University of Basel

OpenMP Loop Scheduling Revisited: Making a Case for More Schedules

Florina M. Ciorba[1]
hpc.dmi.unibas.ch

ADAC6 Workshop Zurich, 20.06.2018

Work conducted with Christian Iwainsky[2] and Patrick Buder[1], to appear at iWomp18
[1] University of Basel, Switzerland
[2] Technische Universität Darmstadt, Germany

FNSNF Swiss National Science Foundation

HESSEN

Parsing the Title
"Scheduling"

- 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

This talk

Multiscale modeling

Macroscopic Scale (+1)
Space: 1 mm^3 - 1 km^3
Time: s-h

Mesoscopic Scale (0)
Space: 0.1-10.1 mm^3
Time: ms

Microscopic Scale (-1)
Space: 0.1-15 μm^3
Time: ns

Atomistic Scale (-2)
Space: 1-300 nm^3
Time: 0-1 ps

Electronic Scale (-3)
Space: 2-10 $Å^3$
Time: 0-1 fs

Hardware parallelism

Global Grid Scale (+2)
Parallelism: 100x → 1,000x sites
Time: s-h

Local Grid or HPC Site Scale (+1)
Parallelism: $2*10^4 \rightarrow 10^5$-10^6 nodes
Time: ms-h

Node Scale (0)
Parallelism: 2-4 → 8-16 sockets
Time: 10^9 instructions × 1 ns

Chip or Socket (MIMD) Scale (-1)
Parallelism:
8-12 → 32-256 CPU cores
5,000 → 10,000 GPU cores
40-60 → 100 Co-proc. cores
Time: 10^6 instructions × 1 ns

Core Scale (-2)
Parallelism: 1:8 scalar:vector ratio
Time: 10^3 instructions × 1 ns

Vector Scale (-3)
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 (-4)
Parallelism:
10 → less instructions (1-thread)
30 →100 instructions (multithread)
Time: several ns

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

Software parallelism

Global Batch Scale (+3)
Parallelism:
number of sites

Local Batch Scale (+2)
Parallelism:
10^0-10^7 processes × number of jobs / time period

Job Scale (+1)
Parallelism:
10^0-10^7 processes or
10^0-10^{10} threads

Process Scale (0)
Parallelism:
5,000 CPU threads
50,000 GPU threads

Thread Scale (-1)
Parallelism:
8-12→32-256 CPU threads
5,000→10,000 GPU threads
40-60→100 Co-proc. threads

SIMD Instruction Scale (-2)
Parallelism: 1 SIMD instruction

SISD Instruction (scalar) Scale (-3)
Parallelism: 1 SISD instruction

Increasing Hardware Parallelism

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

	Piz Daint @ CSCS	SUMMIT @ ORNL	TSUBAME 3.0 @ TokyoTech
CPU cores/node	1×12 (Xeon XC50) 2×18 (Xeon XC40)	2×22 (Power9)	2×14 (Xeon)
GPU cores/node	1×3,584 (CUDA P100) (XC50)	6×640 (Tensor V100) 6×5,120 (CUDA V100)	4×3,584 (CUDA P100)
Nodes	5,320 (XC50) 1,813 (XC40)	4,608	540

This talk

- 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 (SPEComp, 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

Parsing the Title

"OpenMP Loop Scheduling Revisited"

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

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

Are these schedules sufficient for all apps and systems?

Are there any other schedules not yet in OpenMP?

YES

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

Parsing the Title
“Loop Scheduling Revisited”

	Scheduling				Work Queue	Data	Optimization Goal	
	Partitioning	Assignment	Load Balancing: Ordering	Load Balancing: Timing			Explicit	Implicit
Fully static (pre-scheduling)	compilation	compilation	compilation	compilation	central	central | distributed	½ locality ½ scheduling overhead	load imbalance**
Work sharing (static allocation)	compilation | execution	compilation	compilation	execution	central	central | replicated | distributed	½ locality ½ scheduling overhead	load imbalance**
Affinity & Work stealing	compilation | execution	execution	compilation	execution	distributed	central | distributed	½ locality ½ load imbalance**	scheduling overhead
Fully dynamic (self-scheduling)	execution	execution	[compilation] execution	execution	central	central | replicated | distributed	½ scheduling overhead ½ load imbalance***	locality

load imbalance** induced by problem and algorithm
load imbalance*** induced by problem, algorithm, and system

½ one goal vs. ½ another goal — **explicit trade-off** between two optimization goals

Static Scheduling and Work Sharing

Block

Block

N iterations are divided into $\lceil N/P \rceil$ blocks
Suitable for uniformly distributed loop iterations.

Cyclic

Cyclic

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

Block-D

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

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

polyhedral.info

automatic parallelization,
data locality optimizations,
memory management optimizations,
program verification,
communication optimizations,
SIMDization,
code generation for hardware accelerators, high-level synthesis

2002

Static Workload Balance (Tabirca et al.)

