Safe Parallelism

Compiler Analysis Techniques for Ada and OpenMP

Sara Royuela, Xavier Martorell, Eduardo Quiñones, Luis Miguel Pinho
Challenges in safety-critical systems

- **Need for performance**

  Current safety-critical real-time systems require computational power beyond simple single-core architectures.
Challenges in safety-critical systems

– Need for performance

Current safety-critical real-time systems require computational power beyond simple single-core architectures.

– Complexities of Parallel heterogeneous architectures

- **Parallelism**
  - Race conditions
  - Deadlocks and livelocks
  - Starvation

- **Heterogeneity**
  - Different ISAs
  - Different memory views
  - Different microarchitectures
How to cope with such complexity?

Parallel programming models
How to cope with such complexity?

Parallel programming models

OpenMP
How to cope with such complexity?

Parallel programming models

20 years of development gather the benefits of other languages

- Delivers **performance** comparable with Intel TBB, CUDA, OpenCL and MPI
- Offers **robustness** without sacrificing performance compared to Pthreads
- Eases **debugging** by enabling trivial single-threaded compilation
How to cope with such complexity?

Parallel programming models

20 years of development gather the benefits of other languages

- Delivers performance comparable with Intel TBB, CUDA, OpenCL and MPI
- Offers robustness without sacrificing performance compared to Pthreads
- Eases debugging by enabling trivial single-threaded compilation

The latest specification meets the characteristics of heterogeneous architectures

- Accelerator model for improved performance/power consumption
- Allows expressing fine grain, both structured and unstructured, parallelism
- Implemented by several chip (TI Keystone, Kalray MPPA) and compiler vendors (GNU, Intel, IBM)
What is OpenMP and how far is it from the safety-critical domain?
Introduction to OpenMP

- Forms of parallelism:
  - **Thread model**: direct management of threads (structured)
  - **Tasking model**: tasks as an abstraction of threads (structured and unstructured)
Introduction to OpenMP

- Forms of parallelism:
  - **Thread model**: direct management of threads (structured)
  - **Tasking model**: tasks as an abstraction of threads (structured and unstructured)

- Based on user directives and clauses for:
  - **Spawning parallelism**: parallel
  - **Distributing parallelism**: task, taskloop
  - **Synchronization**: barrier, taskwait, depend
  - **Driving execution**: untied, priority, taskyield, ...
Introduction to OpenMP

- Forms of parallelism:
  - **Thread model**: direct management of threads (structured)
  - **Tasking model**: tasks as an abstraction of threads (structured and unstructured)

- Based on user directives and clauses for:
  - **Spawning parallelism**: parallel
  - **Distributing parallelism**: task, taskloop
  - **Synchronization**: barrier, taskwait, depend
  - **Driving execution**: untied, priority, taskyield, ...

```c
void matmul(int N, float A[N][N], float B[N][N], float C[N][N])
{
    #pragma omp parallel num_threads(4)
    #pragma omp master
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            for (int k=0; k<N; k++)
                #pragma omp task depend(in: A[i][k]) depend(in: B[k][j])
                    depend(inout: C[i][j])
                C[i][j] += A[i][k] * B[k][j];
}
```
Introduction to OpenMP

- **Forms of parallelism:**
  - **Thread model:** direct management of threads (structured)
  - **Tasking model:** tasks as an abstraction of threads (structured and unstructured)

- Based on user directives and clauses for:
  - **Spawning parallelism:** parallel
  - **Distributing parallelism:** task, taskloop
  - **Synchronization:** barrier, taskwait, depend
  - **Driving execution:** untied, priority, taskyield, ...

```c
void matmul(int N, float A[N][N], float B[N][N], float C[N][N])
{
    #pragma omp parallel num_threads(4)
    #pragma omp master
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
            for (int k=0; k<N; k++)
                #pragma omp task depend(in:A[i][k]) depend(in: B[k][j])
                    depend(inout:C[i][j])
                C[i][j] += A[i][k] * B[k][j];
}
```
Safety-critical OpenMP: where is the problem?

