Heterogeneous Computing for Scientific Applications

Marc Gonzalez Tallada
Associate professor
Computer Architecture Department
Technical University of Catalonia

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Outline

- Analysis, parallelism characterization of NAS-MZ suite of benchmarks
- NPB-MZ Hybrid Design and Implementation
- Work distribution schemes for hybrid executions
- Evaluation Results

NPB-MZ Parallel Benchmarks

- Data Structures and Input Size
- Computation Structure
- Sources of Parallelism

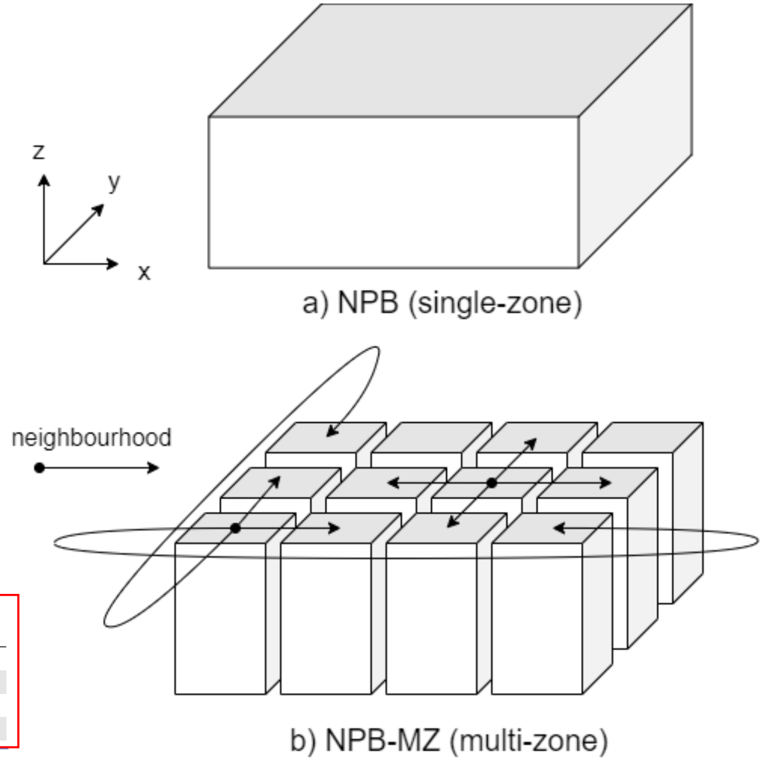
Data Structures and Input Size

- Data Structures
  - 3D mesh organized in “zones”
    - A zone is a set of multidimensional matrices
- Input Size

  size increases

  - Different classes: A, B, C, D, E
  - Number of zones: from 16 to 4096
- Iterative solvers
  - From 250 to 500 iterations

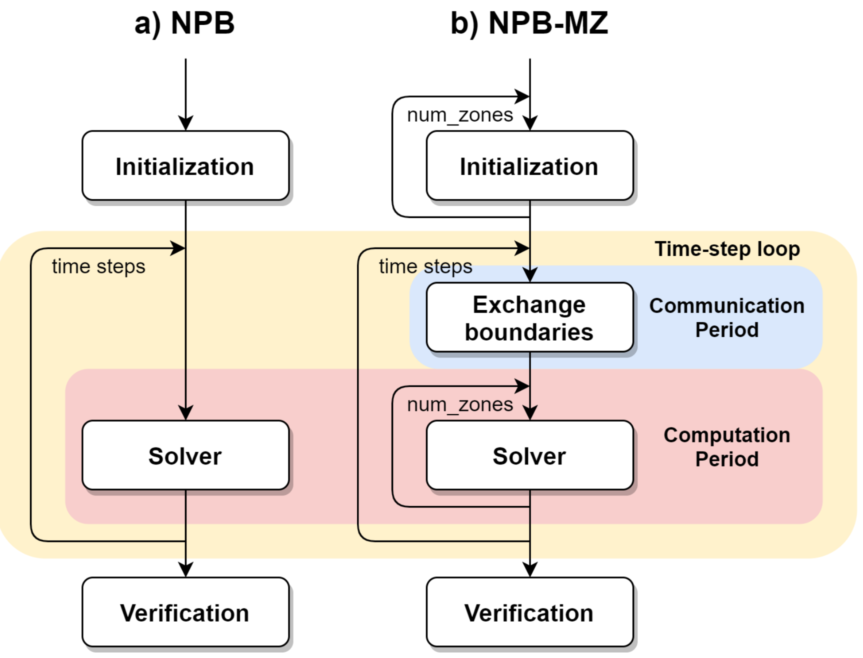
Input Class	3D volume x × y × z (points)	Memory (GB)	Num. zones (x × y) LU	Num. zones (x × y) SP & BT	Zone size (points per zone) LU	Zone size SP	Zone size BT	Time steps LU	Time steps SP	Time steps BT
B	304 × 208 × 17	≈ 0.2	4 × 4	8 × 8	67 184	16 786	from 2 992 to 59 976	250	400	200
C	480 × 320 × 28	≈ 0.8	4 × 4	16 × 16	268 800	16 800	from 2 912 to 60 648	250	400	200
D	1 632 × 1 216 × 34	≈ 13	4 × 4	32 × 32	4 217 088	65 892	from 11 968 to 243 236	300	500	250
E	4 224 × 3 456 × 92	≈ 250	4 × 4	64 × 64	83 939 328	327 888	from 59 248 to 1 203 452	300	500	250



a) NPB (single-zone)
b) NPB-MZ (multi-zone)

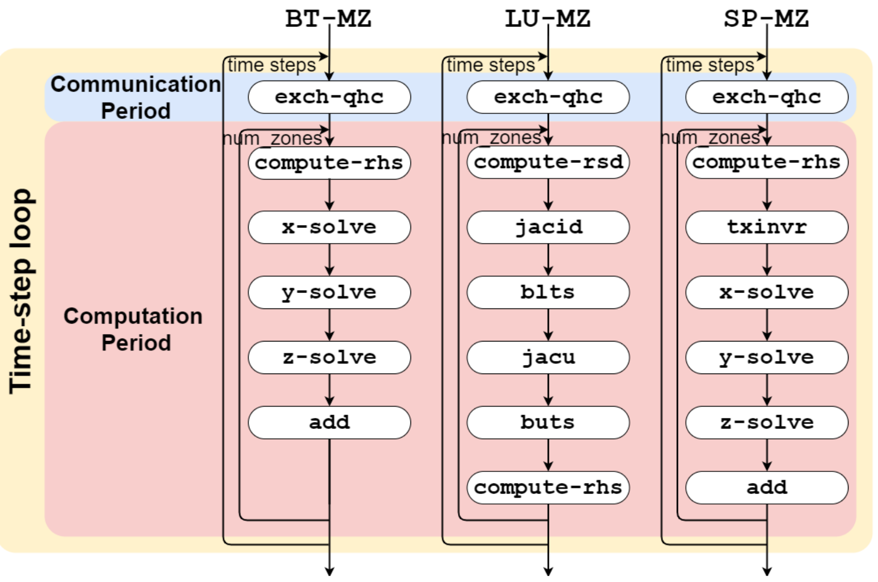
Computational Periods

- For all benchmarks:
  - Initialization
    - Several stages
  - Computation Period
    - Solver composed of several stages
    - Applied to each zone
  - Communication Period
    - Exchange of border values between zones
    - One single stage
  - Verification



Stages within the Compute Period and Communication Period

- Three benchmarks:
  - BT-MZ
    - “Many zones with small, medium and large sizes”
  - SP-MZ
    - “Many zones with small sizes and all of them equal”
  - LU-MZ
    - “Few zones and with very large size”
  - Computation Period
    - Composed of different stages per each benchmark
  - Communication Period
    - Single stage and common to each benchmark