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

2018+

Others?

Affinity Scheduling and Work Stealing

1992
AFS
(Markatos, LeBlanc)
Affinity Scheduling

1993
LDS
(Li et al.)
Locality-based Dynamic Scheduling

1994
AFS
(Subramanian, Eager)
Dynamic Partitioned Affinity Scheduling
Wrapped Partitioned Affinity Scheduling

SAS
(Hamidzadeh, Lilja)
Self-Adjusting Scheduling

1995
LAFS
(Wang et al.)
Localized Affinity Scheduling

MAFS
(Wang, Chang)
Modified Affinity Scheduling

CAFS
(Wang et al.)
Clustered Affinity Scheduling

1997
EA, LA, CA, GA, HA
(Yan et al.)
Adaptive affinity algorithms with **exponential adaptive**, **linearly adaptive**, **conservatively adaptive**, **greedily adaptive**, **heuristic adaptive** mechanisms
Exploit dynamic information

1998
FGDLS: FGBS, FGAFS
(Bull)
Feedback-guided dynamic loop scheduling:
Feedback guided block scheduling
Feedback guided affinity scheduling

1999
Work stealing
(Blumofe, Leiserson)
For fully strict (well-behaved) multithreaded computations

2002
Work Dealing
(Hendler, Shavit)
Low-overhead alternative to Work Stealing

Continuous FGDLS
(Tabirca et al.)
O(log p)

2003
Discrete FGDLS
(Tabirca et al.)
O(p + log p)

2006
2D-FGS, 2D-FGLS, 2D-STATIC, 2D-DYN, 2D-AFS, 2D-DAFS
(Price)
Feedback Scheduling,
Feedback Guided Scheduling,
Static,
Dynamic,
Affinity Scheduling,
Dynamic Affinity Scheduling

Discrete FGDLS
(Tabirca et al.)
O(log p)

2011
HS
(Olivier et al.)
Hierarchical Scheduling
Two levels: process and thread

2012
KASS
(Wang et al.)
Knowledge-based Adaptive Self-Scheduling

MD, AR, RP, NR
(Ahn)
MD: most-dividing
AR: all-redistribution
RP: random-polling
NR: neighbor-redistribution
Model using order statistics

-2013+
Charm++ LB
(Kale et al.)
Centralized:
RandCentLB
MetisLB
ScotchLB
GreedyLB
GreedyRefineLB
GreedyCommLB
TopoCentLB
RefineLB
RefineSwapLB
RefineCommLB
RefineTopoLB
BlockLB
RotateLB
ComboCentLB
Distributed:
NeighborLB
WSLB
DistributedLB
Hierarchical:
HybridLB
Seed:
random, neighbor
spray, workstealing

2014
μSched, νSched, fullsite
(Kale et al.)
Lightweight Scheduling for Balancing the Tradeoff Between Load Balance and Locality

2015
User-Defined Schedules
(Kale)
Static and dynamic (work stealing) scheduling

Dynamic Loop Self-Scheduling

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

1981
SS
(Smith)
Self-scheduling
Among the first dynamic parallel-loop scheduling as optimized implementation of **List Scheduling**

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

1985
CSS, FSC, ECSS
(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

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

1987
GSS
(Polychronopoulos, Kuck)
Guided self-scheduling
Dynamically adjusts (decreases) the number of iterations self-scheduled to each processor

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

1990
Pre-FAC
(Flynn, Flynn Hummel)
Scheduling Variable-Length Parallel Subtasks
Mathematical basis of FAC

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 *P*th order statistics.

1992
AGSS
(Eager, Zahorjan)
Adaptive guided self-scheduling

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

TAPER
(Lucco)
Tapering strategy

1993
TSS
(Tzen, Ni)
Trapezoid self-scheduling

1994
SSS
(Saletore, Lewis)
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

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

MSS
(Jung et al.)
Multithreaded self-scheduling
Distributed memory systems
Multithreads the iterations of a chunk execution into threads

LLPC
(Yue, Lilja)
Loop-level process control
Uses current system load to determine the upper limit on the number of processes the application can create for that parallel section

1996
GDCS
(Lee et al.)
Global Distributed Control Scheduling
Decentralize the scheduling

TS
(Kim, Purtilo)
Tree Scheduling
Decentralized, employs migration

WF
(Hummel et al.)
Weighted Factoring
Weights chunks with processor speeds

BOLD
(Hagerup)
Adapts chunk size at runtime based on chunk execution time

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

1998
BAL
(Bast)
Challenges the assumption of i.i.d.r.v. task processing times
Proposes variance estimator and upper and lower bounds for task processing times

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

2000
AF
(Banicescu, Liu)
Adaptive Factoring
Decentralize the scheduling

AWF
(Banicescu et al.)
Adaptive weighted factoring
Adapts processor weights after a time-step

AHS
(Fann et al.)
Adaptive hybrid scheduling
Employs static and dynamic scheduling based on manually chosen probabilities to fetch / not fetch iterations

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

2007
PEMPIs VRP
(Laine, Midorikawa)
Performance prediction based on VRP: Vector of Relative Performances

MemBankDLS
(Kandemir et al.)
Compare against SS and Tapering
Implem. in compiler
Target embedded sys.

