OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

Florina M. Ciorba¹
hpc.dmi.unibas.ch

ADAC6 Workshop Zurich, 20.06.2018

Work conducted with Christian Iwainsky² and Patrick Buder¹, to appear at iWomp18

¹ University of Basel, Switzerland
² Technische Universität Darmstadt, Germany
Scheduling is a vital part of any successful effort of coordinating and managing parallelism in high performance computers. Remains a challenge, at several levels, for Exscale computing. For compute-intensive applications with irregular (nested) parallelism.

Multiple types, levels, and forms of parallelism.

Focus of the SNSF project Multilevel Scheduling in Large Scale High Performance Computers (2017-2020), p3.snf.ch/project-169123

* ETP4HPC SRA2: 5.2 System software (kernel and run-time), 5.3 Prog. env., 5.7 Math. and algo. for extreme scale HPC systems
** IESP 2.0: Runtimes, compilers, applications, algorithms, performance optimization
... To Multiple Types, Levels, and Forms of Parallelism in Parallel Computing

Multiscale modeling

- **Macroscopic Scale**
  - Space: 1 mm³ - 1 km³
  - Time: s-h

- **Mesoscopic Scale**
  - Space: 0.1-10.1 mm³
  - Time: ms

- **Microscopic Scale**
  - Space: 0.1-15 µm³
  - Time: ns

- **Atomistic Scale**
  - Space: 1-300 nm³
  - Time: 0-1 ps

- **Electronic Scale**
  - Space: 2-10 Å
  - Time: 0-1 fs

Hardware parallelism

- **Global Grid Scale**
  - Parallelism: 100x → 1,000x sites
  - Time: s-h

- **Local Grid or HPC Site Scale**
  - Parallelism: 2*10⁴ → 10⁵-10⁶ nodes
  - Time: ms-h

- **Node Scale**
  - Parallelism: 2-4 → 8-16 sockets
  - Time: 10⁸ instructions × 1 ns

- **Chip or Socket (MIMD) Scale**
  - Parallelism: 8-12 → 32-256 CPU cores
    - 5,000 → 10,000 GPU cores
    - 40-60 → 100 Co-proc. cores
  - Time: 10⁶ instructions × 1 ns

- **Core Scale**
  - Parallelism: 1.8 scalar:vector ratio
  - Time: 10³ instructions × 1 ns

- **Vector Scale**
  - Parallelism: 2-4 → 16-256 data items/vector (accelerator core)
    - 16→1,024 data items/vector (CPU core)
  - Time: vector length × 1 ns

- **Pipeline Scale**
  - Parallelism: 10 → less instructions (1-thread)
    - 30 → 100 instructions (multithread)
  - Time: several ns

- **Instruction Scale**
  - Parallelism: 3-5 → less instructions
  - Time: 1 ns

Software parallelism

- **Global Batch Scale**
  - Parallelism:
  - Time: number of sites

- **Local Batch Scale**
  - Parallelism:
    - 10⁰-10⁷ processes x number of jobs / time period

- **Job Scale**
  - Parallelism:
    - 10⁰-10⁷ processes or
    - 10⁰-10¹⁰ threads

- **Process Scale**
  - Parallelism:
    - 5,000 CPU threads
    - 50,000 GPU threads

- **Thread Scale**
  - Parallelism:
    - 8-12→32-256 CPU threads
    - 5,000→10,000 GPU threads
    - 40-60→100 Co-proc. threads

- **SIMD Instruction Scale**
  - Parallelism: 1 SIMD instruction

- **SISD Instruction (scalar) Scale**
  - Parallelism: 1 SISD instruction

This talk
Increasing Hardware Parallelism

Through increased node count, CPU core count (multi- and manycore), and accelerator core count

