Enabling performance portability of data-parallel OpenMP applications on asymmetric multicore processors

Juan Carlos Sáez[*],Fernando Castro[*], Manuel Prieto-Matías[*,†]

[*]Facultad de Informática
[†]Instituto de Tecnología del Conocimiento (ITC)
Complutense University of Madrid, Spain

49th International Conference on Parallel Processing (ICPP '20)

August 17-20, 2020

This research has been supported by

GOBIERNO DE ESPAÑA

MINISTERIO DE ECONOMÍA Y COMPETITIVIDAD

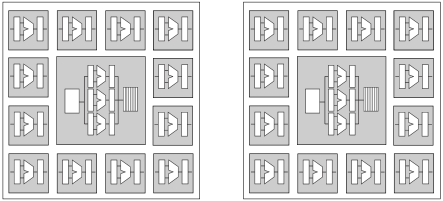
Grant references: RTI2018-093684-B-I00 and S2018/TCS-4423

FEDER

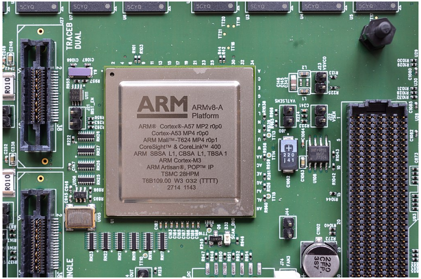
Comunidad de Madrid

Asymmetric Multicore Processors (AMPs)

- Performance asymmetry: big cores + small cores
- Same Instruction Set Architecture (ISA) but different features:
  - Processor frequency and power consumption
  - Microarchitecture
    - In-order vs. out-of-order pipeline
    - Retirement/issue width
  - Cache(s) size and hierarchy

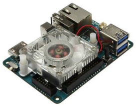


Example: ARM big.LITTLE processor



e.g., Google Pixel 7

e.g., Samsung Galaxy A8



Odroid XU-4

Hikey 960

ARM Juno Platform

Intel Lakefield's *hybrid* processor

1 Sunny Cove core + 4 Tremont cores

Samsung Galaxy Book S

Microsoft Neo Surface

Our goal

- **Goal:** Automatically deliver good performance to data-parallel loop-based OpenMP programs on AMPs

Our goal

- **Goal:** Automatically deliver good performance to data-parallel loop-based OpenMP programs on AMPs

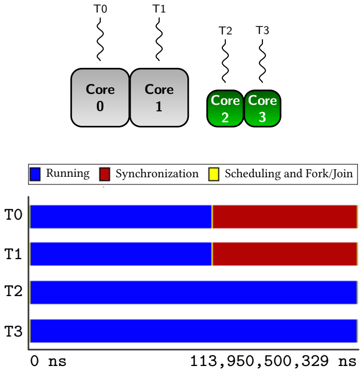
- Main limiting factors for scalability of loop-based OpenMP programs
  1. Phases with limited parallelism (e.g. sequential sections)
  2. Load imbalance in iteration distribution
  3. Shared-resource contention (Last-level cache, memory bandwidth)

Issue AMPs

Cores with different performance introduce load imbalance inherently

Load imbalance on AMPs

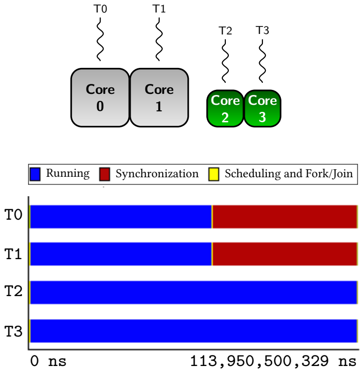
Application with a single parallel loop runs on AMP (2 big cores + 2 small cores)



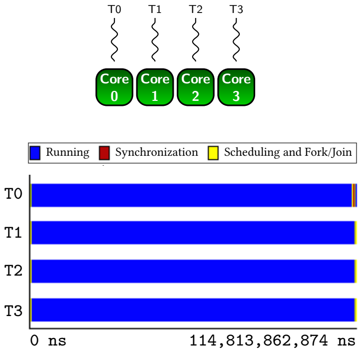
- Legacy OpenMP code targets symmetric multicore
- The `static` schedule is used as iterations have similar amount of work
  - Each thread runs same # of iterations
- Execution of *unmodified* application on an AMP

Load imbalance on AMPs

Application with a single parallel loop runs on AMP (2 big cores + 2 small cores)