2008
AWF-variants
(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)

Loop Scheduling in Compilers and Runtime Systems

1987

Implements FSC
(Butler et al.)
Portable Programs for Parallel Processors

1988

TUS Implements FSC
(Thomas, Crowther)
The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors

1989

Implements AGSS
(Eager, Zahorjan)
Adaptive Guided Self-Scheduling
Proposed a wrapped assignment of iterations to rectify a shortcoming of GSS

1991

Implements FSC
KSR Presto runtime library

Runtime supervisor (RP3)
(Hummel, Schonberg)
Central work queue singly-linked list

1996

Fixed block SS
(Durand et al.)
Impact of memory contention on dynamic scheduling on NUMA multiprocessors for fixed block SS (or CSS) and for decreasing block SS (i.e., factoring-based SS)

1998

Adaptive LB
(Ioannidis, Dwarkadas)
TreadMarks and SUIF compiler

Hectiling
(Russ et al.)
Fractiling + Hector = Hectiling
Implements FRAC in the HECTOR runtime

2000

IPLS
(Fann et al.)
Intelligent parallel loop scheduling: Knowledge-based techniques to select appropriate loop-scheduling algorithms according to loop behaviors and system states

2003

DLS API vs. OpenMP
(Govindaswamy)
An API for Adaptive loop scheduling in shared address space architectures

PREMA
(Balasubramaniam)
Parallel Competitive Runtime Environment for Multicomputer Applications

Affinity Sched. in IBM OpenMP RT
(Ayguadé et al.)
Implements SAS, AFS, FGDLS into IBM's OpenMP RT

2004

Adaptive OpenMP
(Zhang et al.)
Implement **AFS**, **TSS**
Compare against OpenMP's static, dynamic, guided
HT and SMT architectures
Two level scheduling
Uses SPECComp

2005

Runtime Empirical Selection
(Zhang, Voss)
Extends Adaptive OpenMP
Uses NASomp, SPECComp
Static, dynamic, guided, AFS, TSS, do not react to OpenMP applications executing on SMP systems with SMT nodes

2008

OpenMP Task Scheduling
(Duran et al.)
Evaluation of OpenMP Task Scheduling Strategies

2010

ForestGOMP
(Broquedis et al.)
An OpenMP RT extended by "*An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines*" (Durand, et al., 2013)
Introduces Adaptive Loop Scheduling (ALS)
Employs work stealing
First implementation of work stealing in OpenMP

2011

autopin
(Klug et al.)
Automated Optimization of Thread-to-Core Pinning on Multicore Systems
Uses HW performance counters to automatically detect and apply the best binding between threads and processor cores in a shared memory system

2012

Automatic OpenMP Loop Scheduling
(Thoman et al.)
A Combined Compiler and Runtime Approach
Fully automatic loop scheduling policy: adapts to both application characteristics and current system external load
Forwards static code analysis to a runtime system
Employs Insieme compiler
Compares with LIBGOMP 4.5.3: static, guided, dynamic
Test on 3 custom kernels and NAS kernels

2015

User-Defined Schedules
(Kale, Gropp)
OpenMP Interface to define own user schedules
Divide application into static + dynamic fractions for scheduling
Employ work stealing

JIT LB
(Cammarota et al.)
Proposes a new run-time technique
1.Profiles iteration space (use PAPI)
2.Partitions it into non-equal chunks with equal exec. times
3.Attempts to schedule them or finds the dynamic schedule most suitable for particular instance of loop and architecture
4.Targets irregular loops
Implementation in LIBGOMP of the new technique + FAC and TSS
Experiments with 4 to 8 threads

2016

SRR
(Penna et al.)
Smart Round-Robin
Assessing the Performance of the SRR Loop Scheduler with Irregular Workloads
Workload-aware scheduling
Implementation in LIBGOMP
Static allocation, dynamic execution

2017

DLS+OpenMP
(Buder)
Evaluation and Analysis of Dynamic Loop Scheduling in OpenMP
STATIC, SS, FSC, FAC, TAPER, TSS, WF, BOLD in LibGOMP
*Preliminary version of this talk

BinLPT
(Penna et al.)
A Novel Workload-Aware Loop Scheduler for Irregular Parallel Loops
Static allocation, dynamic execution

