A static scheduling approach to enable safety-critical OpenMP applications

Alessandra Melani, Maria A. Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones, Giorgio Buttazzo

ASP-DAC 2017
January 16-19, 2017
Chiba/Tokyo, Japan
Motivation

• There is an increasing demand of new safety-critical real-time applications providing high performance
  – Timing guarantees are fundamental to be fulfilled

• Performance demands can be satisfied by using advanced parallel architectures (multi/many-core)
Parallel programming models

• Fundamental for exploiting the performance of multi- and many-cores
  – Provide the level of abstraction to express parallel applications, while hiding processor complexities
  – Mandatory to exploit the massively parallel computation capabilities

• OpenMP is one of the most used in HPC
  – Increasingly adopted in embedded systems
OpenMP

• Supported by most of current many-core architectures

• Allows expressing fine-grained and unstructured parallelism
  – Tasks
  – Dependencies

Transparent to the programmer
Time predictable OpenMP

• OpenMP tasking model
  – Task Dependency Graph (TDG)

• It resembles the Direct Acyclic Graph (DAG) real-time scheduling model
  – Addresses the time predictability of real-time parallel computation
OpenMP for safety-critical systems?

• Current OpenMP implementations rely on dynamic scheduling approaches
  – Allow schedulability analysis exploiting the work-conserving nature of scheduling [1]
  – Less suitable to safety-critical systems timing analysis

• This work provides OpenMP-compliant static allocation strategies
  – Allow a tighter timing analysis as it knows where each task executes
  – More suitable to safety-critical systems timing analysis

OpenMP tasking model

```c
#pragma omp parallel num_threads(3)
#pragma omp single    // T1
{
    P_{11}
    #pragma omp task depend(out:x) // T2
    P_{21}
    P_{12}
    #pragma omp task depend(in:x) // T3
    P_{31}
    P_{13}
    #pragma omp task               // T4
    P_{41}
    P_{14}
}
```
OpenMP tasking model

Task Scheduling Points (TSPs)

- Task may be suspended

```c
#pragma omp parallel num_threads(N)
#pragma omp single    // T1
{
    P_{11}
    #pragma omp task depend(out:x)    // T2
    P_{21}
    P_{12}
    #pragma omp task depend(in:x)    // T3
    P_{31}
    P_{13}
    #pragma omp task                // T4
    P_{41}
    P_{14}
}
```

Task-parts $p_{i,j}$

- Represented by their WCET $C_{i,j}$
OpenMP tasking model

From an OpenMP program, an OpenMP-DAG can be derived [2]

OpenMP tasking model

Task classification that affects the scheduling

• Tied tasks
  – Must only be executed by the thread that started it

• Untied tasks
  – Can be resumed by any thread after being suspended
OpenMP scheduling

• Dynamic scheduling [1]
  – Valid only for untied tasks

\[ R^{ub} = \text{len}(G) + \frac{1}{m} (\text{vol}(G) - \text{len}(G)) \leq D \]

• Our proposal: Static scheduling
  – Valid for tied and untied tasks
  – Two approaches:
    • Optimal ILP based
    • Sub-optimal Heuristics-based

Strategy 1: Optimal static allocation

• Problem definition: Optimally allocate OpenMP task-parts to threads
  – Determine the minimum time interval needed to execute an OpenMP application on m threads

• Solution
  – ILP formulation for tied tasks
  – ILP formulation for untied tasks

• Complexity
  – NP-hard
  – Number of variables and constrains: $O(N^2 p^2 m)$
Strategy 2: Sub-optimal static allocation

- Heuristics (priority rules) to solve the makespan minimization problem [3,4]:
  - Longest Processing Time (LPT)
  - Shortest Processing Time (SPT)
  - Largest Number of Successors in the Next Level (LNSNL)
  - Largest Number of Successors (LNS)
  - Largest Remaining Workload (LRW)