Computation Period(I): Inter-Zone Parallelism

- Traversal of zones
  - Single loop that runs over the set of zones
  - Zones are identified
    - Number from 0 up to the NUM-ZONES-1
  - Several stages applied per each zone
    - Different in BT-MZ, SP-MZ and LU-MZ

- Example with SP-MZ:

```
for (t=0; t<ITERS; t++) {

  exch-qbc();

  for (zone = 0; zone < NUM-ZONES; zone++) {

    compute-rhs(zone, ...);
    txinvr(zone, ...);
    x-zolve(zone, ...);
    y-solve(zone, ...);
    z-solve(zone, ...);
    add(zone, ...);
  }
}
```

Inter-zone parallelism

Computation Period(II): Intra-Zone Parallelism

- Processing of one stage on one zone
  - Sequence of several loop nests
  - For CPU execution
    - Loop nest is parallelized (e.g.: OpenMP)
  - For GPU execution
    - Loop nest is transformed to CUDA kernel

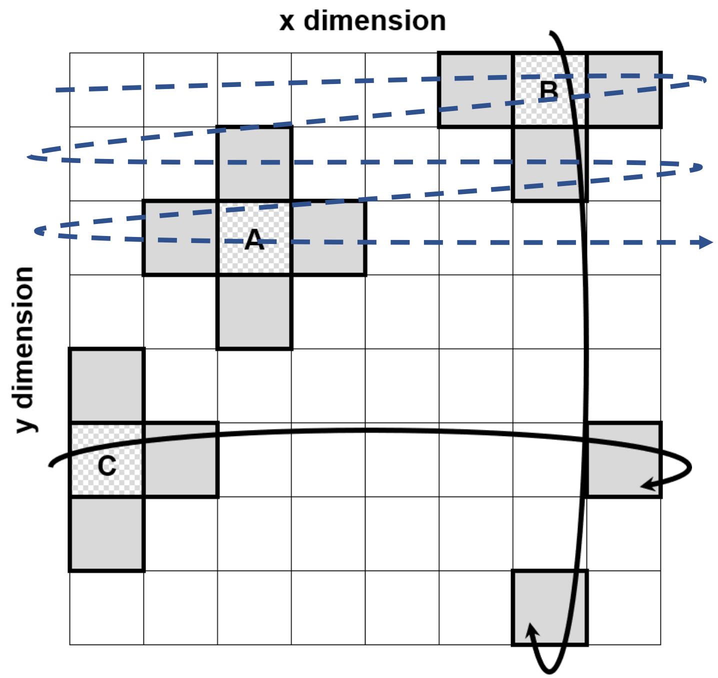
- Example: BT-MZ, `x-solve` stage

Intra-zone parallelism

```
void x-solve(unsigned int zone, ...) {
  for (k=0; k<z-dim; k++)
    for (j=0; j<y-dim; j++)
      for (i=0; i<x-dim; i++)
        /* Some matrix-based
           computations */

  /* Several computations in the
     form of loop nests */

}
```

Communication Period: Zone Adjacency

- Adjacent zones
  - 4 neighbors
    - north
    - south
    - east
    - west
- Exchange border values between adjacent zones



Communication Period: Intra-Zone Parallelism

- Processing of zone borders

```
for (zone=0; zone<NUM-ZONES; zone++) {
  east-zone = adjacency-east[zone]
  north-zone = adjacency-north[zone];

  copy-face(tmpEast, mesh[east-zone]);
  copy-face(tmpNorth, mesh[north-zone]);

  compute-border(mesh[zone], tmpEast, tmpNort

  copy-face(mesh[east-zone], tmpEast);
  copy-face(mesh[north-zone], tmpNorth);
}
```

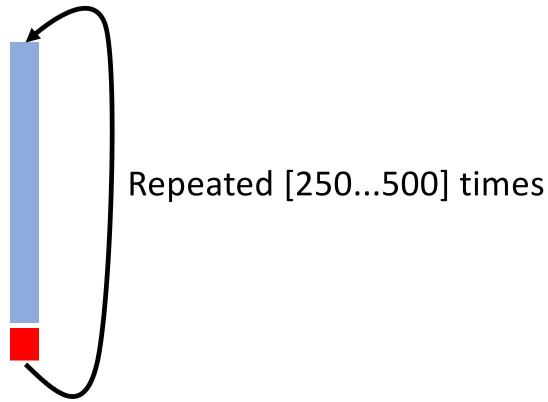
```
void compute-border(...) {

  for (k=0; k<z-dim; k++)
    for (j=0; j<y-dim; j++)
      for (i=0; i<x-dim; i++)
        /* Some matrix-based
           computations */
```

```
void copy-face(...) {

  for (k=0; k<z-dim; k++)
    for (j=0; j<y-dim; j++)
        /* Some matrix-based
           computations */
```

Summary: Computation and Communication Periods, Sources of Parallelism

- Two levels of parallelism:
  - Coarse grain parallelism: INTER-ZONE parallelism
  - Fine grain parallelism : INTRA-ZONE parallelism

- Parallelism is repeated many times
  - Computation Period
    - INTER-ZONE: mapped to CPUs **and** GPUs
    - INTRA-ZONE: mapped to CPUs **or** GPUs
  - Communication Period
    - INTRA-ZONE: mapped to CPUs **or** GPUs



Outline

- Analysis, parallelism characterization of NAS-MZ suite of benchmarks
- NPB-MZ Hybrid Design and Implementation
- Work distribution schemes for hybrid executions
- Evaluation Results

NPB-MZ Hybrid Design and Implementation

- Hybrid Execution Model: OpenMP+CUDA
  - Computing Unit Abstraction
    - `libCU-rtl`
  - Address Space Abstraction
    - `libAS-rtl`
  - Work Distribution Schemes
    - `libSCHEDULING-rtl`
- NPB-MZ Hybrid Parallelization

Computing Units (I)

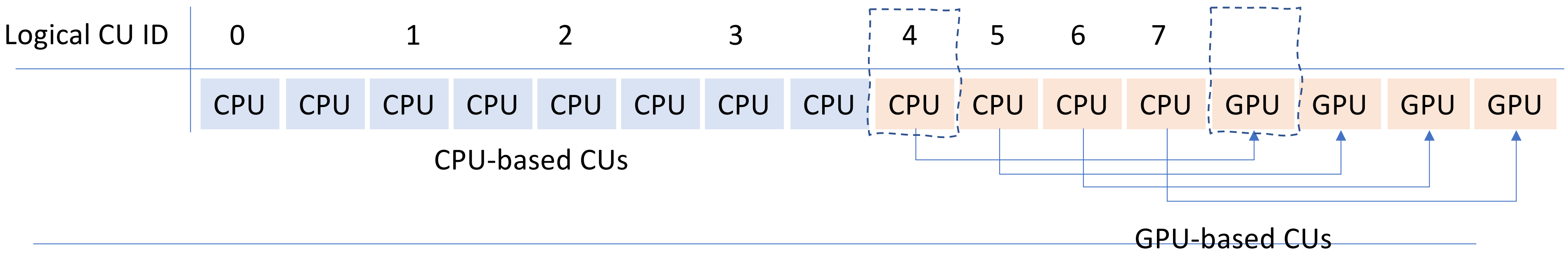
- Computing Units (CUs)
  - Logical view of all available CPUs and GPUs
  - A CU can be of type CPU or GPU
    - If CU type is of `CPU`
      - CU maps onto **one or more** physical `<CPU>`
    - If CU type is of `GPU`
      - CU maps to **one pair** of physical `<CPU,GPU>`
        - The CPU controls the GPU