Parsing the Title
"More Schedules"

- 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

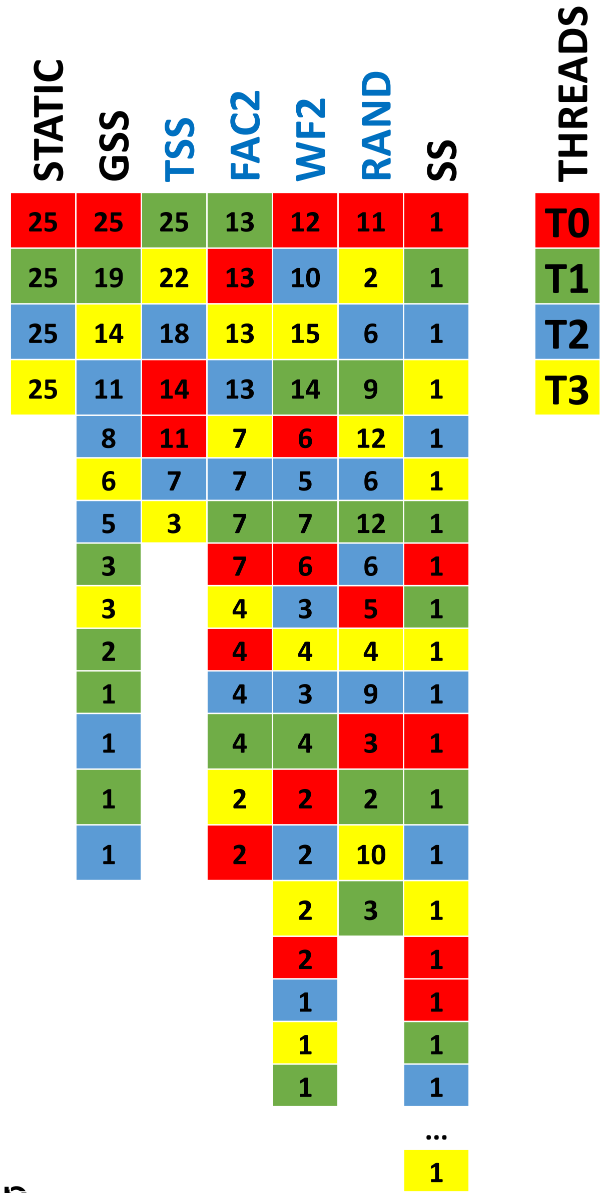
Parsing the Title
"Making the Case"

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

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

Parsing the Title
“Making the Case”

- 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` ∈ [#iterations/100×#cores, #iterations/2×#cores], min ≥ 1, max ≥ 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

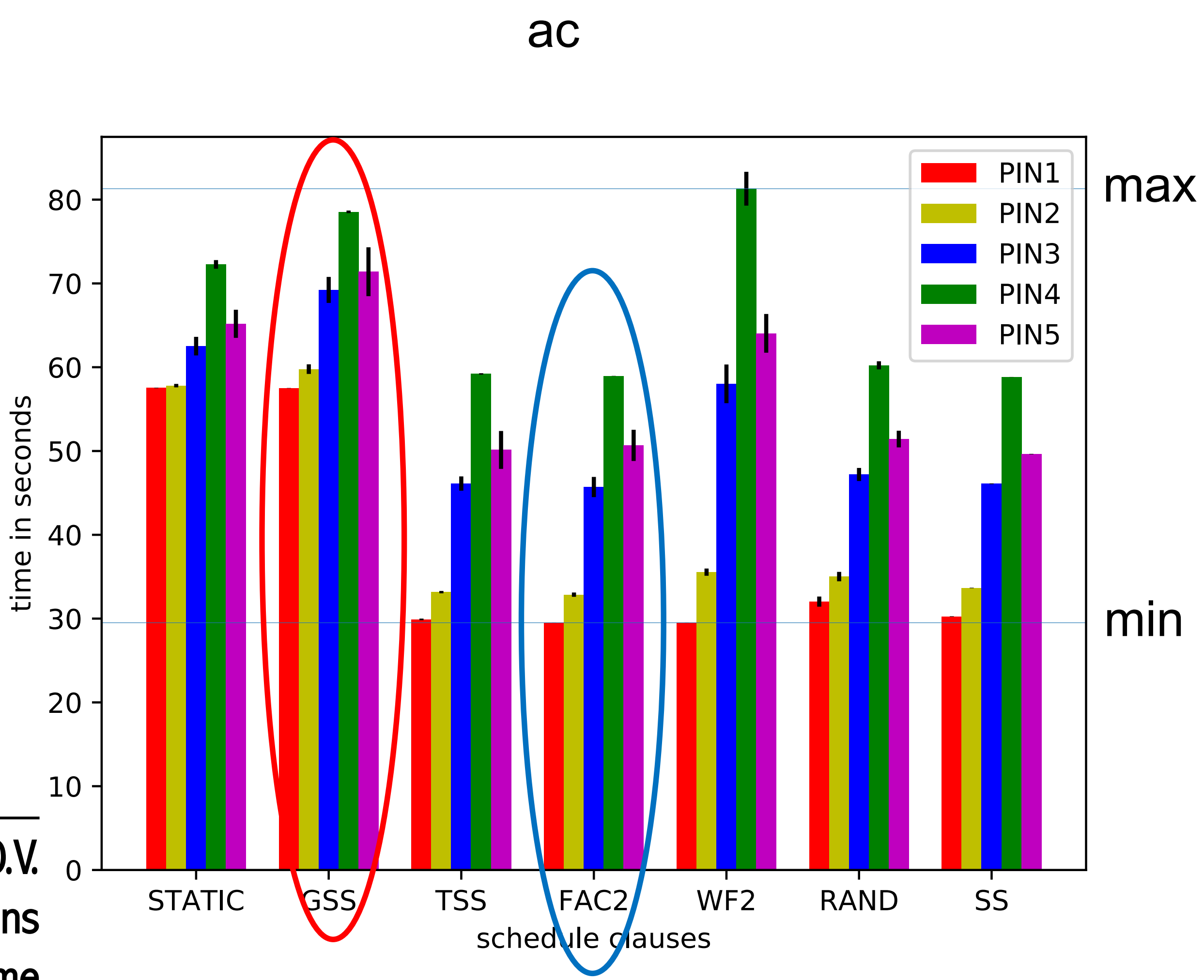
miniHPC node	Characteristic
Sockets	2
Processor	Intel Xeon CPU E5-2650 v4
Clock speed	2.40GHz
Architecture	x86_64
L1D cache	32KB
L1I cache	32KB
L2 cache	256KB
L3 cache	25600KB
RAM	64GB
Physical CPU cores	20
HT CPU cores	40