<table>
<thead>
<tr>
<th></th>
<th>Piz Daint @ CSCS</th>
<th>SUMMIT @ ORNL</th>
<th>TSUBAME 3.0 @ TokyoTech</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU cores/node</td>
<td>$1 \times 12$ (Xeon XC50)</td>
<td>$2 \times 22$ (Power9)</td>
<td>$2 \times 14$ (Xeon)</td>
</tr>
<tr>
<td></td>
<td>$2 \times 18$ (Xeon XC40)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU cores/node</td>
<td>$1 \times 3,584$ (CUDA P100) (XC50)</td>
<td>$6 \times 640$ (Tensor V100)</td>
<td>$4 \times 3,584$ (CUDA P100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$6 \times 5,120$ (CUDA V100)</td>
<td></td>
</tr>
<tr>
<td>Nodes</td>
<td>5,320 (XC50)</td>
<td>4,608</td>
<td>540</td>
</tr>
<tr>
<td></td>
<td>1,813 (XC40)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Intel Xeon Phi x200 Knights Landing ≤ 72 CPU cores, 4 hardware threads/core
Parsing the Title
“OpenMP Loop Scheduling”

✧ **Loops** typically come to mind in the context of shared memory systems
✧ Application and underlying system characteristics determine the best schedule
  ✧ No “one-size-fits-all” loop scheduling technique can address all
    ✧ Sources of load imbalance for
    ✧ Types of scientific applications on
    ✧ Types of computing platforms

✧ **OpenMP**: 20+ years industry standard for shared-memory parallel programming
  ✧ Widely used to parallel program a broad variety of applications
  ✧ Supported by a growing number of hardware and software vendors
  ✧ Several benchmark suites for performance evaluation (SPECComp, NAS)

✧ **Scheduling**: performance critical aspect of loops and important part of most OpenMP programs
  ✧ Not overshadowed by the introduction of explicit tasks in OpenMP
  ✧ Nor by the accelerated computing APIs

✧ The impact of system-induced variability is often neglected in loop scheduling research, particularly by OpenMP schedules
OpenMP standard `schedule()`

- **static,chunk**: predetermined allocation order offset by thread ID
- **dynamic,1**: pure self-scheduling SS [Lusk, Overbeek ‘83]
- **dynamic,chunk**: chunk self-scheduling **CSS** [Kruskal, Weiss ‘85]
- **guided**: guided self-scheduling **GSS** [Polychronopoulos, Kuck ‘87]
- **guided,chunk**: **GSS** with minimum chunk size
- **auto**: implementation determines schedule; no “chunk” support

Shared-memory self-schedules not in standard

- **tss**: trapezoid self-scheduling **TSS** [Tzen, Ni ‘93]
- **fac2**: practical factoring **FAC** [Flynn Hummel et al. ’90-92]
- **wf2**: practical weighted factoring **WF** [Flynn Hummel et al. ’96]
- **rand**: random self-scheduling **RAND**
- **taper**: tapering strategy
- **bold**: bold strategy

Are these schedules good enough to efficiently exploit HW parallelism in 2018+?

Are these schedules sufficient for all apps and systems?

Are there any other schedules not yet in OpenMP?

YES
### Parsing the Title

“Loop Scheduling Revisited”

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Work Queue</th>
<th>Optimization Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Assignment</td>
<td>Load Balancing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ordering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Timing</td>
</tr>
<tr>
<td>Fully static</td>
<td>compilation</td>
<td>compilation</td>
</tr>
<tr>
<td>(pre-scheduling)</td>
<td></td>
<td>compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>compilation central</td>
</tr>
<tr>
<td></td>
<td></td>
<td>central</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ scheduling overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imbalance**</td>
</tr>
<tr>
<td>Work sharing</td>
<td>compilation</td>
<td>compilation</td>
</tr>
<tr>
<td>(static allocation)</td>
<td></td>
<td>compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>central</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ scheduling overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imbalance**</td>
</tr>
<tr>
<td>Affinity &amp;</td>
<td>compilation</td>
<td>compilation</td>
</tr>
<tr>
<td>Work stealing</td>
<td></td>
<td>compilation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>central</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ locality</td>
</tr>
<tr>
<td></td>
<td></td>
<td>load imbalance**</td>
</tr>
<tr>
<td></td>
<td></td>
<td>scheduling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>overhead</td>
</tr>
<tr>
<td>Fully dynamic</td>
<td>execution</td>
<td>execution</td>
</tr>
<tr>
<td>(self-scheduling)</td>
<td></td>
<td>execution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>central</td>
</tr>
<tr>
<td></td>
<td></td>
<td>½ scheduling overhead</td>
</tr>
<tr>
<td></td>
<td></td>
<td>imbalance**</td>
</tr>
<tr>
<td>load imbalance**</td>
<td>induced by problem and algorithm</td>
<td>½ one goal vs. explicit trade-off</td>
</tr>
<tr>
<td>load imbalance***</td>
<td>induced by problem, algorithm, and system</td>
<td>½ another goal two optimization goals</td>
</tr>
</tbody>
</table>