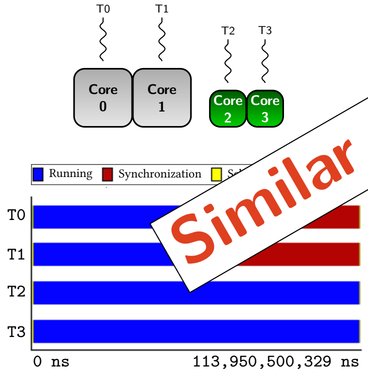
Application with a single parallel loop runs on sCMP (4 small cores)



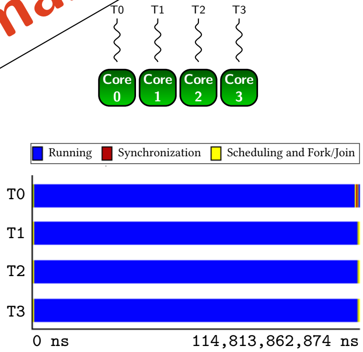
ArTeCS

Load imbalance on AMPs

Application with a single parallel loop runs on AMP (2 big cores + 2 small cores)



Application with ... gle parallel loop runs on ... nall cores)

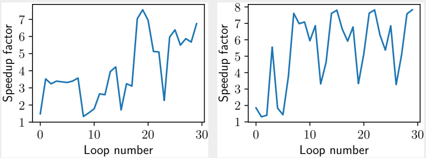


Similar performance!

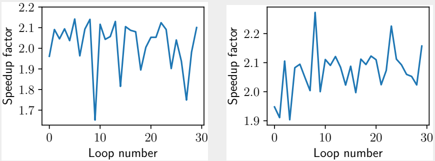
Addressing the load imbalance

- *Cannot just we assign more iterations to big-core threads in proportion to the big-to-small relative performance?*
  - Speedup Factor (SF)[1] ⇒ big-to-small relative performance: $\frac{Ctime_{small}}{Ctime_{big}}$

SF for BT and CG on Platform A



SF for BT and CG on Platform B

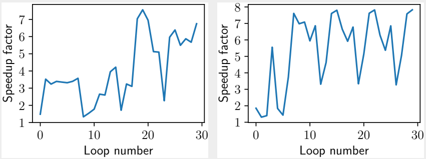


[1] For these experiments, the SF was measured with the ratio of completion times (small-to-big) registered for each loop running with a single thread

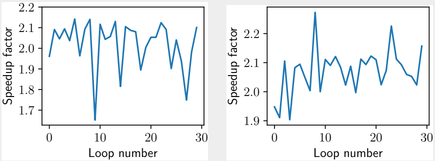
Addressing the load imbalance

- *Cannot just we assign more iterations to big-core threads in proportion to the big-to-small relative performance?*
  - Speedup Factor (SF)[1] $\Rightarrow$ big-to-small relative performance: $\dfrac{Ctime_{small}}{Ctime_{big}}$

SF for BT and CG on Platform A



SF for BT and CG on Platform B



SF is not only platform- and application- specific but may also vary across loops

[1] For these experiments, the SF was measured with the ratio of completion times (small-to-big) registered for each loop running with a single thread

Our proposal

- We proposed three asymmetry-aware loop-scheduling methods
  - **AID:** *Asymmetric Iteration Distribution*
  - Replacements for `static` and `dynamic` methods on AMP
    - Cater to the demands of different applications

Our proposal

- We proposed three asymmetry-aware loop-scheduling methods
  - **AID:** *Asymmetric Iteration Distribution*
  - Replacements for `static` and `dynamic` methods on AMP
    - Cater to the demands of different applications

Features

- Implemented in *libgomp* (GNU OpenMP runtime system)
- Applications need to be recompiled, but no changes required in source code
- The same binary can be used on different platforms with the same ISA
  - The runtime system automatically adapts to the platform

Contents

Contents

Contents

AID loop-scheduling methods

- 3 variants of **Asymmetric-Iteration Distribution (AID)**
  1. `AID-static`: replacement for `static` on AMPs
  2. `AID-hybrid`: “safer” version of `AID-static`
  3. `AID-dynamic`: replacement for `dynamic` on AMPs

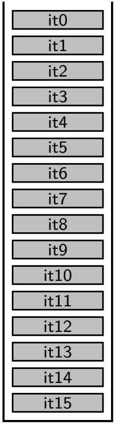
AID loop-scheduling methods

- 3 variants of **Asymmetric-Iteration Distribution (AID)**
  1. `AID-static`: replacement for `static` on AMPs
  2. `AID-hybrid`: "safer" version of `AID-static`
  3. `AID-dynamic`: replacement for `dynamic` on AMPs