Parsing the Title
"Making the Case"

- 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?

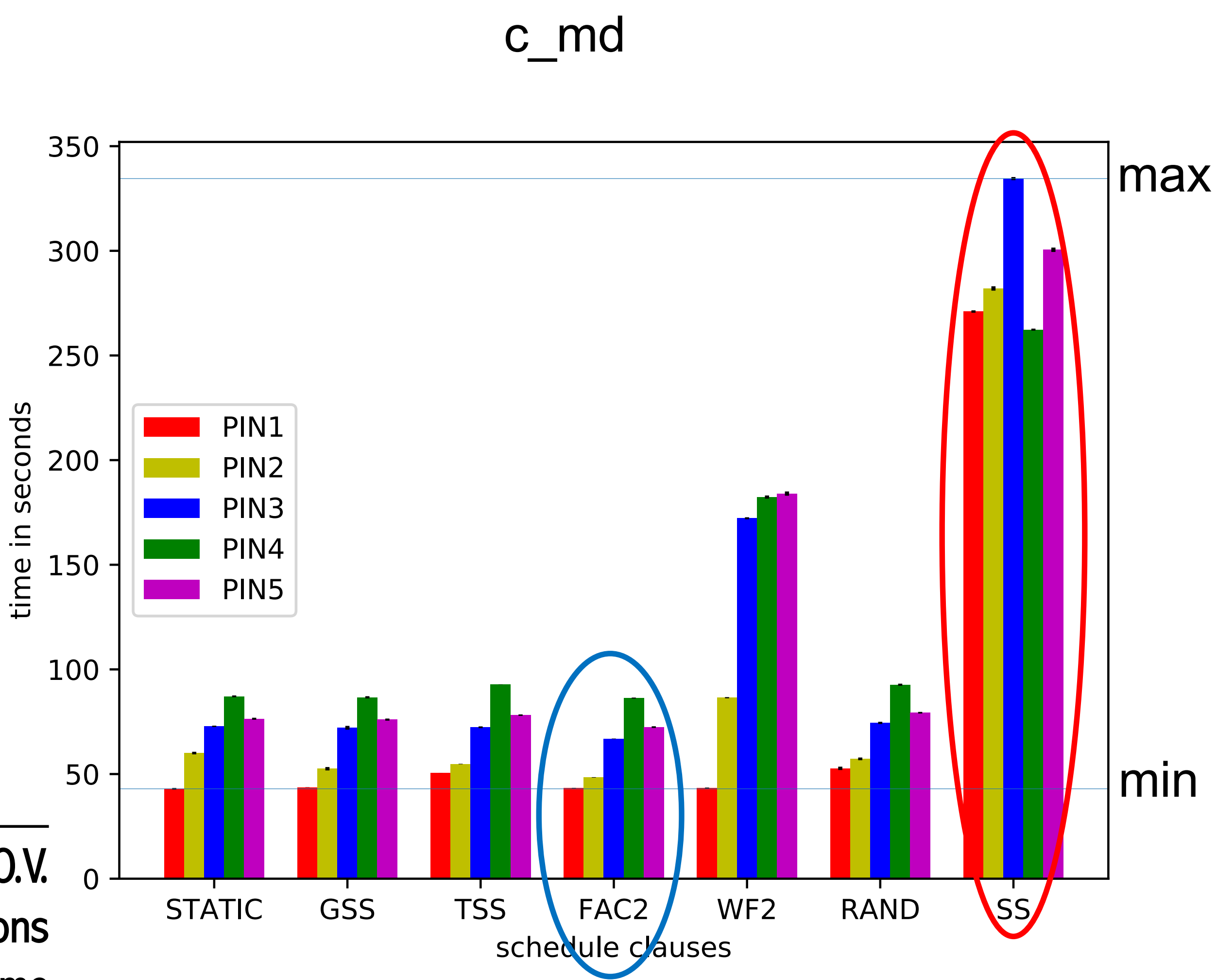
	SOCKET 0										SOCKET 1									
CORES	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19
PIN1	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19
PIN2	T0	T1	T2	T3	T4	T5	T6	T7	T8	×	T9	T10	T11	T12	T13	T14	T15	T16	T17	×
		T18										T19								
PIN3	T0	T1	T2	T3	T4	T5	T6	×	×	×	T7	T8	T9	T10	T11	T12	×	×	×	×
		T13	T14	T15	T16							T17	T18	T19						
PIN4	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	×	×	×	×	×	×	×	×
		T12	T13			T14		T15			T16									
		T17				T18														
		T19																		
PIN5	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	×	×	×	×	×	×	×	×	×	×
		T10	T11		T12	T13	T14													
		T15			T16	T17														
		T18			T19															
CORE WEIGHTS				1		0.5		0.33		0.25										

Parsing the Title
"Making the Case"



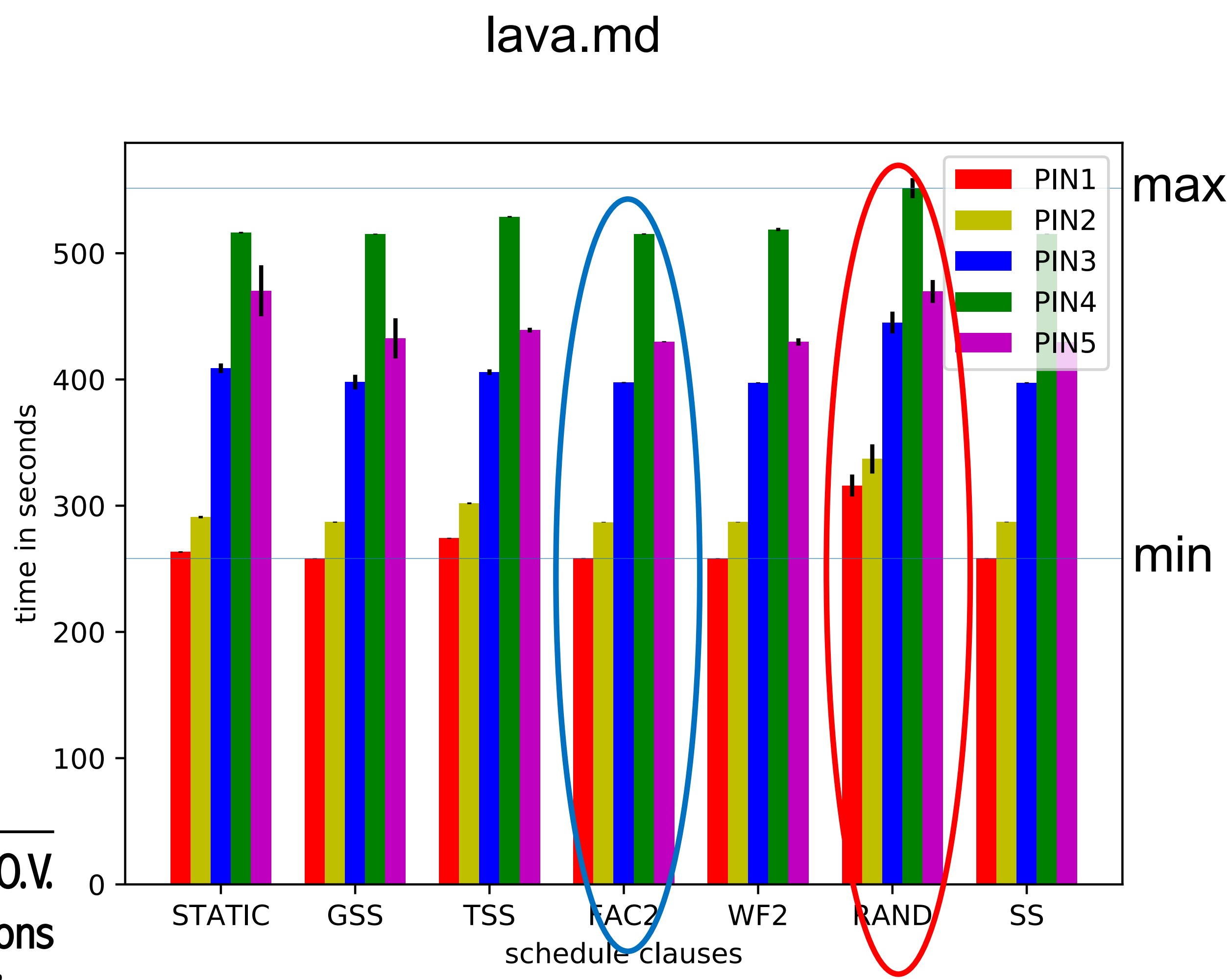
#Iterations	C.O.V. Iterations Exec. Time
10^6	57 %