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules
Static Scheduling and Work Sharing

**Cyclic**
- Iteration $i$ is assigned to processor $i \mod P$.
- Produces more balanced schedules than block scheduling for some non-uniformly distributed parallel loops.

**Block-D**
- Loop is scheduled and data are partitioned to increase locality.
- If loop scheduling is blocked and matches data partitioning = Block-D.

**Cyclic-D**
- Loop is scheduled and data are partitioned to increase locality.
- If both loop scheduling occurs in a cyclic fashion and matches the data partitioning = Cyclic-D

**Polyhedral Compilation**
- Based on workload balancing.
- Workload is equally distributed onto all the processors.
- Proposes an equation for the upper bounds of the workload balance scheduling.

**Others?**
- Polyhedral Compilation
  - polyhedral.info
  - automatic parallelization,
  - data locality optimizations,
  - memory management optimizations,
  - program verification,
  - communication optimizations,
  - SIMDization,
  - code generation for hardware accelerators, high-level synthesis

---

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

ADAC6 | Zurich | June 20, 2018
Affinity Scheduling and Work Stealing

- 1992
  - AFS
    - Dynamic Partitioned Affinity Scheduling
    - Wrapped Partitioned Affinity Scheduling
  - SAS
    - Self-Adjusting Scheduling

- 1993
  - AFS
    - Markatos, LeBlanc
  - SAS
    - Hamidzadeh, Lilja
  - LFS
    - Li et al.
  - LDS
    - Local Dynamic Scheduling

- 1994
  - AFS
    - Subramanian, Eager
  - SAS
    - Eager

- 1995
  - AFS
    - Wang et al.
  - MAFS
    - Wang, Chang
  - CAFS
    - Wang et al.

- 1997
  - AFS
    - Markatos, LeBlanc
  - SAS
    - Eager, LA, CA, GA, HA
    - Yan et al.
  - Work stealing
    - Blumofe, Leiserson

- 1999
  - Work stealing
    - Blumofe, Leiserson
    - Linearly adaptive
    - Greedily adaptive
    - Heuristic adaptive
    - Exploit dynamic information

- 2001
  - HS
    - Olivier et al.
    - Hierarchical Scheduling
    - Two levels: process and thread

- 2002
  - Discrete FGDS
    - Tabirca et al.
    - O(p + log p)
  - Continuous FGDS
    - Tabirca et al.
    - Feedback Guided Scheduling
  - Work Dealing
    - Hendler, Shavit
    - Low-overhead alternative to Work Stealing

- 2003
  - Continuous FGDS
    - Tabirca et al.
    - O(log p)

- 2006
  - 2D-FGS, 2D-FGLS, 2D-DYN, 2D-AFS, 2D-DAFS
    - Feedback Scheduling
    - Feedback Guided Scheduling
    - Static, Dynamic, Affinity Scheduling
    - Dynamic Affinity Scheduling
  - Discrete FGDS
    - Tabirca et al.
    - O(log p)

- 2010
  - KASS
    - Wang et al.
    - Knowledge-based Adaptive Self-Scheduling

- 2014
  - μSched
    - Kale et al.
  - ωSched
    - Fullsite
    - Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality

- 2015
  - Dynamic Partitioned Affinity Scheduling
  - Wrapped Partitioned Affinity Scheduling

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

ADAC6 | Zurich | June 20, 2018
Dynamic Loop Self-Scheduling

1966
List scheduling (Graham)
Optimal online scheduling for tasks with unknown processing times

1969
GSS-MP (Rudolph, Polychronopoulos)
Guided self-scheduling in message passing systems

1983
SS (Lusk, Overbeek)
Self-scheduling Among the first dynamic parallel-loop scheduling as optimized implementation of List Scheduling

1986
SS (Tang, Yew)
Self-scheduling Among the first dynamic parallel-loop scheduling as optimized implementation of List Scheduling

1989
GSS-MP (Rudolph, Polychronopoulos)
Guided self-scheduling in message passing systems

1991
FAC (Flynn Hummel et al., 1991, 1992)
Factoring Probabilistic modeling of task processing times and allocation delay as i.i.d.r.v., using /dh order statistics.

1993
TSS (Tzen, Ni)
Trapezoid self-scheduling

1995
FRAC (Banicescu, Hummel)
Factoring + Tiling (using SFC) N-body: PFMA Data locality and load balancing

1997
FISS, VISS (Philip, Das.)
Fixed increase SS Variable increase SS

1999
DTSS (Xu, Polychronopoulos)
Distributed TSS Extends self-scheduling schemes to heterogeneous distributed systems

1981
CSS, FSC, ESS (Kruskal, Weiss)
Fixed-size chunking with/without optimal chunk size. Found as “static:chunk” in OpenMP. Enhanced CSS assigns dependent iterations to the same processor

1985
SS (Smith)
Self-scheduling Among the first dynamic parallel-loop scheduling as optimized implementation of List Scheduling

1987
Pre-FAC (Flynn, Flynn Hummel)
Scheduling Variable-Length Parallel Subtasks Mathematical basis of FAC

1990
AGSS (Eager, Zahorjan)
Adaptive guided self-scheduling

1992
MIGSS (Wang, Wang)
Multilevel interleaved guided self-scheduling

1994
TAPER (Lucco)
Tapering strategy

1998
Tree Scheduling Decentralized, employs migration

1999
WF (Hummel et al.)
Safe-self-scheduling assigns to each processor the largest number of consecutive iterations having a cumulative execution time just exceeding the average processor workload Mix of STATIC + FAC

2001
TFSS (Chronopoulos et al.)
Trapezoid-factoring self-scheduling

2008
AWF-varients (Cariño et al.)
Use the ratios based on timings from earlier chunks to compute the processor weights for the succeeding chunks AWF-B (batched AWF) AWF-C (chunked AWF) AWF-D (coarser than AWF-B) AWF-E (coarser than AWF-C)

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

ADAC6 | Zurich | June 20, 2018

PEMPIs VRP (Laine, Midorikawa)
Performance prediction based on VRP: Vector of Relative Performances

MemBankDSL (Kandemir et al.)
Compare against SS and Tapering Impl in compiler Target embedded sys.
OpenMP has not yet adopted state of the art scheduling (beyond SS and GSS)

Why more self-scheduling?
- Risk of unexploited parallelism due to increased core counts
- Load imbalance: problem, application, system (e.g., OS preemption, migration; NUMA effects due to smaller caches / core)

Central work queue
- Facilitates a dynamic, even distribution of load among processors
- Ensures no processor remains idle while there is work to be done
- Scalability through hierarchies and distribution

Self-scheduling places the scheduling responsibility on the runtime system rather than on the operating system or the programmer
- The runtime: optimized for a specific programming model and semantics
- The operating system kernel primitives must be general enough to accommodate a variety of programming models and languages
- The programmer: not (always) a scheduling expert
Parsing the Title
“Making the Case”

✧ One in-house linear algebra kernel
✧ Four molecular dynamics codes from various OpenMP benchmark suites
✧ Non-uniformly distributed loops ⇒ Problem and algorithmic variance

<table>
<thead>
<tr>
<th>Benchmark suite: code</th>
<th>#LOC</th>
<th>#Parallel Loops</th>
<th>#Iterations</th>
<th>#Iterations Exec. Time</th>
<th>C.O.V.</th>
<th>Execution time on 1 thread</th>
<th>OpenMP Fraction</th>
<th>Not OpenMP Fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjoint convolution decreasing task: ac</td>
<td>235</td>
<td>1</td>
<td>$10^6$</td>
<td>57 %</td>
<td>591.39 s</td>
<td>99.99 %</td>
<td>0.01 %</td>
<td></td>
</tr>
<tr>
<td>OpenMP SCR: c_md</td>
<td>384</td>
<td>4</td>
<td>$16 \times 10^3$</td>
<td>57 %</td>
<td>865.31 s</td>
<td>100 %</td>
<td>0.00 %</td>
<td></td>
</tr>
<tr>
<td>RODINIA: lava.md</td>
<td>430</td>
<td>1</td>
<td>$13 \times 10^4$</td>
<td>14 %</td>
<td>5168.60 s</td>
<td>99.98 %</td>
<td>0.02 %</td>
<td></td>
</tr>
<tr>
<td>SPEC OpenMP2012: 350.md</td>
<td>3,701</td>
<td>10</td>
<td>$27 \times 10^3$</td>
<td>8700 %</td>
<td>98.57 s</td>
<td>97.19 %</td>
<td>2.81 %</td>
<td></td>
</tr>
<tr>
<td>NAS OpenMP: MG Class C</td>
<td>1,466</td>
<td>13</td>
<td>$10^1 - 10^3$</td>
<td>0-1 %</td>
<td>55.70 s</td>
<td>89.04 %</td>
<td>10.96 %</td>
<td></td>
</tr>
</tbody>
</table>
Newly added self-scheduling techniques

- **tss**: trapezoid self-scheduling 
  - **TSS** [‘93]
  - Collapses to static,chunk when first and last chunk equal #iterations/#cores

- **fac2**: practical factoring 
  - **FAC** [‘90-92]
  - Unknown mean and stdev of iteration execution times

- **wf2**: practical weighted factoring 
  - **WF** [‘96]
  - Unknown mean and stdev of iteration execution times

- **rand**: random self-scheduling 
  - **RAND**
  - Random chunk $\in \frac{\text{#iterations}}{100} \times \#\text{cores}$, $\frac{\text{#iterations}}{2} \times \#\text{cores}$, $\text{min} \geq 1$, $\text{max} \geq \text{min} + 1$

- **Usage via** `schedule(runtime)`

- **Implementation into open source OpenMP runtime**
  - LaPeSD-libGOMP https://github.com/lapesd/libgomp

100 iterations, 4 threads
Parsing the Title
“Making the Case”

miniHPC: Fully-controlled 22-node system used for research and teaching

<table>
<thead>
<tr>
<th>miniHPC node</th>
<th>Characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>2</td>
</tr>
<tr>
<td>Processor</td>
<td>Intel Xeon CPU E5-2650 v4</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.40GHz</td>
</tr>
<tr>
<td>Architecture</td>
<td>x86_64</td>
</tr>
<tr>
<td>L1D cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L1I cache</td>
<td>32KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256KB</td>
</tr>
<tr>
<td>L3 cache</td>
<td>25600KB</td>
</tr>
<tr>
<td>RAM</td>
<td>64GB</td>
</tr>
<tr>
<td>Physical CPU cores</td>
<td>20</td>
</tr>
<tr>
<td>HT CPU cores</td>
<td>40</td>
</tr>
</tbody>
</table>
Executed five benchmarks with their OpenMP loops scheduled using 20 threads with
- STATIC, SS, GSS
- TSS, FAC2, WF, RAND
- Originally: no schedule

System-induced load imbalance
- Five pinning strategies

Parallel execution time statistics (median and stdev) of 20 runs of each experiment

