D) \times P(C|D) + P(E|B, C) \times P(B) \times P(C|D) \]
New exact inference architecture (1/2)

\[
P(A|B), P(B) \quad P(C|D), P(D) \quad P(E|B, C) \quad P(F|E)
\]

\[
P(A|B) = P(B|A) \times P(A)
\]

\[
P(B) = P(A)P(B|A) + P(D)P(C|D)
\]

\[
P(C|D) = P(D|C)P(C)
\]

\[
P(E|B, C) = P(E|B, C, D)P(D)
\]

\[
P(F|E) = P(E|F)P(F)
\]
New exact inference architecture (1/2)

\[ P(A|B), P(B) \]

\[ P(C|D), P(D) \]

\[ P(A, B) = P(A|B) \times P(B) \]
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  $P(B) = \sum_A P(A, B)$
New exact inference architecture (1/2)

\[
P(A|B), P(B) \\
AB \quad B \\
CD \quad C \\
P(C|D), P(D) \\
BCE \quad E \\
P(E|B, C) \\
P(F|E) \\
P(A|B) \quad P(B) \\
\times \\
+ \\
P(A, B) = P(A|B) \times P(B) \\
P(B) = \sum_A P(A, B)
\]
New exact inference architecture (1/2)

\[
P(A|B), P(B)
\]

\[
P(C|D), P(D)
\]

\[
P(E|B, C)
\]

\[
P(F|E)
\]

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. \[P(C) = \sum_D P(C, D)\]
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. $P(C) = \sum_D P(C, D)$
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. \( P(E) = \sum_{B,C} P(B, C, E) \)
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. $P(E) = \sum_{B,C} P(B, C, E)$
\[ P(A|B), P(B) \]

\[ P(A, B) = P(A|B) \times P(B) \]

\[ P(B) = \sum_A P(A, B) \]

\[ P(C, D) = P(C|D) \times P(D) \]

\[ P(C) = \sum_D P(C, D) \]

\[ P(B, C, E) = P(E|B, C) \times P(B) \times P(C) \]

\[ P(E) = \sum_{B,C} P(B, C, E) \]

\[ P(E, F) = P(F|E) \times P(E) \]

\[ P(F) = \sum_E P(E, F) \]
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)

\[ P(A|B), P(B) \]

\[ P(C|D), P(D) \]

\[ P(E|B, C) \]

\[ P(F|E) \]

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. \[ P(E) = \sum_{B,C} P(B, C, E) \]
7. \[ P(E, F) = P(F|E) \times P(E) \]
8. \[ P(F) = \sum_E P(E, F) \]
1 Create a junction tree

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
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
Create a junction tree

Create a schedule from the JT

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
Create a junction tree

Create a schedule from the JT

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

\[
P(A|B) \times P(B) \times P(D) \times P(C|D) + P(E|B, C) \times P(F|E)
\]
Parallel scheduler – an example

\[ P(A|B) \times P(B) \times P(D) \times P(C|D) \]

\[ \times \]

\[ P(E|B, C) \]

\[ + \]

\[ P(F|E) \]
Parallel scheduler – an example
Parallel scheduler – an example
Parallel scheduler – an example

$$P(A|B) \times P(B) = P(E|B, C) = P(F|E)$$
Parallel scheduler – an example

\[ P(A|B) \times P(B) \times P(D) \times P(C|D) \]

\[ + \]

\[ P(E|B, C) \]

\[ + \]

\[ P(F|E) \]
Parallel scheduler – an example

\[ P(A|B) \times P(B) \times P(D) \times P(C|D) \]

\[ P(E|B, C) \]

\[ P(F|E) \]
LazyPropagation’s inferences on Munin4

![Graph showing the relationship between number of threads and inference time.](image-url)
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
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
Synchronization mechanisms:

- Mutexes / locks
- Condition variables
- Atomics

... but there is still an overhead.
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
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
- Performing a large amount of inference experiments: use 1 thread per LazyPropagation instance and perform experiments in parallel
- Performing a small amount $K$ of inferences: use 4 threads per LazyPropagation instance and perform $K/4$ experiments in parallel
Different situations for large BNs:

- Performing just one inference:
  parallelize with pyAgrum number of threads

- Performing a large amount of inference experiments:
  use 1 thread per LazyPropagation instance
  perform experiments in parallel
Different situations for large BNs:

- Performing just one inference:
  parallelize 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**: $T \downarrow v \mapsto T$
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

→ 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
  - 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.
  \[\Rightarrow\] requires splitting Potential operators computations.
- Add a scheduler exploiting GPU.
  \[\Rightarrow\] requires `ScheduleOperator` for changing the order of variables in potentials.
aGrUM’s multithreading facility
Multithreaded objects support both openMP and STL threads

- By default, openMP is used except if:
  - the user compiled aGrUM with the --threads=stl option
  - the compiler does not support openMP

- Parallelism achieved by using ThreadExecutor instances

- Advantages:
  - Multithreaded objects are agnostic
  - Exceptions can be caught
  - When one thread: no overhead
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
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 caught
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 \texttt{--threads=stl} option
  - or the compiler does not support openMP

- Parallelism achieved by using \texttt{ThreadExecutor} instances

- Advantages:
  - Multithreaded objects are agnostic
  - Exceptions can be caught
  - When one thread: no overhead
```
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 caught" << std::endl;
}
```
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 caught" << std::endl;
}
ThreadExecutors – an example

```cpp
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 caught" << std::endl;
}
```

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

⇒ Exceptions can be caught in Python!
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 ");
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
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
Parallelism speeds-up learning and inference computations
Parallelism speeds-up learning and inference computations

Many things to do yet for inferences

- in particular, check \texttt{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
Parallelism speeds-up learning and inference computations

Many things to do yet for inferences
  in particular, check \texttt{const} objects...

Reduce schedules' creations overhead

inferences over GPU