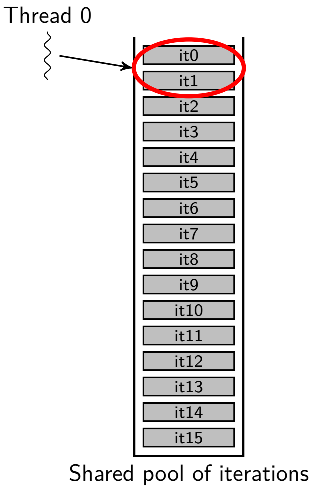
Common aspects

- Usually assign more loop iterations to big-core threads than to small-core threads
  - Based on the loop's SF (predicted at runtime)
- Designed for scenarios with no oversubscription
- There is no need to modify applications to activate them
  - Environment variables for enabling and setting parameters

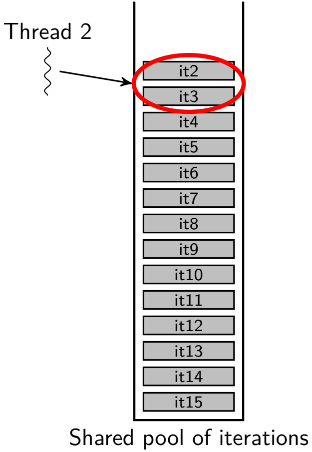
Implementation of `dynamic` schedule in *libgomp*



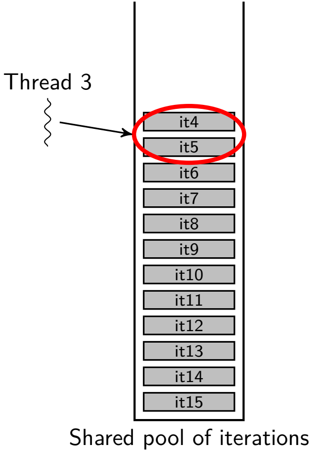
Implementation of `dynamic` schedule in *libgomp*



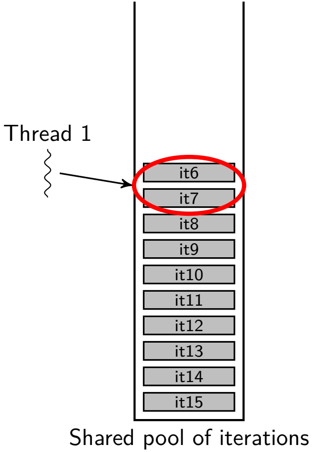
Implementation of `dynamic` schedule in *libgomp*



Implementation of `dynamic` schedule in *libgomp*



Implementation of `dynamic` schedule in *libgomp*

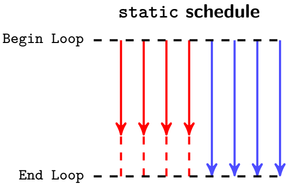


Lock-free implementation

- 2 shared counters: `next` and `end`
- *chunk* (default value 1)
- Uses `fetch-and-add`
  - Atomic: `next+=chunk`
- Each thread invokes `gomp_iter_dynamic_next()` until `next>=end`

AID-Static

Designed for loops where iterations have the same amount of work

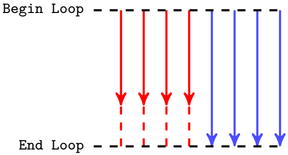


- *All threads are allotted "the same" amount of iterations*
- Big-core threads complete their share earlier causing imbalance

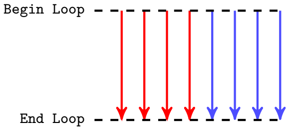
AID-Static

Designed for loops where iterations have the same amount of work

`static` **schedule**



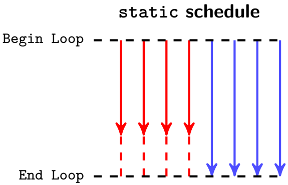
`AID-static`

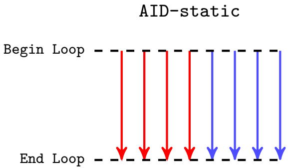


- *All threads are allotted "the same" amount of iterations*
- Big-core threads complete their share earlier causing imbalance

AID-Static

Designed for loops where iterations have the same amount of work





- *All threads are allotted "the same" amount of iterations*
- Big-core threads complete their share earlier causing imbalance

