Exploring Single Source Shortest Path Parallelization on Shared Memory Accelerators

Daniele Palossi†, Andrea Marongiu†

†IIS - ETH Zürich

http://hipert.unimore.it/hercules
Introduction: Single Source Shortest Path

**SSSP problem**: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$. 
Introduction: Single Source Shortest Path

SSSP problem: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

Embedded domain use cases:
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

Introduction: Single Source Shortest Path

**SSSP problem**: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

**Embedded domain use cases:**
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

---

Introduction: Single Source Shortest Path

**SSSP problem:** given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

**Embedded domain use cases:**
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

SSSP problem: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

Embedded domain use cases:
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

Introduction: Single Source Shortest Path

SSSP problem: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

Embedded domain use cases:
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

Introduction: Single Source Shortest Path

SSSP problem: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

Embedded domain use cases:
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

Introduction: Single Source Shortest Path

**SSSP problem**: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

Embedded domain use cases:

- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

---

Introduction: Single Source Shortest Path

**SSSP problem:** given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

**Embedded domain use cases:**
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]


---

Introduction: Single Source Shortest Path

**SSSP problem**: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

**Embedded domain use cases:**
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]

---

**Introduction: Single Source Shortest Path**

**SSSP problem**: given a weighted graph $G = (V, E, c)$ find a minimal weight path from one chosen node $s \in V$ to all other nodes in $V$.

**Embedded domain use cases:**
- IoT Local Area Network routing [1]
- Body Area Network least total-route-temperature in implanted sensors [2]
- Micro-size UAV path planner for autonomous navigation [3]
Energy Efficiency

<table>
<thead>
<tr>
<th></th>
<th>Current system</th>
<th>Next Gen system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size [Ø, weight]</td>
<td>50 cm / few Kg</td>
<td>few cm / few g</td>
</tr>
<tr>
<td>Propellers Power Cons.</td>
<td>hundreds of W</td>
<td>few W / hundred mW</td>
</tr>
<tr>
<td>Processing Device Class</td>
<td>desktop CPU</td>
<td>LP/ULP embedded</td>
</tr>
<tr>
<td>Cognitive Skills</td>
<td>fully autonomous</td>
<td></td>
</tr>
</tbody>
</table>

If we want bring advanced cognitive skills of state-of-the-art systems into next-generation nano-size UAV with limited power envelope → **parallel algorithms are key**

Our target: Heterogeneous Systems $\rightarrow$ Host + Accelerator

Ubiquitous paradigm, shared:

- from HPC environment
- to embedded systems
- lastly in ultra-low-power embedded domain
Our target: Heterogeneous Systems → Host + Accelerator
Target Architectural Template

Our target: Heterogeneous Systems → Host + Accelerator
Our target: Heterogeneous Systems → Host + Accelerator [5]

Class of Graph: Game-Map

Topological representations used in:
- Robot planning trajectories [6]
- Grid-based Gaming [7]

In our work cost edges represent the euclidean distance between locations:

Other possibilities (not explored here), with different properties, are: small-world graph, etc.

Δ-Stepping is an extension of Dijkstra’s algorithm to achieve parallel efficiency. It is carried out by dividing the problem-space into boundaries, where the computation is performed. Edges with a cost lower than a given threshold (Δ) are explored first, in each phase.

Primary structures:
- **B**: the Buckets keep track of the exploration boundaries
- **Req**: the Request set of nodes to be explored in the current phase
- **S**: auxiliary Set structure to keep temporary copies of Req
- **Tent**: Tentative cost proposed for each node in the graph

---

Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into *light* and *heavy* sets $\rightarrow$ can be avoided due to our constant costs
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[
\begin{align*}
B_0 & \rightarrow B_1 & \cdots \\
1 & \quad \quad & \\
\end{align*}
\]
**Δ-Stepping Algorithm: example**

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into *light* and *heavy* sets → can be avoided due to our constant costs
2. while *Bucket* is not empty perform the exploration

![Diagram](image)

3. for each element in the current bucket, build a *Request* set of nodes reachable with *light* edges $[id, cost]$

```
Req 2,10 5,10
```
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, Δ = 14

Stages:
1. divide edges into *light* and *heavy* sets → can be avoided due to our constant costs
2. while *Bucket* is not empty perform the exploration

\[ B_0 \rightarrow B_1 \rightarrow \ldots \]

3. for each element in the current bucket, build a *Request* set of nodes reachable with light edges \([id, cost]\)

\[
\text{Req} \begin{bmatrix} 2,10 & 5,10 \end{bmatrix}
\]

4. make a copy of current *Bucket* in *S* \((S \leftarrow B_i)\) and reset it, then for each element in *Req* call the *relax* function

\[
\text{tent} \begin{bmatrix} 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}
\]

\[
\begin{bmatrix} 1 & \infty & \infty & 4 & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}
\]
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into *light* and *heavy* sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[ B_0 \rightarrow B_1 \rightarrow \ldots \]

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[i, d, c]$

\[ \text{Req} \begin{bmatrix} 2, & 10 & 5, & 10 \end{bmatrix} \]

4. make a copy of current Bucket in $S \leftarrow B_i$ and reset it, then for each element in Req call the relax function

\[ \text{S} \begin{bmatrix} 1 \end{bmatrix} \]

5. if proposed cost < previous one → insert node into the Bucket, store the new cost

\[ \text{tent} \begin{bmatrix} 0 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix} \]
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[
\begin{array}{c}
B_0 \rightarrow B_1 \rightarrow \cdots \\
\downarrow \\
2 \rightarrow 5
\end{array}
\]

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[\text{id}, \text{cost}]$

\[
\text{Req} \quad 2, 10 \quad 5, 10
\]

4. make a copy of current Bucket in $S$ ($S \leftarrow B_1$) and reset it, then for each element in Req call the relax function

\[
\text{S} \quad 1
\]

5. if proposed cost < previous one → insert node into the Bucket, store the new cost

\[
\begin{array}{c}
\text{tent} \\
0 \quad \infty \quad \infty \quad \infty \quad \infty \quad \infty \quad \infty \quad \infty \quad \infty \quad \infty
\end{array}
\]

\[
\begin{array}{c}
1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10
\end{array}
\]
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[ B_0 \rightarrow B_1 \rightarrow \ldots \]

\[ 2 \rightarrow 5 \]

3. for each element in the current bucket, build a Request set of nodes reachable with light edges [id, cost]

\[ \text{Req} \begin{array}{c} 2,10 \end{array} \begin{array}{c} 5,10 \end{array} \]

4. make a copy of current Bucket in $S$ ($S \leftarrow B_i$) and reset it, then for each element in Req call the relax function

\[ S \begin{array}{c} 1 \end{array} \]

5. if proposed cost < previous one → insert node into the Bucket, store the new cost

\[ \text{tent} \begin{array}{c} 0 \end{array} \begin{array}{c} 10 \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} 10 \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \]

\[ 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \]

\[ 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \]

\[ 9 \rightarrow 10 \rightarrow 11 \rightarrow 12 \]

\[ 13 \rightarrow 14 \rightarrow 15 \rightarrow 16 \]
Δ-Stepping Algorithm: example

**Configuration:** perpendicular edges cost 10, diagonal 15, $\Delta = 14$

**Stages:**
1. divide edges into *light* and *heavy* sets → can be avoided due to our constant costs
2. while *Bucket* is not empty perform the exploration
   \[ B_0 \rightarrow B_1 \rightarrow \ldots \]  
   \[ 2 \rightarrow 5 \]
3. for each element in the current bucket, build a *Request* set of nodes reachable with light edges $[id, cost]$
   \[ \text{Req} \begin{array}{c} 2, 10 \end{array} \begin{array}{c} 5, 10 \end{array} \]
4. make a copy of current *Bucket* in $S$ ($S \leftarrow B_i$) and reset it, then for each element in *Req* call the *relax* function

\[ S \begin{array}{c} 1 \end{array} \]

5. if proposed cost < previous one → insert node into the *Bucket*, store the new cost

\[ \text{tent} \begin{array}{c} 0 \end{array} \begin{array}{c} 10 \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} 10 \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \begin{array}{c} \infty \end{array} \]

\[ \begin{array}{c} 1 \end{array} \begin{array}{c} 2 \end{array} \begin{array}{c} 3 \end{array} \begin{array}{c} 4 \end{array} \begin{array}{c} 5 \end{array} \begin{array}{c} 6 \end{array} \begin{array}{c} 7 \end{array} \begin{array}{c} 8 \end{array} \begin{array}{c} 9 \end{array} \begin{array}{c} 10 \end{array} \]

\[ \begin{array}{c} 1 \end{array} \begin{array}{c} 2 \end{array} \begin{array}{c} 3 \end{array} \begin{array}{c} 4 \end{array} \begin{array}{c} 5 \end{array} \begin{array}{c} 6 \end{array} \begin{array}{c} 7 \end{array} \begin{array}{c} 8 \end{array} \begin{array}{c} 9 \end{array} \begin{array}{c} 10 \end{array} \]

\[ \text{1 function relax}(w, c): \]
\[ 2 \quad \text{if } d<\text{tent}[w] \]
\[ 3 \quad B[[\text{tent}[w]/\Delta]] - B[[\text{tent}[w]/\Delta]] \setminus \{w\} \]
\[ 4 \quad B[[d/\Delta]] - B[[d/\Delta]] U \{w\} \]
\[ 5 \quad \text{tent}[w] = d \]
\[ 6 \quad \text{end if} \]
\[ 7 \quad \text{end function} \]
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, Δ = 14

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[ \begin{array}{c}
\text{B}_0 & \rightarrow & \text{B}_1 & \ldots \\
2 & \rightarrow & 5 
\end{array} \]

3. for each element in the current bucket, build a Request set of nodes reachable with light edges \([i_\text{d}, c_\text{ost}]\)

\[ \text{Req} \begin{array}{cccc}
1,20 & 3,20 & 6,20 & 9,20 
\end{array} \]

4. make a copy of current Bucket in \(S \leftarrow \text{B}_i\) and reset it, then for each element in Req call the relax function

\[ \text{S} \begin{array}{c}
1 
\end{array} \]

5. if proposed cost < previous one → insert node into the Bucket, store the new cost

\[ \begin{array}{cccccccccccc}
\text{tent} & 0 & 10 & \infty & \infty & 10 & \infty & \infty & \infty & \infty & \infty & \infty \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 
\end{array} \]
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[i, d, c]$

4. make a copy of current Bucket in $S$ ($S \leftarrow B_i$) and reset it, then for each element in Req call the relax function

5. if proposed cost < previous one → insert node into the Bucket, store the new cost
**Δ-Stepping Algorithm: example**

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

![Diagram of Δ-Stepping Algorithm](image)

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[i,d,cost]$

![Request set diagram](image)

4. make a copy of current Bucket in $S$ ($S \leftarrow B_i$) and reset it, then for each element in Req call the relax function

![Relax function](image)

5. if proposed cost < previous one → insert node into the Bucket, store the new cost
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

\[ B_0 \rightarrow B_1 \rightarrow \ldots \]

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[id, cost]$

\[ \text{Req} \begin{pmatrix} 1, 20 & 3, 20 & 6, 20 & 9, 20 \end{pmatrix} \]

4. make a copy of current Bucket in $S$ $(S \leftarrow B_1)$ and reset it, then for each element in Req call the relax function

5. if proposed cost < previous one → insert node into the Bucket, store the new cost
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:
1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration

3. for each element in the current bucket, build a Request set of nodes reachable with light edges $[id, cost]$

4. make a copy of current Bucket in $S$ ($S \leftarrow B_i$) and reset it, then for each element in Req call the relax function

5. if proposed cost < previous one → insert node into the Bucket, store the new cost
Δ-Stepping Algorithm: example

Configuration: perpendicular edges cost 10, diagonal 15, $\Delta = 14$

Stages:

1. divide edges into light and heavy sets → can be avoided due to our constant costs
2. while Bucket is not empty perform the exploration
3. for each element in the current bucket, build a Request set of nodes reachable with light edges
4. make a copy of current Bucket in S ($S \leftarrow B_i$) and reset it, then for each element in Req call the relax function
5. if proposed cost < previous one → insert node into the Bucket, store the new cost

Then we do the same with S and heavy edges and we are sure that new reinsertion into the Bucket will affect only following buckets.

Lastly a new phase can start if Bucket is not empty.

Req $6, 15, 7, 25, 10, 25$

S $\begin{array}{ccc}1 & 2 & 5\end{array}$
Δ-Stepping Algorithm: where the parallelism is

The exploration is split into **phases** divided by **boundaries** → wavefront parallelism
We can perform parallel exploration within the same phase (yellow area)
Δ-Stepping Algorithm: where the parallelism is

The exploration is split into **phases** divided by **boundaries** → wavefront parallelism.

We can perform parallel exploration within the same phase (yellow area).
Δ-Stepping Algorithm: where the parallelism is

The exploration is split into **phases** divided by **boundaries** → wavefront parallelism
We can perform parallel exploration within the same phase (**yellow area**)
Δ-Stepping Algorithm: where the parallelism is

The exploration is split into *phases* divided by *boundaries* → wavefront parallelism
We can perform parallel exploration within the same phase (yellow area)
In order to prevent race conditions we evaluate:

- coarse-grain synchronization mechanism
- fine-grain synchronization mechanism
- hardware-assisted operation (*compare-and-swap: CAS*)

Our Δ-Stepping implementation is slightly different from the reference implementation:

- avoiding *pre-processing* stage due to the regular graph structure
- *buckets represented* by mono-dimensional static array $|V|$
- *data-packing* of cost and predecessor id
**Δ-Stepping implementation: Bucket structure**

**Buckets** represented by mono-dimensional static array, instead of dynamic double list.

**Pro:**
- lightweight synchronization mechanism
- avoiding costly pointer-based data structures
- more parallelism exposed and easier parallelization techniques required

**Cons:**
- every phase we have to check the entire array (i.e. every node in the graph)
- potential load balancing issues (several techniques evaluated)
In order to exploit fine-grain synchronization mechanism (i.e. CAS), we package two pieces of information into one 64 bit integer, using a cost array (tent) that contains both the cost and the predecessor id, thus only the left-most piece affects the comparisons.

**Δ-Stepping implementation : data-packing**

![Diagram showing data-packing for Δ-stepping](diagram)

**Old value for Vi** (first explored)
- **id**: 1
- **cost**: 2

**Vi**

**New value for Vi** (second explored)
- **id**: 4294967295
- **cost**: 1

**Cost Vj : 2 > Cost Vk : 1**

**Old tent[Vi] (from Vj)**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>⋯</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

**New tent[Vi] (from Vk)**

| 0 | 0 | 0 | 0 | ⋯ | 0 | 0 | 0 | 1 |

**MSB**: Cost associated with the vertex
**LSB**: Predecessor vertex ID

**tent form Vj : 8589934593 > tent from Vk : 8589934591**
**TI® KeyStone II:**
- Embedded Low-Power SoC
- Host: ARM-A15 (4 cores)
- Accelerator: DSP C66x VLIW

<table>
<thead>
<tr>
<th></th>
<th>ARM A15</th>
<th>DSP: C66x</th>
</tr>
</thead>
<tbody>
<tr>
<td># Cores</td>
<td>4</td>
<td>8 (VLIW)</td>
</tr>
<tr>
<td>Core Frequency</td>
<td>1.4 GHz</td>
<td>1.2 GHz</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4096 KB (cluster)</td>
<td>1024 KB (core)</td>
</tr>
<tr>
<td>System DDR Memory</td>
<td>1 GB</td>
<td></td>
</tr>
</tbody>
</table>

**OpenMP**

**Diagram:**
- Memory Subsystem
- 6 MB MSMC SRAM
- 1 GB DDR3
- DDR3 EMIF
- 2x HyperLink
- TeraNet
- ARM Host Cluster
- DSP Accelerator
- ARM A15
- DSP C66x
- 4 MB L2 cache
- 1024 KB L2 cache
- 8x
Results outline

Host: Cortex A15

We want to evaluate:

1. parallelization **scalability**
   - OpenMP static scheduling
   - OpenMP dynamic scheduling

2. Speedups: 1 core vs. 4 cores

3. Scalability of parallelization **chunking**

Accelerator: DSP C66x

We want to evaluate:

1. parallelization **scalability** w.r.t. several synchronization mechanisms:
   - OpenMP critical section
   - OpenMP fine-grain locks
   - “CAS-equivalent”

Performance evaluation in presence of **obstacles** on the graph for both architectures
Results: Host

Evaluation of \#pragma omp for schedule(static)

Running on 4-cores ARM Cortex A15 (hw CAS)
1. hyper-threading does not help
2. bigger problem size amortize overhead
Results: Host

Evaluation of \#pragma omp for schedule(static)

Running on 4-cores ARM Cortex A15 (hw CAS)
1. hyper-threading does not help
2. bigger problem size amortize overhead
3. bucket overhead limit @ 1M nodes after than is dominating (i.e. 1500x1500)
4. load imbalance
Evaluation of \#pragma omp for schedule(dynamic, C)
Problem size: 10k, 1M - Chunk size: 1k, 8k, 32k

Running on 4-cores ARM Cortex A15 (hw CAS)
1. Small problem → chunk 1k best trade-off load balancing/overhead
Results: Host

Evaluation of `#pragma omp for schedule(dynamic, C)`

Problem size: 10k, 1M - Chunk size: 1k, 8k, 32k

Running on 4-cores ARM Cortex A15 (hw CAS)

1. Small problem → chunk 1k best trade-off load balancing/overhead
2. Big problem → chunk 8k best trade-off load balancing/overhead
Evaluation of `#pragma omp for schedule(dynamic, C)`

Problem size: 10k, 1M – Chunk size: 1k, 8k, 32k

Running on 4-cores ARM Cortex A15 (hw CAS)

1. Small problem → chunk 1k best trade-off load balancing/overhead
2. Big problem → chunk 8k best trade-off load balancing/overhead
3. Below these values overheads are predominant
4. Above these values imbalanced execution (too coarse-grain)
Evaluation of synchronization primitives

```c
#pragma omp for schedule(dynamic, 8k)
#pragma omp critical
omp_set_lock(*lock)
```

"CAS-equivalent"

A, B, C: baseline sequential (no race → no synch). D, E, F: baseline parallel 1 thread.
Results: Accelerator

Evaluation of synchronization primitives

```c
#pragma omp for schedule(dynamic, 8k)
#pragma omp critical
omp_set_lock(*lock)
```

“CAS-equivalent”

A, B, C: baseline sequential (no race → no synch). D, E, F: baseline parallel 1 thread.

OpenMP overhead is dominant with small graphs (e.g. 100x100)
Results: Accelerator

Evaluation of synchronization primitives

#pragma omp for schedule(dynamic, 8k)
#pragma omp critical
omp_set_lock(*lock)
“CAS-equivalent”

A,B,C: baseline sequential (no race → no synch). D,E,F: baseline parallel 1 thread.
Software-managed synchronization significantly limits performance
Several configurations:
• 0-50% of obstacles
• Host: 4 threads
• Accelerator: 8 threads
• OpenMP dynamic scheduling
• Chunk size: 8k

More obstacles → less nodes
Results: Obstacles evaluation

Several configurations:
- 0-50% of obstacles
- Host: 4 threads
- Accelerator: 8 threads
- OpenMP dynamic scheduling
- Chunk size: 8k

Smaller number $\rightarrow$ higher useful parallel workload i.e. some of the non-necessary checks are eliminated

More obstacles $\rightarrow$ less nodes
Results: Obstacles evaluation

Several configurations:
- 0-50% of obstacles
- Host: 4 threads
- Accelerator: 8 threads
- OpenMP dynamic scheduling
- Chunk size: 8k

More obstacles → less nodes

Locks: predominant effect is the reduction of the available parallelization due to the serialization
The exploration of Δ-Stepping algorithm on heterogeneous embedded system (architectural template Host+Accelerator) highlights:

> goodness of fine-grain synchronization mechanism (hw support)

> effectiveness of dynamically scheduling loop (medium granularity chunks)

> limitation on the maximum parallelism achievable with \textit{game-map} class of graph

> needs of algorithmic enhancements to reduce dynamically the area of the graph evaluated during bucket checking

> overall performance strongly depends on the class of graph and its properties (e.g. density)
Thanks for the attention.
Questions?
References