- Computing Units (CUs)
  - NCU = number of available CUs
  - CUs are numbered from 0 up to the NCU-1
    - NCPU = Number of CPU-based CUs
    - NGPU = Number of GPU-based CUs
    - `[0 .. NCPU-1]` correspond to CPU-based CUs
    - `[NCPU .. NCU-1]` correspond to GPU-based CUs
- CUs and Threads
  - We run as many threads as the number of NCU

Computing Units (II)

- Example:
  - Execution with 8 CUs = 4 x CPU-based CUs + 4 GPU-based CUs
    - CPU-based CU = 2 x CPUs
    - GPU-based CU = CPU + GPU

    => 12 physical CPUS + 4 physical GPUS
  - 12 threads mapped onto 12 physical CPUs

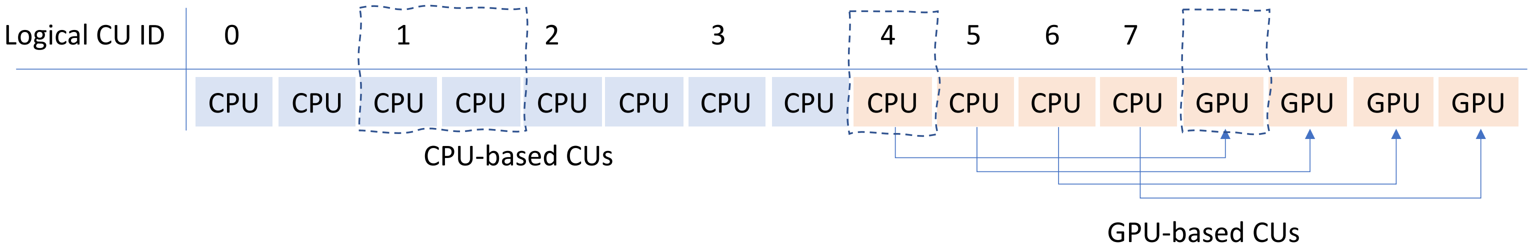


Hybrid Execution Model: OpenMP + CUDA

- How do we introduce the abstraction of a CU in OpenMP+CUDA?
  - Implement runtime support to bridge the two execution models
    - `libCU-rtl`
- Parallelism for CUs is created through `omp parallel` construct
  - 1 CPU-based CU: 1 OpenMP thread
  - 1 GPU-based CU: 1 OpenMP thread that controls a device
- CUs are formed through `OMP_PLACES`
- Name spaces for CU, OpenMP threads and CUDA devices
  - Logical CU id ⇔ OpenMP Thread id
    - `OpenMP Thread id = CU id`
  - Logical CU id ⇔ CUDA Device id
    - `CUDA Device id = CU id – NCPU`

CUs definition and OpenMP thread places

- Example
  - `OMP_PLACES = “{0, 1}, {2, 3}, {4, 5}, {6, 7}, {8}, {9}, {10}, {11}”`
  - `OMP_PROC_BIND=TRUE`
  - NCPU = 4 with 2 inner CPUs
  - NGPU = 4
  - Logical CU ids match OpenMP logical thread ids



Runtime Support for CUs

- Small set of runtime primitives
  - RUNTIME::gpus
  - RUNTIME::cpus
  - RUNTIME::isGPU()
  - RUNTIME::isCPU()
  - RUNTIME::GPU()
  - RUNTIME::CPU()
  - RUNTIME::innerCPUs()
  - RUNTIME::getTask()
  - RUNTIME::getCU()
  - RUNTIME::setCU()
  - RUNTIME::commitTask()
  - RUNTIME::executeTask()
  - RUNTIME::synchronize()

Number of GPU-based CUs and CPU-based CUs

Query if CU is mapped onto CPU or GPU

Get a CPU/GPU id in OpenMP/CUDA

Get a number of inner threads for a CPU-based CU

Get a task mapped to the current CU

Get/set the CU logical ID

Start/end of the execution of an assigned task

Synchronize with corresponding device if CU is mapped to a GPU

Code Transformation for CU support

CU parallelism

```
#pragma omp parallel \
              num_threads(RUNTIME::cpus+RUNTIME::gpus)
{
  unsigned int task;
  task = RUNTIME::getTask();
  while (task!=NO_TASK) {

    unsigned int CU = RUNTIME::getCU();

    RUNTIME::executeTask()

    /* COMPUTATIONS */

    if (RUNTIME::isGPU(CU)) RUNTIME::synchronize();

    RUNTIME::commitTask();

    task = RUNTIME::getTask();
  } // while
} // parallel
```

```
OMP_PROC_BIND=TRUE
OMP_PLACES = "{0, 1},{2, 3},{4, 5},{6, 7},
              {8},   {9},   {10},  {11}"
```

CU computations

```
  unsigned int CU = RUNTIME::getCU();
  if (RUNTIME::isCPU(CU)) {
    NT = RUNTIME::innerCPUs();
#pragma parallel num_threads(NT) proc_bind(master)
{
    /* COMPUTATIONS */

} // parallel

  }
  else if (RUNTIME::isGPU(CU)) {
    cudaSetDevice(RUNTIME::GPU(CU));
    some-GPU-kernel-1<<<grid, block, shared>>>(...);
  }
```

Distributed Address Space: OpenMP + CUDA

- How do we introduce the abstraction of a AS in OpenMP+CUDA?
  - Implement runtime support to bridge the two execution models
    - `libAS-rtl`
- Several Address Spaces (AS)
  - Host
  - Devices
- Name space for AS
  - `[0...NCPU-1]` => HOST
  - `[NCPU...NGPU-1]` => DEVICES
- Monitor data placement
  - Maintain a map of data structures and AS

Runtime Support for Data Placement

- Small set of runtime primitives
  - PLACEMENT::getAS()
  - PLACEMENT::alloc()
  - PLACEMENT::migrate()
  - PLACEMENT::copy()
  - PLACEMENT::isGPU()
  - PLACEMENT::isCPU()

Determine AS for a memory region (e.g.: a zone)

Memory allocation (e.g.: a zone) in an AS

Migrate a memory (e.g.: a zone) to an AS

Generate a copy of a zone to an AS

Query if memory (e.g.: a zone) is mapped onto CPU or GPU

Work Distribution: OpenMP + CUDA

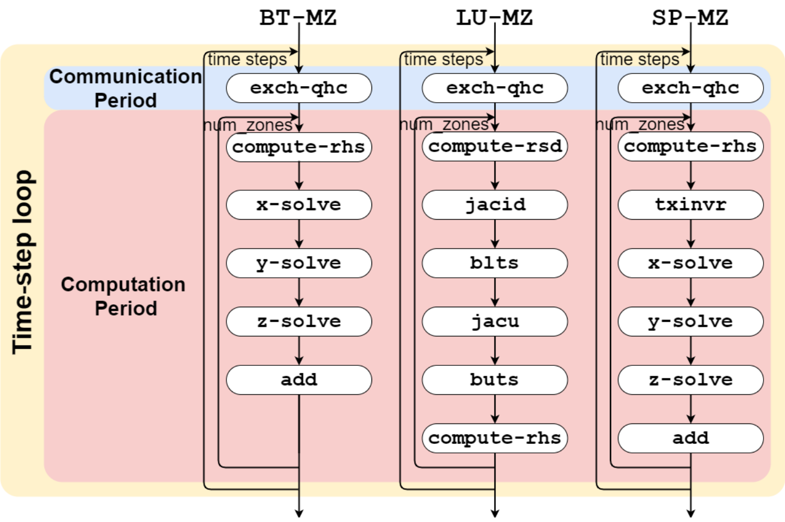
- OpenMP work distribution schemes are not suitable
  - TASKS: you do not have control over which thread will execute a task
  - LOOPs: available loop schedulers are not suitable, not designed for heterogeneity
    - STATIC, DYNAMIC, GUIDED
- Implement a runtime support to bridge the two execution models at the application level
  - libSCHEDULING-rtl