OpenMP 4.5 (API, page 1)

- OpenMP-compliant implementations are not required to check
  - for data dependencies, data conflicts, race conditions, or deadlocks, (…)
  - for code sequences that cause a program to be classified as non-conforming
- Application developers are responsible for correctly using the OpenMP API to produce a conforming program
Safety-critical OpenMP: requirements

OpenMP 4.5 (API, page 1)

- OpenMP-compliant implementations are not required to check
  - for data dependencies, data conflicts, race conditions, or deadlocks, (...)
  - for code sequences that cause a program to be classified as non-conforming
- Application developers are responsible for correctly using the OpenMP API to produce a conforming program

<table>
<thead>
<tr>
<th>Functional Safety</th>
<th>Time Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>What to achieve</td>
<td></td>
</tr>
<tr>
<td>Reliability: do as expected</td>
<td>Predictability: analyzable</td>
</tr>
<tr>
<td>Resiliency: recover from errors</td>
<td>Feasibility: fulfill deadlines</td>
</tr>
<tr>
<td>How to achieve it</td>
<td></td>
</tr>
<tr>
<td>Programming model restrictions</td>
<td>WCET analysis</td>
</tr>
<tr>
<td>Compiler analysis</td>
<td>Schedulability analysis</td>
</tr>
<tr>
<td>Runtime mechanisms</td>
<td></td>
</tr>
</tbody>
</table>
Safety-critical OpenMP: requirements

**OpenMP 4.5 (API, page 1)**

- OpenMP-compliant *implementations are not required* to check
  - for data dependencies, data conflicts, race conditions, or deadlocks, (...)
  - for code sequences that cause a program to be classified as non-conforming
- Application *developers are responsible* for correctly using the OpenMP API to produce a conforming program

<table>
<thead>
<tr>
<th>Functional Safety</th>
<th>Time Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What to achieve</strong></td>
<td></td>
</tr>
<tr>
<td>Reliability: do as expected</td>
<td>Predictability: analyzable</td>
</tr>
<tr>
<td>Resiliency: recover from errors</td>
<td>Feasibility: fulfill deadlines</td>
</tr>
<tr>
<td><strong>How to achieve it</strong></td>
<td></td>
</tr>
<tr>
<td>Programming model restrictions</td>
<td>WCET analysis</td>
</tr>
<tr>
<td>Compiler analysis</td>
<td>Schedulability analysis</td>
</tr>
<tr>
<td>Runtime mechanisms</td>
<td></td>
</tr>
</tbody>
</table>
Solutions for a safety-critical OpenMP

- Force implementations to detect:
  - race conditions
  - deadlocks
  - non-conforming sequences

Compiler

Related work
Solutions for a safety-critical OpenMP

- Force implementations to detect:
  - race conditions
  - deadlocks
  - non-conforming sequences

- Avoid unexpected termination defining default values for unexpected argument passing
- Allow serialization when parallelism is not well defined (dependence clauses)

Compiler

Runtime

Related work
Solutions for a safety-critical OpenMP

- Force implementations to detect:
  - race conditions
  - deadlocks
  - non-conforming sequences

- Avoid unexpected termination defining default values for unexpected argument passing
- Allow serialization when parallelism is not well defined (dependence clauses)

- Use directives to always allow whole program analysis.
- Forbid uncheckable features (flushes with arguments, priorities, etc.)
- Introduce error handling mechanisms

Related work
Parallelism in Ada202X
Ada: concurrency and parallelism now

- Ada **concurrent model** integrated at base language level
  - Tasking facilities for exposing concurrency at **coarse grain**
  - Synchronization mechanisms: **protected objects**, rendezvous
Ada: concurrency and parallelism now

- Ada **concurrent model** integrated at base language level
  - Tasking facilities for exposing concurrency at **coarse grain**
  - Synchronization mechanisms: protected objects, rendezvous

- Ada **parallel model** to be included in Ada202X
  - **Tasklets** for exposing parallelism at fine grain
  - Support for **structured parallelism**