Parsing the Title
"Making the Case"



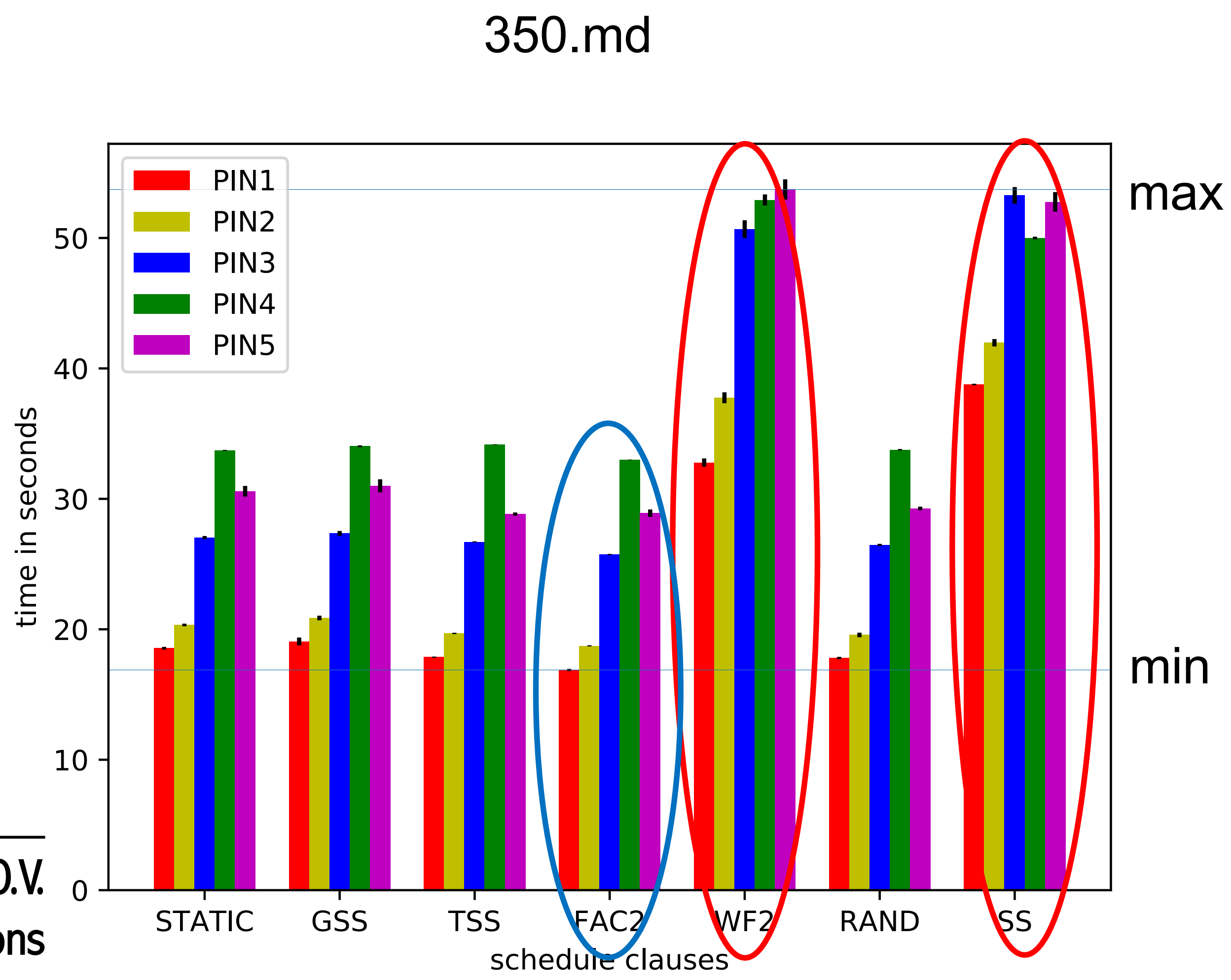
#Iterations	C.O.V. Iterations Exec. Time
16×10^3	57 %