- Small-core threads $\rightarrow$ k iterations
- Big-core threads $\rightarrow$ $SF \cdot k$ iterations
- $total\_iterations = N_{big} \cdot SF \cdot k + N_{small} \cdot k$

AID-Static

Designed for loops where iterations have the same amount of work

- *All threads are allotted "the same" amount of iterations*
- Big-core threads complete their share earlier causing imbalance

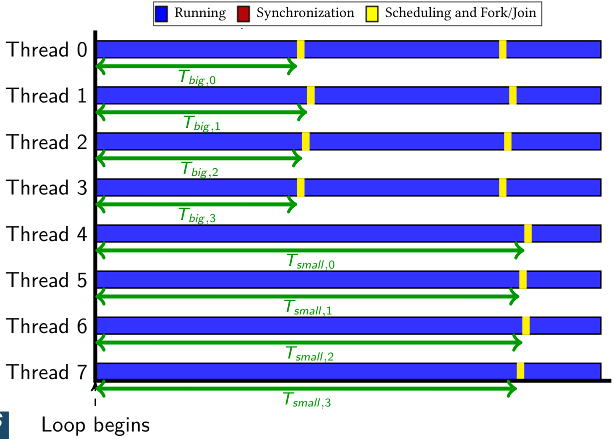
- Small-core threads $\rightarrow k$ iterations
- Big-core threads $\rightarrow SF \cdot k$ iterations
- $total\_iterations = N_{big} \cdot SF \cdot k + N_{small} \cdot k$

$$k = \frac{total\_iterations}{N_{big} \cdot SF + N_{small}}$$

AID-Static: SF prediction

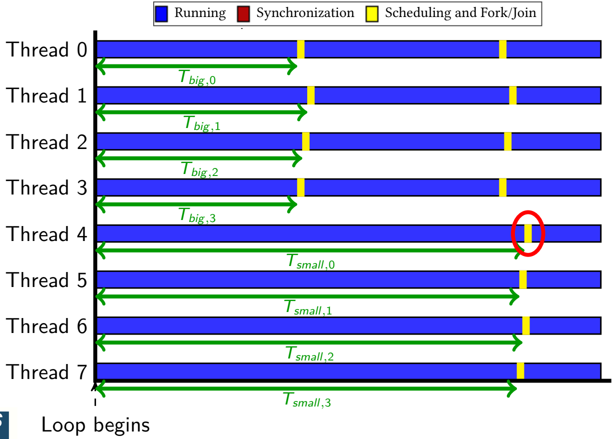
- Run *chunk* iterations on big-cores and on small-core threads
- Last thread that completes sampling is the one that calculates *SF* and *k*
- $$SF = \frac{\frac{1}{N_{small}} \cdot \sum_{i=0}^{N_{small}-1} T_{small,i}}{\frac{1}{N_{big}} \cdot \sum_{j=0}^{N_{big}-1} T_{big,j}}$$

AID-Static: SF prediction



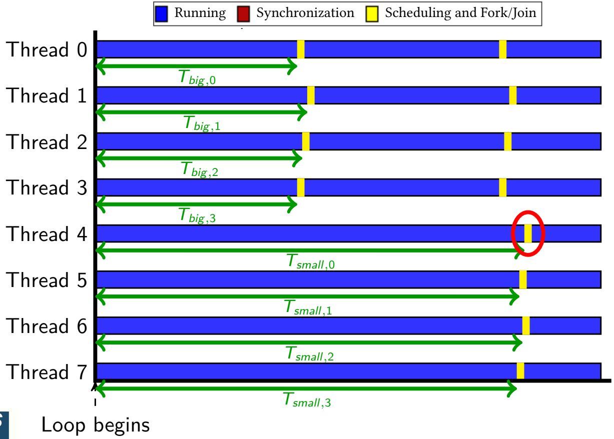
- Run *chunk* iterations on big-cores and on small-core threads
- Last thread that completes sampling is the one that calculates *SF* and *k*
- $$SF = \frac{\frac{1}{N_{small}} \cdot \sum_{i=0}^{N_{small}-1} T_{small,i}}{\frac{1}{N_{big}} \cdot \sum_{j=0}^{N_{big}-1} T_{big,j}}$$

AID-Static: SF prediction



- Run *chunk* iterations on big-cores and on small-core threads
- Last thread that completes sampling is the one that calculates *SF* and *k*
- $SF = \dfrac{\frac{1}{N_{small}} \cdot \sum_{i=0}^{N_{small}-1} T_{small,i}}{\frac{1}{N_{big}} \cdot \sum_{j=0}^{N_{big}-1} T_{big,j}}$