Runtime Support for Scheduling

- Assume tasks are numbered from `[0...`N_{tasks}`-1]`
  - Inputs for schedulings
    - N_{tasks}
    - `NCPU`
    - `NGPU`
- SCHEDULING-rtl
  - SCHEDULING::Static()
  - SCHEDULING::Dynamic()
  - SCHEDULING::pcf-Static()
  - SCHEDULING::pcf-Guided()
  - SCHEDULING::ClusteredGuided()

Computation Period: Code Transformation

- INTER-ZONE parallelism



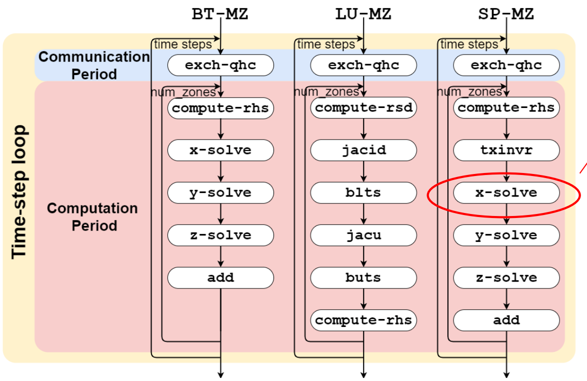
- Example: SP-MZ

```
   SCHEDULE::Static(NUM_ZONES, RUNTIME::cpus, RUNTIME::gpus);
#pragma omp parallel \
            num_threads(RUNTIME::cpus+RUNTIME::gpus)
{
  unsigned int task;
  task = RUNTIME::getTask();
  while (task!=NO_TASK) {
    unsigned int zone = task;

    unsigned int CU = RUNTIME::getCU();
    if (PLACEMENT::getAS(mesh[zone])!=CU) PLACEMENT::migrate(mesh[zone], CU);

    RUNTIME::executeTask();

    compute-rhs(zone, ...);
    txinvr(zone, ...);
    x-zolve(zone, ...);
    y-solve(zone, ...);
    z-solve(zone, ...);
    add(zone, ...);

    if (RUNTIME::isGPU(CU)) RUNTIME::synchronize();

    RUNTIME::commitTask();
    task = RUNTIME::getTask();
  } // while
} // parallel
```

OMP_PROC_BIND=TRUE

Computation Period: Code Transformation

- INTRA-ZONE parallelism




```
void x-solve(unsigned int zone, ...) {
  unsigned int CU = RUNTIME::getCU();
  if (RUNTIME::isCPU(CU)) {
    NT = RUNTIME::innerCPUs();
#pragma parallel for schedule(static)\
        num_threads(NT) proc_bind(master)
    for (k=0; k<z-dim; k++)
      for (j=0; j<y-dim; j++)
        for (i=0; i<x-dim; i++)
          /* Some matrix-based
             computations */
  }
  else if (RUNTIME::isGPU(CU)){
    x-xolve-kernel-1<<<grid, block, shared>>>(zone,...);
  }

  /* Several other computations computations in the
     previous form */

}
```

Communication Period: Border Processing

```
for (zone=0; zone<NUM-ZONES; zone++) {
  east-zone = adjacency-east[zone]
  north-zone = adjacency-north[zone];
  east-AS = PLACEMENT::getAS(mesh[east-zone]]);
  north-AS = PLACEMENT::getAS(mesh[north-zone]);
  zone-AS = PLACEMENT::getAS(mesh[zone]);

  PLACEMENT::pack-face(tmpEast, east-AS, mesh[east-zone]);
  PLACEMENT::pack-face(tmpNorth, north-AS, mesh[north-zone]);
  if (PLACEMENT::isGPU(zone-AS)) {
    CUDA::setDevice(RUNTIME::GPU(zone-AS)):
    gpu-compute-border<<< grid, block, shared>>>(mesh[zone],
                                                 tmpEast,
                                                 tmpNorth);
  else if (PLACEMENT::isCPU(zone-AS)) {
    cpu-compute-border(mesh[zone], tmpEast, tmpNorth);
  }
  PLACEMENT::unpack-face(tmpEast, east-AS, mesh[east-zone]);
  PLACEMENT::unpack-face(tmpNorth, north-AS, mesh[north-zone]);
}
```

Communication Period: Data Packing

```
void PLACEMENT::pack-face(buffer, zoneAS, mesh-zone) {
  myAS = PLACEMENT::getAS();
  if (myAS!=zoneAS) {
    // remote packing of border data and transfer to myAS
     if (PLACEMENT::isGPU(zoneAS) {
       CUDA::setDevice(RUNTIME::GPU(zone-AS));
       gpu-pack-face<<<grid, block, shared>>>(...);
       CUDA::cudaMemCpy(buffer, ..., cudaDeviceToHost);
     } else if (PLACEMENT::isCPU(zoneAS){
       cpu-pack-face(...);
     }
  } else {
    // pack border data
  }
}
void PLACEMENT::unpack-face(buffer, zoneAS, mesh-zone);
```

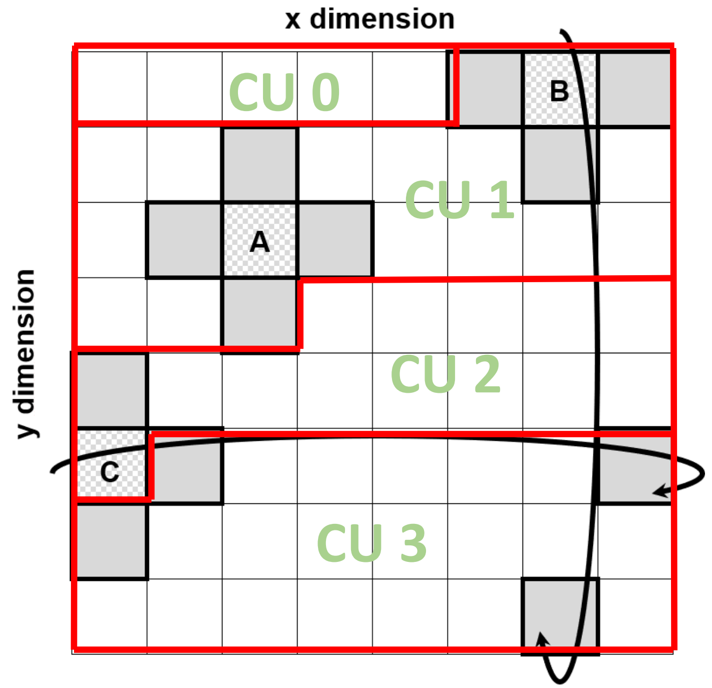
```
void cpu-pack-face(buffer, mesh-zone);
void cpu-unpack-face(buffer, mesh-zone);
```

```
__global__ gpu-pack-face(buffer, mesh-zone);
__global__ gpu-unpack-face(buffer, mesh-zone);
```

- In a hybrid execution the **CUDA::cudaMemCpy** is the source of overhead

Communication Period: Data placement and Communication

- Trade off between the number of data transfers and data placement
  - NCU =4
  - Zone B
    - Assigned to a CU where 1 adjacent zone is assigned to other CUs
  - Zone A
    - Assigned to a CU where all 4 adjacent zones are assigned to the same CU
  - Zone C
    - Assigned to a CU where 3 adjacent zones are assigned to other CUs



Outline

- Analysis, parallelism characterization of NAS-MZ suite of benchmarks
- NPB-MZ Hybrid Design and Implementation
- Work distribution schemes for hybrid executions
- Evaluation Results

Work Distribution Schemes For OpenMP+CUDA

- Baseline Schedulers
  - STATIC
  - DYNAMIC
- Performance Factor Conversion (PCF) Schedulers
  - PCF-STATIC
  - PCF-GUIDED
- Self Adaptive Schedulers
  - CLUSTERED GUIDED