Parsing the Title
"Making the Case"



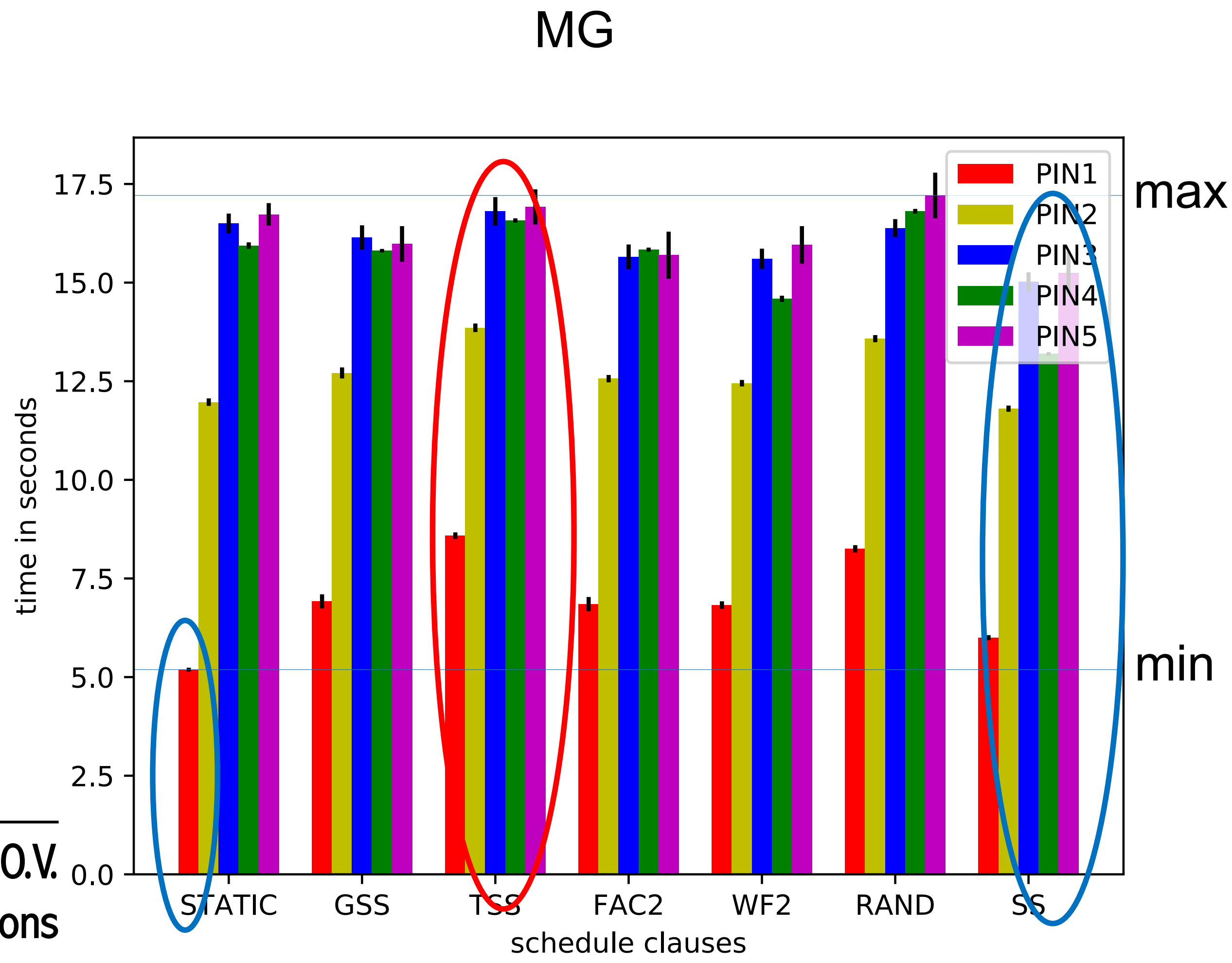
#Iterations	C.O.V. Iterations Exec. Time
13×10^4	14 %

Parsing the Title
"Making the Case"



#Iterations	C.O.V. Iterations Exec. Time
27×10^3	8700 %

Parsing the Title
"Making the Case"



#Iterations (Class C)	C.O.V. Iterations Exec. Time
$2\text{-}10^3$	0-1 %

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

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?

Thank you for your attention!

- 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]