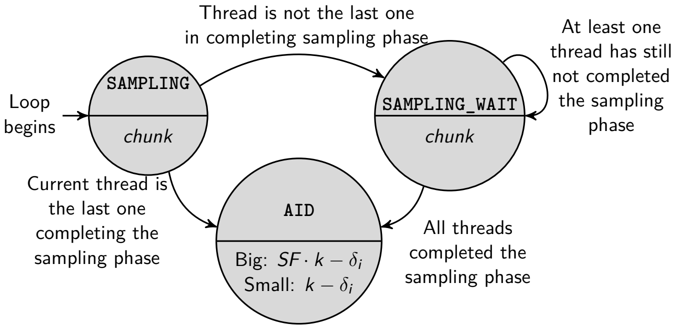
AID-Static: SF prediction



- Efficient lock-free implementation
- Threads complete iterations even during the sampling phase (δ_i)
- Each thread needs to gather 2 timestamps (*vsyscall*)
- Shared counters to maintain aggregate completion time

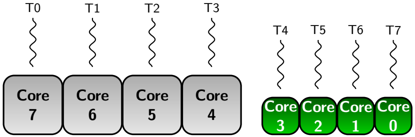
AID-Static: Implementation

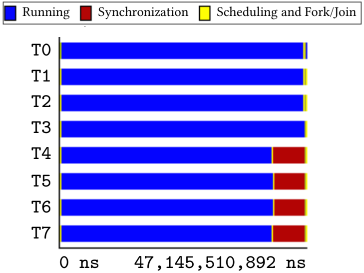
- Threads in 3 possible states
  - A state transition may occur when the thread “steals” work from the shared pool



AID-static: Limitations

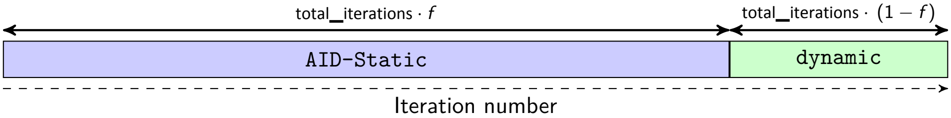
- Predicted SF may not be representative throughout the loop
  - Processing varies slightly across iterations
  - SF misprediction



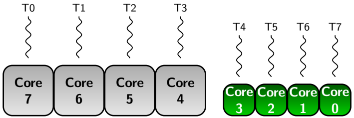


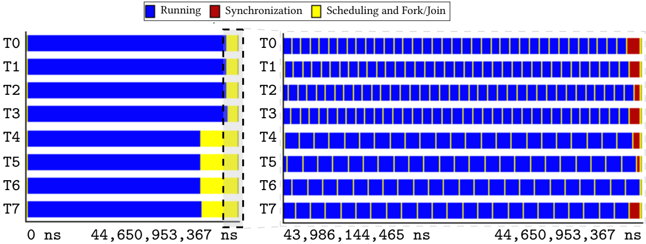
AID-Static could introduce load imbalance

AID-Hybrid: Implementation



- AID-hybrid: AID-static + OpenMP's dynamic
  - f is a configurable parameter (percentage)





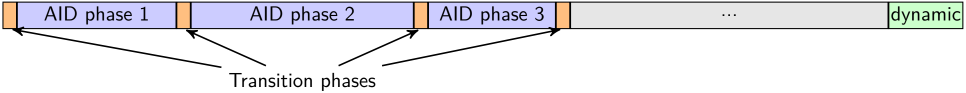
AID-dynamic

- Goal: To make a good replacement for `dynamic` on AMPs
- It relies on two configurable *chunk* values:
  - **major** (M): Used for AID phases (variant of `dynamic`)
    - small-core threads $\rightarrow$ M iterations
    - big-core threads $\rightarrow$ $M \cdot R$ iterations
    - $R = g(SF)$
  - **minor** (m): Used in between AID phases and at the end of the loop's execution

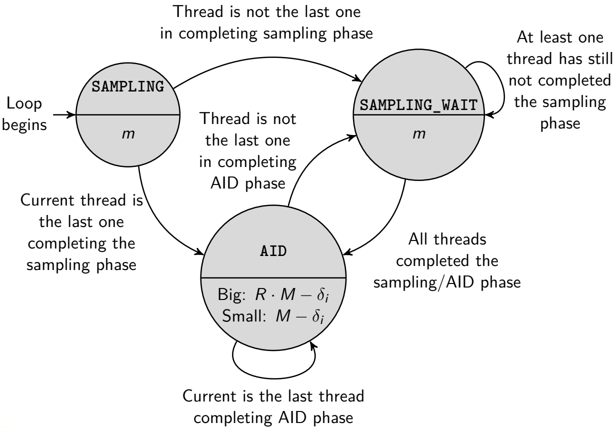
```
mode=AID;
cur_aid_phase=0;

while (!pool.is_empty()) {
  if (pool.remaining_iter()<=M*nr_threads)
      mode=DYNAMIC;

  if (mode==AID &&
    prev_phase_completed(cur_aid_phase)){
      R=calculate_progress(cur_aid_phase);
      chunk=big_core_thread()?R*M:M;
      dynamic(chunk,pool);
      current_aid_phase++;
  }
  else {
     dynamic(m,pool);
  }
}
```



AID-dynamic



$$R(t+1) = \begin{cases} SF & t = 0 \\ R(t) \cdot \dfrac{AvgTimeAID_{small}(t)}{AvgTimeAID_{big}(t)} & t > 0 \end{cases}$$

Required changes in the GCC compiler

- To guarantee performance portability with our proposal:
  1. The runtime system must be deployed as a dynamic library (`libgomp.so`)
  2. The compiled program must invoke loop-related runtime API calls
- **Issue**: GCC omits loop-related API calls when `schedule` clause not provided

```
...
#pragma omp for
  for (j = 0; j < grid_points[1]; j++) {
    eta = (double)j * dnym1;
    for (k = 0; k < grid_points[2]; k++) {
      zeta = (double)k * dnzm1;
      exact_solution(xi, eta, zeta, temp);
      for (m = 0; m < 5; m++) {
        u[i][j][k][m] = temp[m];
      }
    }
  }
```

Terminal

```
$ nm -u bt.B | grep -i GOMP_
  U GOMP_barrier@@GOMP_1.0
  U GOMP_parallel@@GOMP_4.0
```

The runtime system cannot control the schedule of those loops

Required changes in the GCC compiler

- We changed *default* value for `schedule` clause in GCC: `static` → `runtime`
  - If clause omitted, runtime uses schedule defined in `OMP_SCHEDULE` env. variable
  - Very simple change in GCC 8.3: `omp_extract_for_data()` at *gcc/omp-general.c*

```c
...
#pragma omp for
  for (j = 0; j < grid_points[1]; j++) {
    eta = (double)j * dnym1;
    for (k = 0; k < grid_points[2]; k++) {
      zeta = (double)k * dnzm1;
      exact_solution(xi, eta, zeta, temp);
      for (m = 0; m < 5; m++) {
        u[i][j][k][m] = temp[m];
      }
    }
  }
```

Terminal

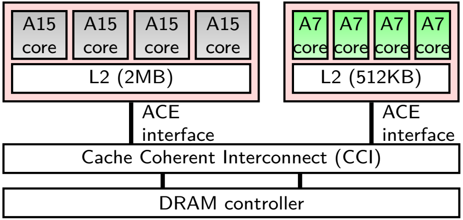
```
$ nm -u bt.B_modified | grep -i GOMP_
  U GOMP_loop_end@@GOMP_1.0
  U GOMP_loop_end_nowait@@GOMP_1.0
  U GOMP_loop_runtime_next@@GOMP_1.0
  U GOMP_loop_runtime_start@@GOMP_1.0
  U GOMP_parallel@@GOMP_4.0
```

Runtime system is now notified when each loop begins (`GOMP_loop_*_start()`) and when each thread requests work to be assigned to it (`GOMP_loop_*_next()`)

Contents

Experimental platforms

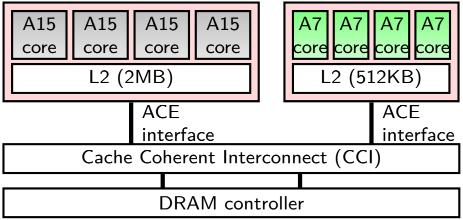
Platform A (Odroid-XU4 Board)



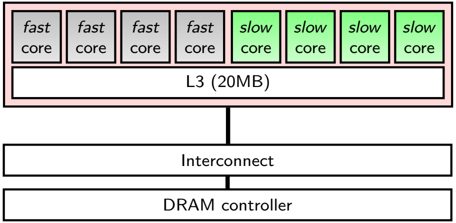
- 32-bit ARM big.LITTLE processor
  - 4 x Cortex A15 *big* cores @ 2.0Ghz
  - 4 x Cortex A7 *small* cores @ 1.5Ghz
- 2GB LPDDR3 SDRAM @ 933MHz