STATIC scheduler

- Tasks are identified from 0 up to the number of tasks -1
- `NTASKS = NUM_ZONES;`
- `CU-WORK = NTASKS/NCU;`
- If remaining tasks, those are assigned to lower CUs within the numbering
  - CPUs first, then the GPUs
- `<start,end>:` identifies the set of tasks assigned to a CU
  - `start` = initial task
  - `end` = final task
- Maximizes adjacency, so minimizes communications

DYNAMIC scheduler

- Fast CUs (e.g.: GPUs) will tend to get more work
- Slow CUs (e.g.: CPUs) will tend to get less work
- `CHUNK = 1;`
- Memorizes TASK-CU mapping
  - Avoids memory migration overheads
- Breaks adjacency, so will tend to increase the communications

PCF-STATIC scheduler

- Variant of STATIC scheduler
- Based on a Performance Conversion Factor (PCF)
  - PCF = relation between the computational power of CPU-based CUs and GPU-based CUs
  - Example
    - PCF = 2, means that GPU-based CUs are twice faster than CPU-based CUs
    - PCF = 1, means both CPU and GPU based CUs have same computation power

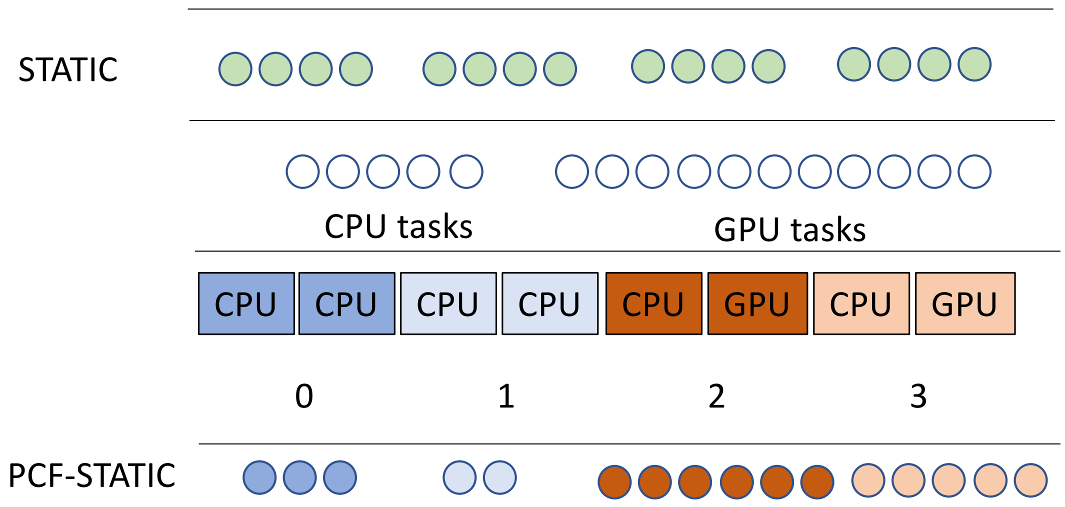
PCF-STATIC scheduler

- Divide in two parts the set of tasks
  - `NTASKS = PCF x CPU-TASKS + CPU-TASKS = (PCF+1) x CPU-TASKS`
  - `GPU-TASKS = NTASKS – CPU-TASKS`
- Apply STATIC over `CPU-TASKS` and among all CPU-based CUs
- Apply STATIC over `GPU-TASKS` and among all GPU-based CUs

- Maximizes adjacency, so minimizes communications

PCF-STATIC scheduler

- Example:
  - PCF = 2
  - NCU = 4
    - CPU-based CU = 2
    - GPU-based CU = 2
  - NTASKS = 16
  - CPU-TASKS = 5
  - GPU-TASKS = 11

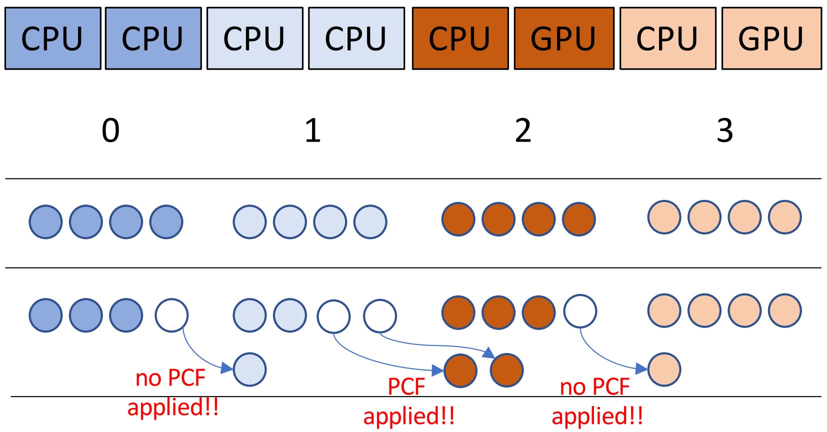


<start,end>: interval of tasks assigned to a CU

STATIC:	<0,3>	<4,7>	<8,11>	<12,15>
PCF-STATIC:	<0,2>	<3,4>	<5,10>	<11,15>

PCF-GUIDED scheduler

- Initially, apply a STATIC scheduling
- Execute and monitor task execution time
- Adapt the work distribution so that it gets balanced
    - ALL-WORK = addition of all task execution times factorized with PCF
    - WORK-CU = ALL-WORK / NCU
    - Keep the assigned work per CU close to WORK-CU
- Maximizes adjacency, so minimizes communications



<start,end>: interval of tasks assigned to a CU

STATIC: <0,3> <4,7> <8,11> <12,15>
PCF-GUIDED: <0,2> <3,5> <6,10> <11,15>

PCF-GUIDED scheduler

- Samples of execution time are taken from `RUNTIME::executeTask()` and `RUNTIME::commitTask()`
- Code scheme for work balance

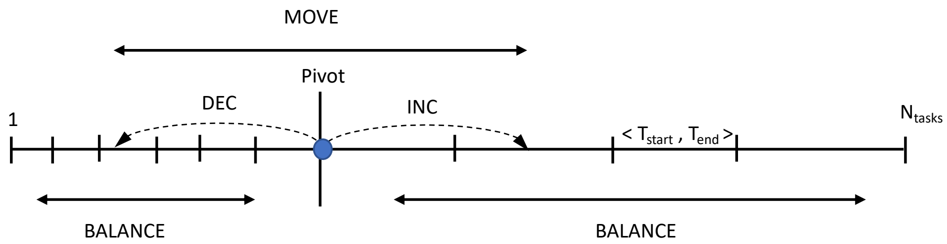
```
for cu = 0, NCU-1
  WORK[cu]= "add tasks execution times"

for cu = 0, NCU-2
  while WORK[cu]>WORK-CU /* With some threshold */
    <start[cu],end[cu]-->
    WORK[cu] -= "execution time of end task"
    WORK[cu+1] += "execution time of end task refactorized with PCF"

  while WORK[cu]<WORK-CU /* With some threshold */
    <start[cu],end[cu]++>
    WORK[cu] += "execution time of end task refactorized with PCF"
    WORK[cu+1] -= "execution time of end task"

  start[cu+1] = end[cu]
```

```
#pragma omp parallel \
            num_threads(RUNTIME::cpus+RUNTIME::gpus)\
            proc_bind(true)
{
  unsigned int task;
  task = RUNTIME::getTask();
  while (task!=NO_TASK) {
    unsigned int zone = task;

    unsigned int CU = RUNTIME::getCU();
    if (PLACEMENT::zonePlacement[zone]!=CU &&
        RUNTIME::isGPU(CU))
      PLACEMENT::migrateZone(CU);

    RUNTIME::executeTask();

     /* Computational Stages */

    if (RUNTIME::isGPU(CU)) RUNTIME::synchronize();

    RUNTIME::commitTask();

    task = RUNTIME::getTask();
  } // while
} // parallel
```