Strategy 2: Sub-optimal static allocation

- **Tied tasks**
  - **Input**
    - \( G \): OpenMP DAG
    - \( m \): Num. threads
  - **Output**
    - \( \mu \): Makespan
    - \( \Psi \): Task-parts starting times
    - \( \theta \): Task-to-thread mapping
  - \( A \): Allocated task-parts
  - \( R \): Ready task-parts
  - \( L[1..m] \): Last idle time of each thread
  - \( S[1..m] \): Tasks suspended on each thread

```plaintext
1: procedure HEURTIED(G, m)
2:      \( A \leftarrow \emptyset; R \leftarrow p_{1,1} \)
3:      \( L \leftarrow ARRAY(m, \emptyset); S \leftarrow ARRAY(m, \emptyset) \)
4:      \textbf{while} \( \text{SIZE}(A) \neq \sum_{i=1}^{N} n_i \) \textbf{do}
5:          \( k \leftarrow \text{FIRSTIDLETHREAD}(L) \)
6:          \( P_{i,j} \leftarrow \text{NEXTREADYJOB}(k, R, S_k, G) \)
7:          \textbf{if} \( j == 1 \) \textbf{then}
8:              \( \theta_i \leftarrow k \)
9:          \textbf{else if} \( j != n_i \) \textbf{then}
10:             \( S_k \leftarrow \text{APPEND}(i, S_k) \)
11:          \textbf{end if}
12:      \textbf{else if} \( j == n_i \) \textbf{then}
13:          \( S_k \leftarrow \text{REMOVE}(i, S_k) \)
14:      \textbf{end if}
15:          \( \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} \leftarrow \psi_{i,j} + C_{i,j} \)
16:          \( A \leftarrow \text{APPEND}(P_{i,j}, A); R \leftarrow \text{REMOVE}(P_{i,j}, R) \)
17:      \textbf{for} \( P_{k,z} \mid (P_{i,j}, P_{k,z}) \in E \) \textbf{do}
18:          \textbf{if} \( \psi_{k,z} < \psi_{i,j} + C_{i,j} \) \textbf{then}
19:              \( \psi_{k,z} \leftarrow \psi_{i,j} + C_{i,j} \)
20:          \textbf{end if}
21:              \( F_{k,z} \leftarrow F_{k,z} + 1 \)
22:          \textbf{if} \( F_{k,z} == \text{SIZE}(\text{INDEG}_{k,z}) \) \textbf{then}
23:              \( R \leftarrow \text{APPEND}(P_{k,z}, R) \)
24:          \textbf{end if}
25:      \textbf{end for}
26:  \textbf{end while}
27:  \( \mu = \max_{i=1}^{m} L_i \)
28:  \textbf{return} (\mu, \psi, \theta)
29:  \textbf{end procedure}
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

Iterates until all task-parts have been allocated

```
1: procedure HEURTIED(G, m)
2:     A ← ∅; R ← p_{1,1}
3:     L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4: while SIZE(A) != \sum_{i=1}^{N} n_i do
5:     k ← FIRSTIDLETHREAD(L)
6:     P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7: if j == 1 then
8:     \theta_i ← k
9: if j != n_i then
10:     S_k ← APPEND(i, S_k)
11: end if
12: else if j == n_i then
13:     S_k ← REMOVE(i, S_k)
14: end if
15: \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} ← \psi_{i,j} + C_{i,j}
16: A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17: for P_{k,z} | (P_{i,j}, P_{k,z}) ∈ E do
18: if \psi_{k,z} < \psi_{i,j} + C_{i,j} then
19:     \psi_{k,z} ← \psi_{i,j} + C_{i,j}
20: end if
21: F_{k,z} ← F_{k,z} + 1
22: if F_{k,z} == SIZE(INDEGES_k, z) then
23:     R ← APPEND(P_{k,z}, R)
24: end if
25: end for
26: end while
27: \mu = \max_{i=1}^{m} L_i
28: return (\mu, \psi, \theta)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

Find the earliest available thread

```
1: procedure HEURTIED(G, m)
2:    A ← ∅; R ← p_{1,1}
3:    L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4:    while SIZE(A) ! = \sum_{i=1}^{N} n_i do
5:        k ← FIRSTIDLETHREAD(L)
6:        P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:        if j == 1 then
8:            \theta_i ← k
9:        if j ! = n_i then
10:           S_k ← APPEND(i, S_k)
11:       end if
12:    else if j == n_i then
13:       S_k ← REMOVE(i, S_k)
14:    end if
15:    \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} ← \psi_{i,j} + C_{i,j}
16:    A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17:    for P_{k,z} | (P_{i,j}, P_{k,z}) \in E do
18:        if \psi_{k,z} < \psi_{i,j} + C_{i,j} then
19:           \psi_{k,z} ← \psi_{i,j} + C_{i,j}
20:    end if
21:    F_{k,z} ← F_{k,z} + 1
22:    if F_{k,z} == SIZE(INDEGS_{k,z}) then
23:        R ← APPEND(P_{k,z}, R)
24:    end if
25: end for
26: end while
27: \mu = \max_{i=1}^{m} L_i
28: return (\mu, \psi, \theta)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

Find the next ready task-part according to previous heuristics
- Checks tied tasks scheduling restrictions

```
1: procedure HEURTIED(G, m)
2:     A ← ∅; R ← p_{1,1}
3:     L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4:     while SIZE(A) ≠ \sum_{i=1}^{N} n_i do
5:         k ← FIRSTIDLETHREAD(L)
6:         P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:     if j == 1 then
8:         θ_i ← k
9:     if j != n_i then
10:        S_k ← APPEND(i, S_k)
11:     end if
12:     else if j == n_i then
13:        S_k ← REMOVE(i, S_k)
14:     end if
15:     ψ_{i,j} = \max(L_{θ_i}, ψ_{i,j}); L_{θ_i} ← ψ_{i,j} + C_{i,j}
16:     A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17:     for P_{k,z} \mid (P_{i,j}, P_{k,z}) ∈ E do
18:         if ψ_{k,z} < ψ_{i,j} + C_{i,j} then
19:             ψ_{k,z} ← ψ_{i,j} + C_{i,j}
20:         end if
21:         F_{k,z} ← F_{k,z} + 1
22:     if F_{k,z} == SIZE(INDEGS_{k,z}) then
23:         R ← APPEND(P_{k,z}, R)
24:     end if
25: end for
26: end while
27: μ = \max_{i=1}^{m} L_i
28: return (μ, ψ, θ)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

Update task-part mapping

```
1: procedure HEURTIED(G, m)
2:     A ← φ; R ← p_{1,1}
3:     L ← ARRAY(m, 0); S ← ARRAY(m, φ)
4:     while SIZE(A) ! = \sum_{i=1}^{N} n_i do
5:         k ← FIRSTIDLETHREAD(L)
6:         P_{i, j} ← NEXTREADYJOB(k, R, S_k, G)
7:     if j == 1 then
8:         \theta_i ← k
9:         if j ! = n_i then
10:             S_k ← APPEND(i, S_k)
11:     end if
12:     else if j == n_i then
13:         S_k ← REMOVE(i, S_k)
14:     end if
15:     \psi_{i, j} = \max(L_{\theta_i}, \psi_{i, j}); L_{\theta_i} ← \psi_{i, j} + C_{i, j}
16:     A ← APPEND(P_{i, j}, A); R ← REMOVE(P_{i, j}, R)
17:     for P_{k, z} \mid (P_{i, j}, P_{k, z}) \in E do
18:         if \psi_{k, z} < \psi_{i, j} + C_{i, j} then
19:             \psi_{k, z} ← \psi_{i, j} + C_{i, j};
20:         end if
21:         F_{k, z} ← F_{k, z} + 1
22:     if F_{k, z} == SIZE(INDEG_K, z) then
23:         R ← APPEND(P_{k, z}, R)
24:     end if
25:     end for
26: end while
27: \mu = \max_{i=1}^{m} L_i
28: return (\mu, \psi, \theta)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

Update
- task-part starting time
- thread next idle time

```plaintext
1: procedure HEURTIED(G, m)
2:     A ← ∅; R ← p_{1,1}
3:     L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4:     while SIZE(A) ! = \sum_{i=1}^{N} n_i do
5:         k ← FIRSTIDLETHREAD(L)
6:         P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:         if j == 1 then
8:             \theta_i ← k
9:         if j != n_i then
10:             S_k ← APPEND(i, S_k)
11:         end if
12:     else if j == n_i then
13:         S_k ← REMOVE(i, S_k)
14:     end if
15:     \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} ← \psi_{i,j} + C_{i,j}
16:     A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17:     for P_{k,z} | (P_{i,j}, P_{k,z}) ∈ E do
18:         if \psi_{k,z} < \psi_{i,j} + C_{i,j} then
19:             \psi_{k,z} ← \psi_{i,j} + C_{i,j}
20:         end if
21:         F_{k,z} ← F_{k,z} + 1
22:         if F_{k,z} == SIZE(INDEGS_{k,z}) then
23:             R ← APPEND(P_{k,z}, R)
24:         end if
25:     end for
26:     end while
27:     \mu = \max_{i=1}^{m} L_i
28:     return (\mu, \psi, \theta)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

• Tied Tasks

```plaintext
1: procedure HEURTIED(G, m)
2: A ← ∅; R ← p1,1
3: L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4: while SIZE(A) != \sum_{i=1}^{N} n_i do
5:   k ← FIRSTIDLETHREAD(L)
6:   \text{P}_{i,j} ← \text{NEXTREADYJOB}(k, R, S_k, G)
7:   if j == 1 then
8:      \theta_i ← k
9:   if j != n_i then
10:      S_k ← APPEND(i, S_k)
11:   end if
12: else if j == n_i then
13:      S_k ← REMOVE(i, S_k)
14:   end if
15:   \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} ← \psi_{i,j} + C_{i,j}
16:   A ← APPEND(\text{P}_{i,j}, A); R ← REMOVE(\text{P}_{i,j}, R)
17:   for \text{P}_{k,z} | (\text{P}_{i,j}, \text{P}_{k,z}) \in E do
18:      if \psi_{k,z} < \psi_{i,j} + C_{i,j} then
19:         \psi_{k,z} ← \psi_{i,j} + C_{i,j}
20:       end if
21:      \text{F}_{k,z} ← \text{F}_{k,z} + 1
22:   if \text{F}_{k,z} == \text{SIZE(INEDGES}_{k,z}) then
23:      R ← APPEND(\text{P}_{k,z}, R)
24:   end if
25: end for
26: end while
27: \mu = \max_{i=1}^{m} L_i
28: return (\mu, \psi, \theta)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks

```
1: procedure HEURTIRED(G, m)
2:     A ← Φ; R ← p_{1,1}
3:     L ← ARRAY(m, 0); S ← ARRAY(m, Φ)
4:     while SIZE(A) ≡ ∑_{i=1}^{N} n_i do
5:         k ← FIRSTIDLETHREAD(L)
6:         P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:         if j == 1 then
8:             θ_i ← k
9:         if j != n_i then
10:            S_k ← APPEND(i, S_k)
11:         end if
12:     else if j == n_i then
13:         S_k ← REMOVE(i, S_k)
14:     end if
15:     ψ_{i,j} = max(L_{θ_i}, ψ_{i,j}); L_{θ_i} ← ψ_{i,j} + C_{i,j}
16:     A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17:     for P_{k,z} | (P_{i,j}, P_{k,z}) ∈ E do
18:         if ψ_{k,z} < ψ_{i,j} + C_{i,j} then
19:             ψ_{k,z} ← ψ_{i,j} + C_{i,j}
20:         end if
21:     F_{k,z} ← F_{k,z} + 1
22:     if F_{k,z} == SIZE(INDEGS_{k,z}) then
23:         R ← APPEND(P_{k,z}, R)
24:     end if
25:     end for
26: end while
27: μ = max_{i=1}^{m} L_i
28: return (μ, ψ, θ)
29: end procedure
```
Strategy 2: Sub-optimal static allocation

- Tied Tasks
- Untied task
  - Slightly simpler algorithm
- Complexity: $O(N^2 p^2)$