Does a schedule benefit a parallel loop?
Can it handle HW heterogeneity?
Parsing the Title
“Making the Case”

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

ADAC6 | Zurich | June 20, 2018
Parsing the Title
“Making the Case”

<table>
<thead>
<tr>
<th>#Iterations</th>
<th>C.O.V.</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>16×10³</td>
<td>57 %</td>
<td></td>
</tr>
</tbody>
</table>

Exec. Time: 57 %

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Iterations</th>
<th>C.O.V.</th>
<th>Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>16×10³</td>
<td>57 %</td>
<td></td>
</tr>
<tr>
<td>GSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FAC2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WF2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RAND</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Graph:**
- x-axis: Schedule clauses
- y-axis: time in seconds
- Graph shows performance comparison among different schedules.
Parsing the Title
“Making the Case”

#Iterations | C.O.V. | Exec. Time

<table>
<thead>
<tr>
<th>STATIC</th>
<th>GSS</th>
<th>TSS</th>
<th>RAC2</th>
<th>WF2</th>
<th>RAND</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>13×10⁴</td>
<td>14%</td>
<td>14%</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

lava.md
OpenMP Loop Scheduling Revisited: Making a Case for More Schedules
Parsing the Title
“Making the Case”

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

#Iterations Iterations (Class C) Exec. Time

2-10^3 0-1 %

C.O.V.

min max

time in seconds

MG

schedule clauses

PIN1 PIN2 PIN3 PIN4 PIN5
To Use or Not to Use Dynamic Loop Self-Scheduling?
No “One-Size-Fits-All”, Wide Gap Between Best and Worst

✧ Additional schedules provide benefit over existing schedules

✧ When application and system parallelism is regular, **STATIC is sufficient**

✧ When the #iterations is **too small** to generate enough work and when the #threads is **large**, then **STATIC is sufficient**

✧ When the cost of allocating loop iterations to a thread is **larger** than the cost to execute the loop iterations then **dynamic loop scheduling is not beneficial**
  ✧ Static and affinity-based methods can be used instead

✧ In the other cases
  ✧ High compute intensity
  ✧ Nested and irregular parallelism
  ✧ System-induced variabilities (e.g., OS, NUMA) **dynamic loop scheduling is needed** and **self-scheduling offers benefits over affinity- and work stealing-based methods**
So What?

Advantage

✧ The newly implemented DLS are **immediately usable** by existing programs using our non-standard prototype implementation via `schedule(runtime)`
  ✧ Numerous OpenMP production codes in active use
  ✧ Numerous multi/manycore platforms available
  ✧ [https://bitbucket.org/PatrickABuder/libgomp/src](https://bitbucket.org/PatrickABuder/libgomp/src)

Usefulness

✧ On heterogeneous platforms
  ✧ Multi/manycore CPUs
    ✧ Fat cores or faster connected cores self-schedule more frequently
    ✧ Thin cores or slower connected cores self-schedule more rarely
  ✧ Multi/manycore CPUs and accelerator cores
    ✧ `schedule(runtime)` for the CPU threads
    ✧ `dist_schedule(static,chunk)` with `schedule(runtime[,chunk])` for target teams and their threads on accelerator cores
Now What?

✧ Advocate for the inclusion of more self-scheduling techniques into the OpenMP standard or as an interface for user-defined schedulers
  ✧ To address all sources of load imbalance (problem, algorithmic, systemic) during execution
  ✧ Runtime should exploit user expert knowledge about the application
  ✧ Global OpenMP task scheduling still unaddressed

✧ Implement further state-of-the-art loop self-scheduling techniques (with feedback loops) into LLVM/Clang

✧ Extend the proof of concept beyond benchmarks into real applications
  ✧ Combine with self-scheduling in MPI layer
  ✧ PASC project SPH-EXA, www.pasc-ch.org/projects/2017-2020/sph-exa/

✧ Implement an intelligent selection mechanism among the many available options, based on previous work [Boulmier et al. 2017; Banicescu et al. 2013]