g for

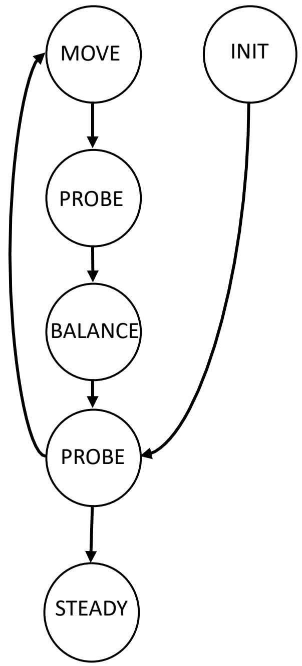
CLUSTERED GUIDED scheduler

- Set of tasks is divided in two clusters
  - One for CPU-based CUs, the other for GPU-based CUs
  - PIVOT: separating point for the two clusters
- At each instance of the scheduler, the PIVOT moves left/right to balance the work assigned to each cluster
  - Sort of a dichotomic search
- The scheduler balances the work distribution within each cluster



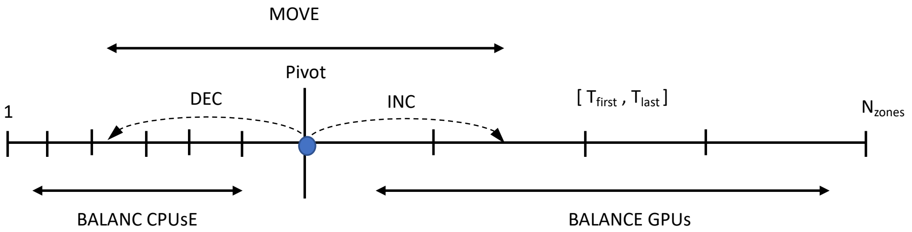
CLUSTERED GUIDED scheduler

- The scheduler evolves at runtime, switching between different states:
  - `INIT`: PIVOT is initialized (e.g.: PIVOT=NCPU). Applies a STATIC scheduling in both clusters.
  - `MOVE`: Moves PIVOT, according to where the maximum execution time has occurred.
  - `PROBE`: Samples execution time for each CU. Determines the maximum value observed in each cluster.
  - `BALANCE`: Applies a PCF-GUIDED scheduler with PCF=1 to each cluster.
  - `STEADY`: Records final configuration for work distribution.



CLUSTERED GUIDED scheduler

<u>Data Structures:</u>
Pivot: Index that indicates the border between tasks assigned to CPUs and tasks assigned to GPUs
DEC: Value to decrement the Pivot
INC: Value to inclement the Pivot
TaskTime []: Vector of N_{zones} elements, each one describing the execution time of a task
CUtime []: Vector of N_{CUs} elements, each one describing the execution time of CU
FirstTask []: Vector of N_{CUs} elements, each one describing the T_{first} task assigned to a CU
LastTask []: Vector of N_{CUs} elements, each one describing the l_{Tast} task assigned to a CU



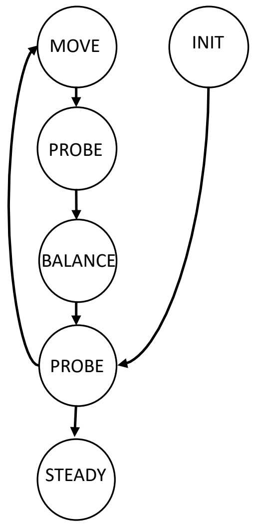
<u>MOVE:</u>
T_{cpu} = max time for CPUs
T_{gpu} = max time for GPUs
if $T_{cpu} > T_{gpu}$
 INC = INC / 2
 goingLEFT = false
 goingRIGHT = true
 pivot += INC

if $T_{gpu} > T_{cpu}$
 DEC = DEC / 2
 goingLEFT = true
 goingRIGHT = false
 pivot-= DEC

<u>BALANCE (CPUs/GPUs):</u>
for cu = 1 , N_{CUs}
 TotalWork = CUtime[cu]

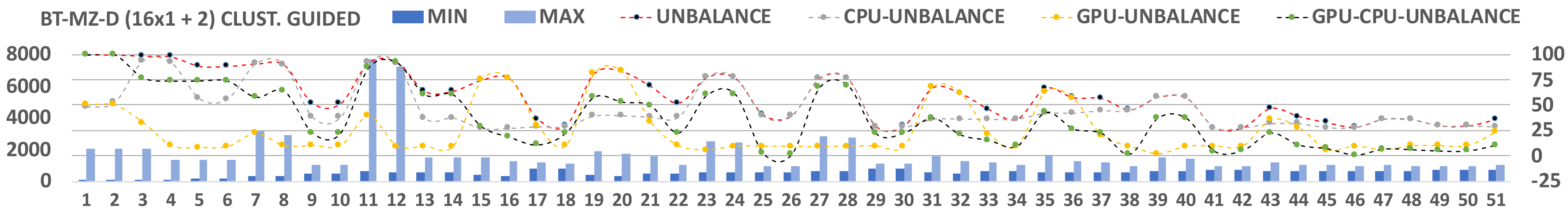
WorkPerCU = TotalWork / N_{CUs}
for cu = 1 , N_{CUs}
 First = FirstTask[cu]
 Last = LastTask[cu]
 Work = $\sum_{First}^{Last} T_i$
 DIFF = | Work - WorkPerCU |
 if (Work < WorkPerCU && **DIFF > TH**)
  while (Work < WorkPerCU) Last++; Work =+ T_{last}
 else if (Work > WorkPerCU && **DIFF > TH**)
  while (Work > WorkPerCU) Last--; Work =- T_{last}
 LastTask[cu] = Last
 FirstTask[cu] = Last + 1