Experimental platforms

Platform A (Odroid-XU4 Board)



Platform B (Intel server platform)



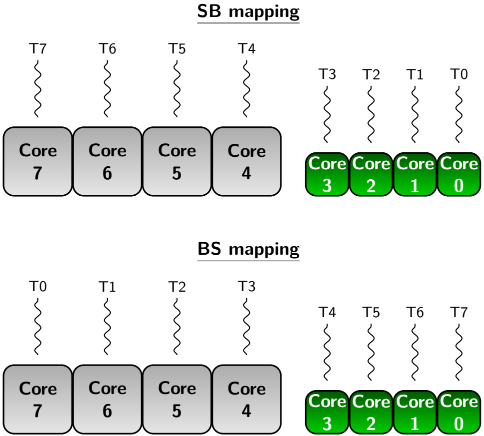
- 32-bit ARM big.LITTLE processor
  - 4 x Cortex A15 *big* cores @ 2.0Ghz
  - 4 x Cortex A7 *small* cores @ 1.5Ghz
- 2GB LPDDR3 SDRAM @ 933MHz

- 64-bit Intel Xeon E5-2620 v4 (Broadwell-EP)
  - 4 x *fast* cores @ 2.1Ghz
  - 4 x *slow* cores @ 1.2Ghz and 87.5% duty cycle
- 32GB DDR4 SDRAM @ 2133MHz

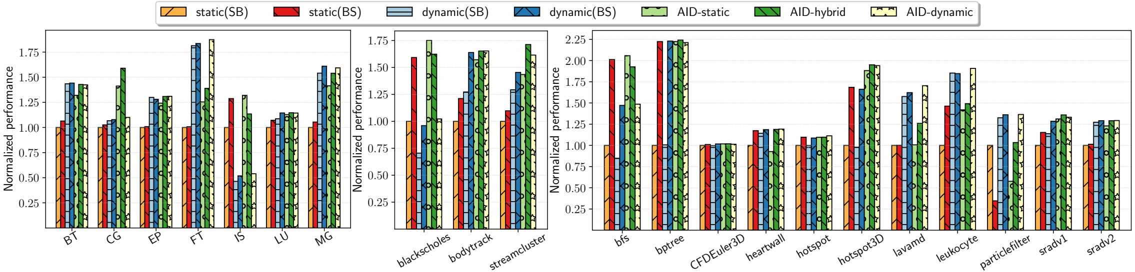
ArTeCS

Applications and thread-to-core mappings

- **21 OpenMP benchmarks**
  - NAS Parallel
  - PARSEC 3
  - Rodinia
- **GCC** 8.3 + **Linux** kernel 4.14.165
- Evaluated loop-scheduling methods
  - `static` (BS and SB)
  - `dynamic` (BS and SB)
  - `guided` (BS and SB)
  - `AID-static`
  - `AID-hybrid`
  - `AID-dynamic`



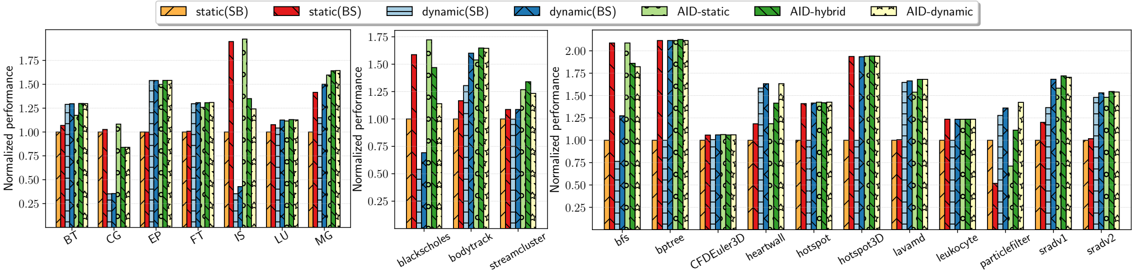
Relative performance on Platform A



- Running the master thread on a big core brings substantial improvements in some cases
- `AID-static` and `AID-hybrid` make good replacements for `static` (up to 30.7% and 56% improvement)
- OpenMP `dynamic` and `AID-dynamic` perform in a close range but a >10% improvement is observed