```
1: procedure HEURTIED(G, m)
2:   A ← ∅; R ← p_1,1
3:   L ← ARRAY(m, 0); S ← ARRAY(m, ∅)
4:   while SIZE(A) ≠ \sum_{i=1}^{N} n_i do
5:     k ← FIRSTIDLETHREAD(L)
6:     P_{i,j} ← NEXTREADYJOB(k, R, S_k, G)
7:     if j == 1 then
8:       \theta_i ← k
9:     if j ≠ n_i then
10:        S_k ← APPEND(i, S_k)
11:      end if
12:     else if j == n_i then
13:        S_k ← REMOVE(i, S_k)
14:      end if
15:     \psi_{i,j} = \max(L_{\theta_i}, \psi_{i,j}); L_{\theta_i} ← \psi_{i,j} + C_{i,j}
16:     A ← APPEND(P_{i,j}, A); R ← REMOVE(P_{i,j}, R)
17:     for P_{k,z} | (P_{i,j}, P_{k,z}) ∈ E do
18:       if \psi_{k,z} < \psi_{i,j} + C_{i,j} then
19:         \psi_{k,z} ← \psi_{i,j} + C_{i,j}
20:     end if
21:     F_{k,z} ← F_{k,z} + 1
22:     if F_{k,z} = SIZE(INEDGES_{k,z}) then
23:       R ← APPEND(P_{k,z}, R)
24:     end if
25:   end for
26: end while
27: \mu = \max_{i=1}^{m} L_i
28: return (\mu, \psi, \theta)
29: end procedure
```
Evaluation: Experimental setting

• Static allocation strategies vs. Response-Time upper bound [1]

• Task sets
  – Real OpenMP 3D path planning application
  – Synthetic DAG task-sets

• Intel® Core™ i7-4770K CPU 3.50 GHz
  – 16GB RAM
  – ILP solver: IBM ILOG CPLEX Optimization Studio v.12.61

Evaluation: 3D Path Planning application

• Real case study: Airborne collision avoidance

• Application set ups: 3DPP1 and 3DPP2
  – DAGs composed of 129 and 257 nodes, respectively
Evaluation: 3D Path Planning application

- Real case study: Airborne collision avoidance
- Application set ups: 3DPP1 and 3DPP2
  - DAGs composed of 129 and 257 nodes, respectively

<table>
<thead>
<tr>
<th></th>
<th>3DPP1 (m=8)</th>
<th>3DPP2 (m=2)</th>
<th>3DPP2 (m=4)</th>
<th>3DPP2 (m=8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>254</td>
<td>506</td>
<td>506</td>
<td>506</td>
</tr>
</tbody>
</table>

ILP: Converged in ~10 sec. to the best found solution
Evaluation: 3D Path Planning application

- Real case study: Airborne collision avoidance
- Application set ups: 3DPP1 and 3DPP2
  - DAGs composed of 129 and 257 nodes, respectively

<table>
<thead>
<tr>
<th></th>
<th>3DPP1 (m=8)</th>
<th>3DPP2 (m=2)</th>
<th>3DPP2 (m=4)</th>
<th>3DPP2 (m=8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>254</td>
<td>506</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>SPT</td>
<td>317</td>
<td>824</td>
<td>660</td>
<td>571</td>
</tr>
<tr>
<td>LPT</td>
<td>254</td>
<td>659</td>
<td>577</td>
<td>530</td>
</tr>
<tr>
<td>LNS</td>
<td>254</td>
<td>715</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>LNSNL</td>
<td>300</td>
<td>748</td>
<td>619</td>
<td>549</td>
</tr>
<tr>
<td>LRW</td>
<td>254</td>
<td>717</td>
<td>506</td>
<td>506</td>
</tr>
</tbody>
</table>

Sub-optimal heuristics
Evaluation: 3D Path Planning application

- Real case study: Airborne collision avoidance

- Application set ups: 3DPP1 and 3DPP2
  - DAGs composed of 129 and 257 nodes, respectively

<table>
<thead>
<tr>
<th></th>
<th>3DPP1 (m=8)</th>
<th>3DPP2 (m=2)</th>
<th>3DPP2 (m=4)</th>
<th>3DPP2 (m=8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>254</td>
<td>506</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>SPT</td>
<td>317</td>
<td>824</td>
<td>660</td>
<td>571</td>
</tr>
<tr>
<td>LPT</td>
<td>254</td>
<td>659</td>
<td>577</td>
<td>530</td>
</tr>
<tr>
<td>LNS</td>
<td>254</td>
<td>715</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>LNSNL</td>
<td>300</td>
<td>748</td>
<td>619</td>
<td>549</td>
</tr>
<tr>
<td>LRW</td>
<td>254</td>
<td>717</td>
<td>506</td>
<td>506</td>
</tr>
</tbody>
</table>

Sub-optimal heuristics

(7s) (11m41s) (11m48s) (11m53s)
Evaluation: 3D Path Planning application

- Real case study: Airborne collision avoidance
- Application set ups: 3DPP1 and 3DPP2
  - DAGs composed of 129 and 257 nodes, respectively

<table>
<thead>
<tr>
<th></th>
<th>3DPP1 (m=8)</th>
<th>3DPP2 (m=2)</th>
<th>3DPP2 (m=4)</th>
<th>3DPP2 (m=8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ILP</td>
<td>254</td>
<td>506</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>SPT</td>
<td>317</td>
<td>824</td>
<td>660</td>
<td>571</td>
</tr>
<tr>
<td>LPT</td>
<td>254</td>
<td>659</td>
<td>577</td>
<td>530</td>
</tr>
<tr>
<td>LNS</td>
<td>254</td>
<td>715</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>LNSNL</td>
<td>300</td>
<td>748</td>
<td>619</td>
<td>549</td>
</tr>
<tr>
<td>LRW</td>
<td>254</td>
<td>717</td>
<td>506</td>
<td>506</td>
</tr>
<tr>
<td>BOUND-untied</td>
<td>331</td>
<td>827</td>
<td>666.5</td>
<td>586.25</td>
</tr>
</tbody>
</table>

Dynamic approach: Max. over estimation: 63%
Evaluation: Synthetic OpenMP-DAGs

• **Small task sets, 4 cores**

  Max. over estimation
dynamic vs. static ILP: ~40%

Larger solution space for the untied model: 50% slower
Evaluation: Synthetic OpenMP-DAGs

- Large task sets, 4 cores
  Best feasible solution by ILP solver in 300 s

OpenMP Tied Model

OpenMP Untied Model

LNSNL outperforms ILP for untied
Conclusions

• Parallel programing models are fundamental to exploit the performance capabilities of parallel architectures
  – OpenMP, one of the most advanced

• However, relies on dynamic scheduling, not suitable in certain safety-critical domains

• We propose two OpenMP-complain static allocation strategies:
  – A computationally expensive but optimal ILP solver
  – More efficient but sub-optimal heuristics
A static scheduling approach to enable safety-critical OpenMP applications

Alessandra Melani, Maria A. Serrano, Marko Bertogna, Isabella Cerutti, Eduardo Quiñones, Giorgio Buttazzo

This work was supported by the EU projects P-SOCRATES (FP7-ICT-2013-10) and HERCULES (H2020/ICT/2015/688860) and the Spanish Ministry of Science and Innovation grant TIN2015-65316-P