<u>INIT</u>:
Pivot = N_{CPUs}
INC = DEC = N_{tasks} / 2

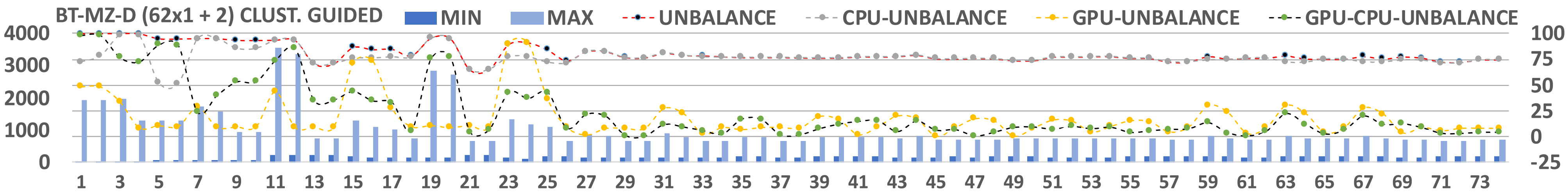


Number of steps to STEADY state: $log_2(N_{tasks})\ x\ 4$

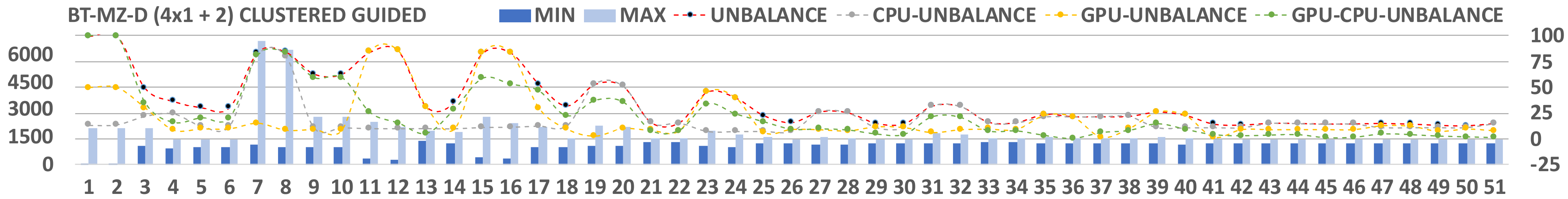
Examples of Clustered Guided



Examples of Clustered Guided



Examples of Clustered Guided



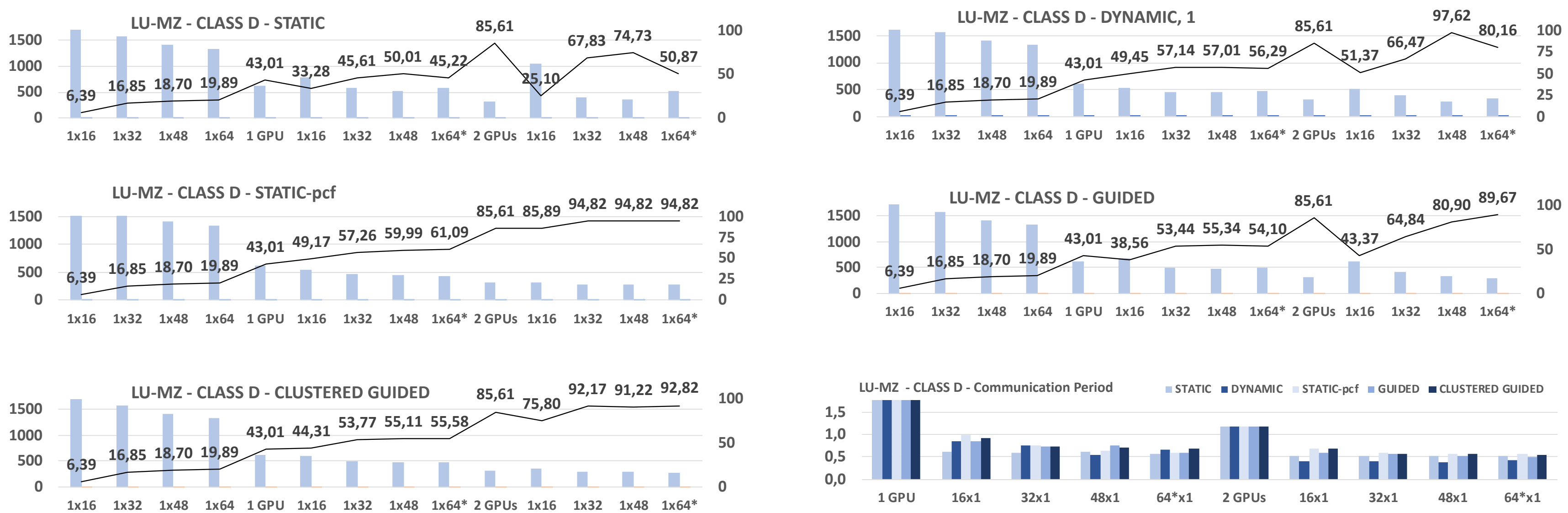
Outline

- Analysis, parallelism characterization of NAS-MZ suite of benchmarks
- NPB-MZ Hybrid Design and Implementation
- Work distribution schemes for hybrid executions
- Evaluation Results

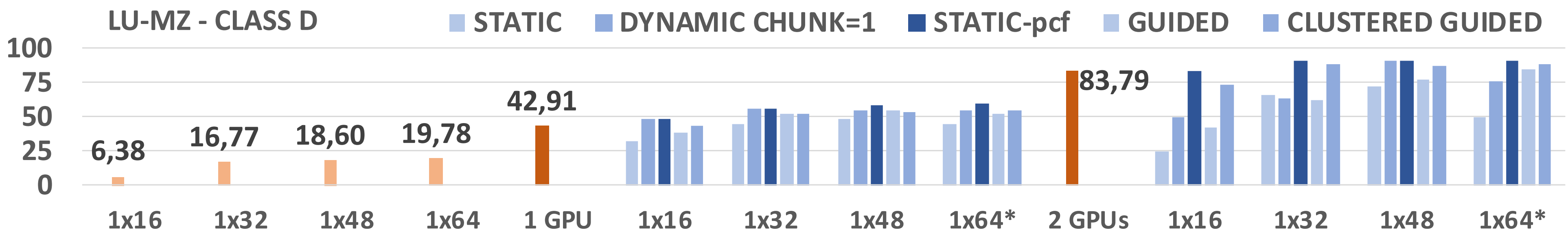
Environment

- AMD CTE @ BSC
  - AMD EPYC 7742 @ 2.250GHz (64 cores)
  - 2 x GPU AMD Radeon Instinct MI50 with 32GB
- Software stack
  - GCC 8.3.1–4
  - Radeon Open Compute (ROCm) Runtime software stack 3.5.0
- Hybrid NPB-MZ implementation
  - Parallel CUDA multi-gpu version of the NPB-MZ in C++
  - Parallel (OpenMP) implementation in Fortran of NPB-MZ
  - Runtime Libraries: `libCU-rtl`, `libPLACEMENT-rtl`, `libSCHEDULING-rtl`