ArTeCS

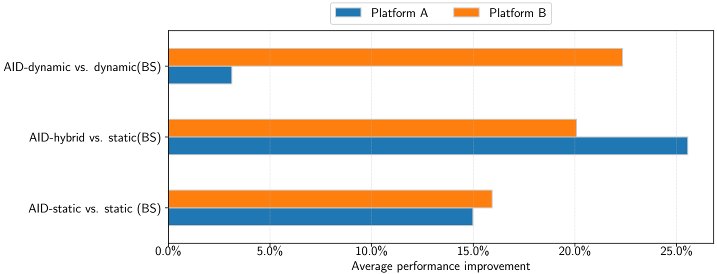
Relative performance on Platform B



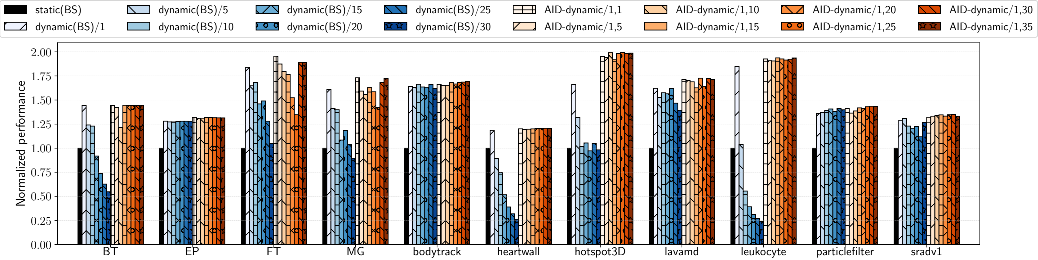
- Smaller big-to-small performance ratios (max. 2.3x vs. 8.9x on Platform A)
- The overhead of `dynamic` negates its benefits in some cases due to lower SF values
  - `AID-dynamic` delivers higher gains vs. `dynamic` on this platform (22% on average)

ArTeCS

Average relative performance



Dynamic vs AID-dynamic: different chunk values



- The average improvement with best chunk settings for AID-dynamic vs. static is 5.5%
- `AID-dynamic` delivers up to a 21.9% performance improvement
- With `AID-dynamic` performance is less sensitive to the choice of the chunk values

Contents

ArTeCS

Conclusions

- Conventional OpenMP loop-scheduling methods are not suitable for AMPs
  - `static` introduces load imbalance
  - `dynamic` better than `static` but subject to high overhead

Conclusions

- Conventional OpenMP loop-scheduling methods are not suitable for AMPs
  - `static` introduces load imbalance
  - `dynamic` better than `static` but subject to high overhead
- We proposed 3 alternative asymmetry-aware loop-scheduling methods
  - Implemented in *libgomp* (GCC 8.3)
  - No changes required in application code
  - Applications must be recompiled with our modified compiler

Conclusions

- Conventional OpenMP loop-scheduling methods are not suitable for AMPs
  - `static` introduces load imbalance
  - `dynamic` better than `static` but subject to high overhead
- We proposed 3 alternative asymmetry-aware loop-scheduling methods
  - Implemented in *libgomp* (GCC 8.3)
  - No changes required in application code
  - Applications must be recompiled with our modified compiler
- Our experimental evaluation on real AMP hardware reveals their effectiveness
  - `AID-static`, `AID-hybrid` outperform `static` by up to 30.7% and 56%, respectively
  - `AID-dynamic` improves `dynamic` by up to 16.8%
    - Higher relative improvements when using the best chunk settings for each application

Future Work

1. Explore the potential from using multiple AID methods in the same application
   - Loops with same-sized iterations → `AID-static` or `AID-hybrid`
   - Loops amenable to `dynamic` → `AID-dynamic`
   - Requires making changes in the application and parameter-tunning + profiling

Future Work

1. Explore the potential from using multiple AID methods in the same application
   - Loops with same-sized iterations → `AID-static` or `AID-hybrid`
   - Loops amenable to `dynamic` → `AID-dynamic`
   - Requires making changes in the application and parameter-tunning + profiling
2. Leverage AID in multi-application scenarios
   - Devise interaction mechanisms between the OS and the runtime system

Future Work

1. Explore the potential from using multiple AID methods in the same application
   - Loops with same-sized iterations → `AID-static` or `AID-hybrid`
   - Loops amenable to `dynamic` → `AID-dynamic`
   - Requires making changes in the application and parameter-tunning + profiling
2. Leverage AID in multi-application scenarios
   - Devise interaction mechanisms between the OS and the runtime system
3. Evaluate the effectiveness of AID in other types of applications and heterogeneous platforms