<table>
<thead>
<tr>
<th>Parallel blocks</th>
<th>Parallel loops</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parallel do</code></td>
<td><code>parallel</code></td>
</tr>
<tr>
<td><code>handled_sequence_of_statements</code></td>
<td><code>for I in LB..UB loop</code></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>sequence_of_statements</code></td>
</tr>
<tr>
<td><code>handled_sequence_of_statements</code></td>
<td><code>end loop;</code></td>
</tr>
<tr>
<td><code>{and</code></td>
<td></td>
</tr>
<tr>
<td><code>handled_sequence_of_statements</code></td>
<td></td>
</tr>
<tr>
<td><code>end do;</code></td>
<td></td>
</tr>
</tbody>
</table>

- Does not allow **blocking operations** within parallel regions
- Under implementation (e.g., AdaCore)
OpenMP to implement the tasklet model

- OpenMP mimics the tasklet model behavior at all levels:
  - *Forms of parallelism*: parallel blocks and parallel loops
  - *Execution model*: run-to-completion
  - *Memory model*: relaxed consistency memory model
  - *Progression model*: immediate, eventual and limited

Related work

OpenMP to implement the tasklet model

- OpenMP mimics the tasklet model behavior at all levels:
  - *Forms of parallelism*: parallel blocks and parallel loops
  - *Execution model*: run-to-completion
  - *Memory model*: relaxed consistency memory model
  - *Progression model*: immediate, eventual and limited

- OpenMP offers more flexibility

<table>
<thead>
<tr>
<th>Parallel blocks</th>
<th>Parallel loops</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tasklet</strong></td>
<td><strong>OpenMP</strong></td>
</tr>
<tr>
<td><img src="image" alt="Tasklet Diagram" /></td>
<td><img src="image" alt="OpenMP Diagram" /></td>
</tr>
<tr>
<td><strong>Parallel loops</strong></td>
<td><strong>OpenMP</strong></td>
</tr>
<tr>
<td><img src="image" alt="Parallel Loop 1 Diagram" /></td>
<td><img src="image" alt="Parallel Loop 2 Diagram" /></td>
</tr>
</tbody>
</table>

OpenMP to further exploit parallelism in Ada

Matrix (coarse grain synchronization)

Perfect speed-up
OpenMP to further exploit parallelism in Ada

**Matrix** (coarse grain synchronization)

Perfect speed-up

**LU** (fine grain synchronization)

Great speed-up enhancement

Memory boundaries NUMA effect

Related work
OpenMP to further exploit parallelism in Ada

**Matrix** (coarse grain synchronization)

- Ada tasklets (OpenMP)
- Ada tasks
- Ada Paraffin

**LU** (fine grain synchronization)

- Ada tasklets (OpenMP)
- Ada tasks
- Ada Paraffin

**Cholesky** (unstructured parallelism)

- Ada tasklets (OpenMP)
- Ada + OpenMP dependences

Perfect speed-up

Great speed-up enhancement

Memory boundaries NUMA effect

Related work
Analyze Ada/OpenMP programs for data-race detection
Compiler analysis for Ada/OpenMP programs

Currently:

– Ada **lacks static analyses** for data-race detection
– OpenMP correctness* techniques do not **consider concurrency**

---

Currently:
- Ada lacks static analyses for data-race detection
- OpenMP correctness* techniques do not consider concurrency

Solution:
- Extend current OpenMP techniques*

Currently:
– Ada lacks static analyses for data-race detection
– OpenMP correctness* techniques do not consider concurrency

Solution:
– Extend current OpenMP techniques*

The Ada Ravenscar profile eases the generation of blocks of concurrency because dynamic task allocation and task termination are forbidden

Solve race conditions in Ada/OpenMP

1. Build an interprocedural PCFG
2. Recognize the different blocks of concurrency
3. Apply the following solutions if race conditions may arise:

<table>
<thead>
<tr>
<th>Race condition between</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada tasks</td>
<td>Ada mechanisms: protected object</td>
</tr>
<tr>
<td>Ada and OpenMP tasks</td>
<td></td>
</tr>
<tr>
<td>different binding regions(^1)</td>
<td>OpenMP mechanisms(^2):</td>
</tr>
<tr>
<td>same binding region(^1)</td>
<td>- Synchronization constructs and clauses: taskwait, barrier, depend</td>
</tr>
<tr>
<td></td>
<td>- Mutual exclusion constructs: critical, atomic</td>
</tr>
<tr>
<td></td>
<td>- Data-sharing attributes: private, firstprivate, lastprivate</td>
</tr>
</tbody>
</table>

\(^1\) Binding region: the enclosing region that determines the execution context and limits the scope of the effects of the bound region.

\(^2\) S. Royuela et al., “Compiler Analysis for OpenMP Tasks Correctness”, CF2015.
Ravenscar application (HRT-HOOD)

Ravenscar application (HRT-HOOD)

package body Production_Workload is
  type Dim is range 1..512;
  type M is array (Dim, Dim) of Float;
  M_A, M_B, M_C: M;

  procedure Read_Sensor_A is
  begin
    pragma OMP (parallel);
    pragma OMP (single);
    pragma OMP (taskloop);
    for I in Dim loop
      for J in Dim loop
        M_A(I, J) := sensor(1, I, J);
      end loop;
    end loop;
  end Read_Sensor_A;

  procedure Read_Sensor_B is
  begin
    pragma OMP (parallel);
    pragma OMP (single);
    pragma OMP (taskloop);
    for I in Dim loop
      for J in Dim loop
        M_B(I, J) := sensor(2, I, J);
      end loop;
    end loop;
  end Read_Sensor_B;

  procedure Fuse_Sensors is
  begin
    pragma OMP (parallel);
    pragma OMP (single);
    pragma OMP (taskloop);
    for I in Dim loop
      for J in Dim loop
        M_C(I, J) := M_A(I, J) + M_B(I, J);
      end loop;
    end loop;
  end Fuse_Sensors;

  procedure Small_Whetstone (Workload:Positive) is
  begin
    case Workload is
      when 1 => Read_Sensor_A;
      when 2 => Read_Sensor_B;
      when 3 => Fuse_Sensors;
      when others => null;
    end case;
  end Small_Whetstone;

end Production_Workload;
Ravenscar application (PCFG)

Evaluation

Manually tested!
Ongoing implementation

SHARED OBJECT

- Read
- Write
- Read/Write

On_CallProducer

ActivationManager.Synchronize.Activation_Cyclic

Production_Workload.Small_Wheatstone

Auxiliary.Due_Activation

On_CallProducer.Start

Auxiliary.Check_Due

ActivationLogReader.Signal

Request_Buffer.Deposit

Factor

Run_Count

Request.Counter

Insert_Index

Current_Size

Barrier

My_Request_Buffer(Insert_Index)

Task_Start_Time

System_Start_Time

System_Start_Time

ActivationLogReader.Wait

Production_Workload.Small_Wheatstone

ActivationLogReader.Read

Small_Wheatstone

Barrier

Poll_Time

Wait_Time

Poll_Time = Poll_Time + WaitTime;


Delay until Poll_Time;

EventQueue.Handler.Wait

EventQueue.Handler.Signal

ActivationLogWrite

ActivationCounter

ActivationTime

LocalSuspension_Object

Evaluation

2 blocks of concurrency

Elaboration time

Program execution
Ravenscar application (PCFG)
Ravenscar application (PCFG)
Ada moves towards introducing fine-grain mechanisms for parallel execution

The tasklet model covers some important aspects but has several limitations that may be overcome by OpenMP

Mixing Ada with OpenMP introduces complexities for static analysis because it mixes concurrency with parallelism

Ada lacks mechanisms for data-race detection and OpenMP mechanisms only consider parallelism

OpenMP mechanisms can be used by properly representing concurrency in the PCFG

Non-Ravenscar applications can be tackled by further enriching the PCFG
Safe Parallelism

Compiler Analysis Techniques for Ada and OpenMP

Reach us at:

Sara Royuela    sara.royuela@bsc.es
Eduardo Quiñones eduardo.quinones@bsc.es
Luis Miguel Pinho lmp@isep.ipp.pt