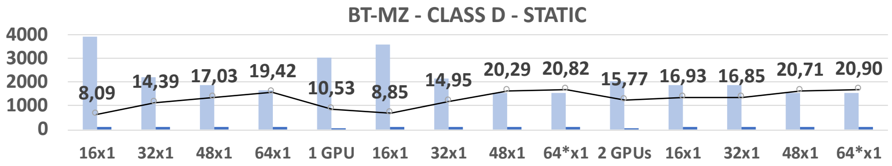
LU-MZ: Compute and Communication

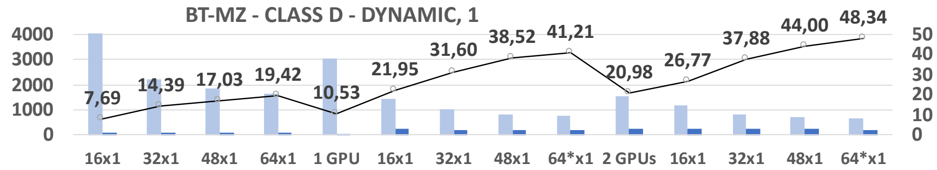


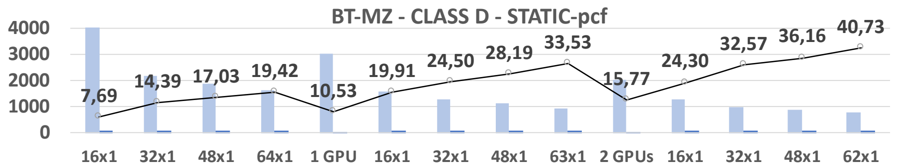
LU-MZ: Overall Performance

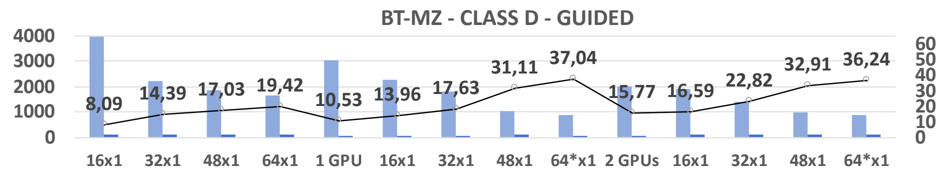


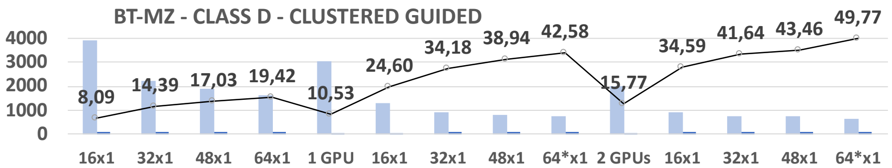
BT-MZ: Compute and Communication

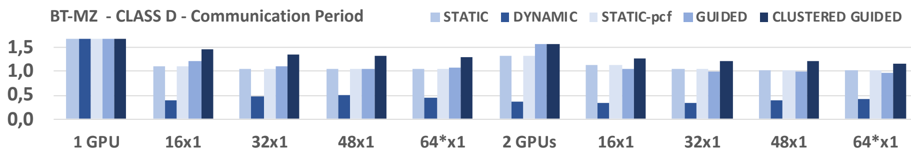




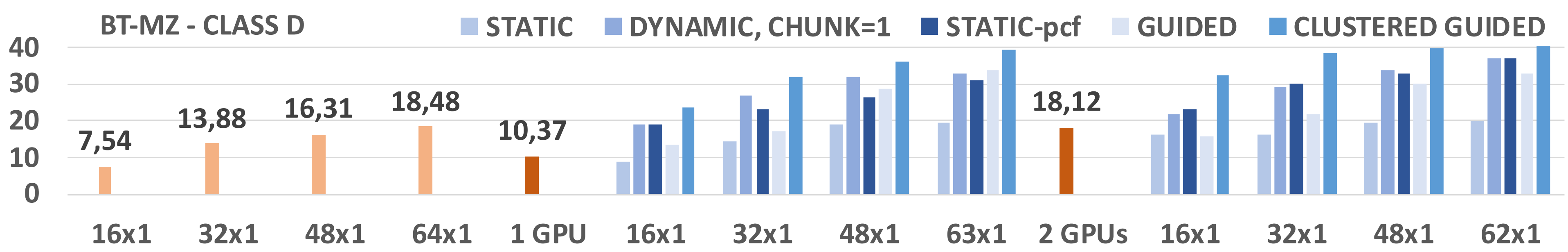




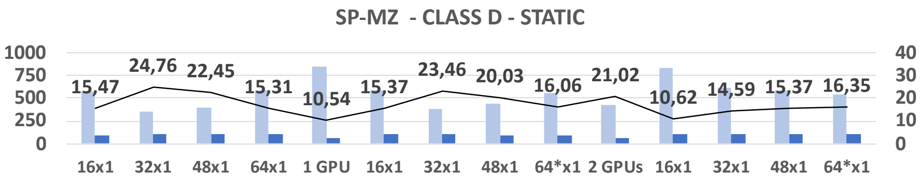


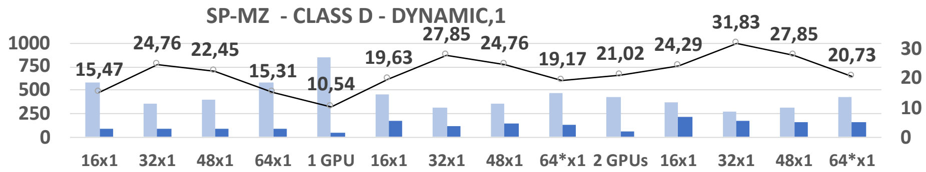


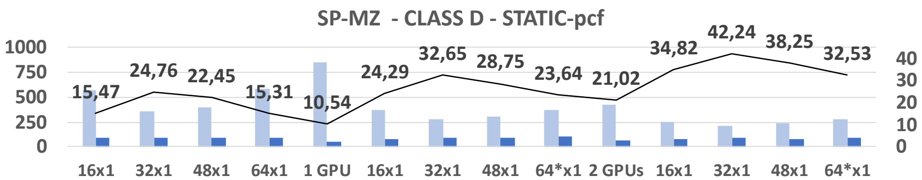
BT-MZ: Overall Performance

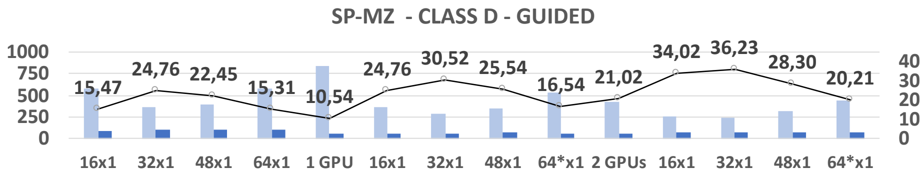


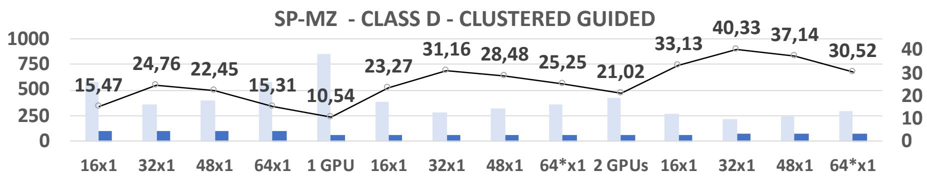
SP-MZ: Compute and Communication

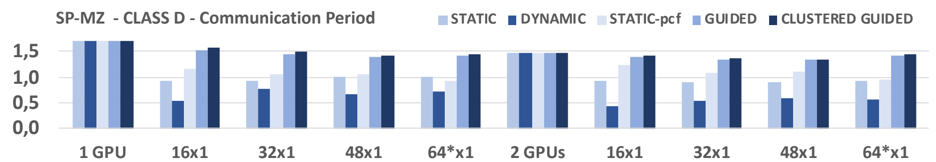




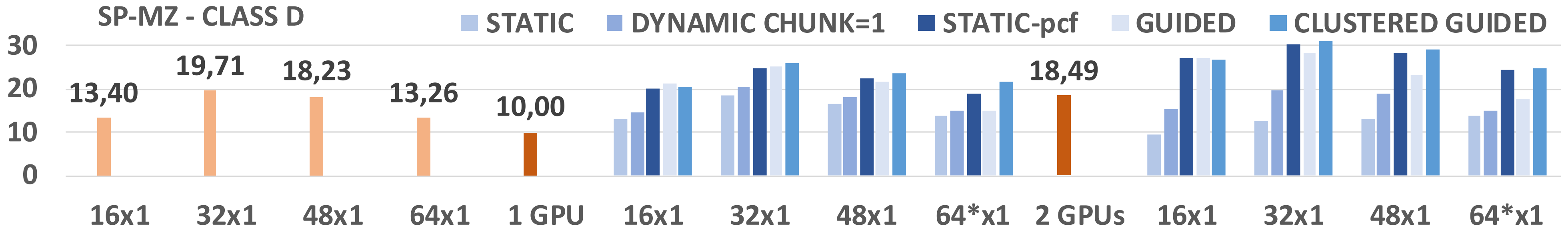








SP-MZ: Overall Performance



Conclusions

- Case of study for hybrid computing (NPB-MZ), using OpenMP+CUDA
  - Methodology based on runtime implementation for bridging the two execution models
    - CU abstraction, AS abstraction at the programming model level
    - Work distribution schemes for hybrid executions
      - Based on rough performance comparison (PFC)
      - Monitoring task execution times (Clustered Guided)
- In general, the more GPUs in place, less space for using the CPUs
  - From a pure “FLOPS” perspective, we should try to use the CPUs for other purposes
    - Design runtime systems that solve programming limitations: memory allocators for dynamically changing data layouts, work distribution schemes, even code optimization ...

Heterogeneous Computing for Scientific Applications

Marc Gonzalez Tallada
Associate professor
Computer Architecture Department
Technical University of Catalonia

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH