Power Savings

the quest for green MPI

Martin Hilgeman

EMEA product technologist HPC

Disclaimer

The content covered in this presentation is just an investigation

- It is not:
  - Related to any Dell specific hardware or HPC solution
  - On the development roadmap
  - Likely to used in any customer environment (at this time)

Agenda

Agenda

- Definition of system load
- Observations
- MPI basics
- Processor features
- Putting it all together
- Benchmark numbers
- MPI profiling interface
- Conclusion

Just some HPC application

The system load of my application

```
Tasks: 425 total,  17 running, 408 sleeping,   0 stopped,   0 zombie
Cpu(s):  99.3%us,  0.5%sy,  0.0%ni, 0.3%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  65929784k total, 23199744k used, 42730040k free,   125628k buffers
Swap: 33038328k total,        0k used, 33038328k free,  8317304k cached

   PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+   P COMMAND
 32500 martinh   20   0 1090m 787m  17m R 100.0  1.2   6:43.74 11 my_application
 32488 martinh   20   0 2157m 1.8g  23m R 100.0  2.9   6:38.10  0 my_application
 32493 martinh   20   0 1088m 788m  20m R 100.0  1.2   6:43.77  4 my_application
 32495 martinh   20   0 1092m 792m  20m R 100.0  1.2   6:43.60  6 my_application
 32496 martinh   20   0 1089m 788m  19m R 100.0  1.2   6:42.21  7 my_application
 32497 martinh   20   0 1091m 789m  19m R 100.0  1.2   6:42.71  8 my_application
 32499 martinh   20   0 1081m 779m  18m R 100.0  1.2   6:44.46 10 my_application
 32503 martinh   20   0 1083m 781m  16m R 100.0  1.2   6:45.14 13 my_application
 32489 martinh   20   0 1083m 779m  17m R 99.9   1.2   6:43.60  1 my_application
 32490 martinh   20   0 1084m 782m  18m R 99.9   1.2   6:43.28  2 my_application
 32492 martinh   20   0 1084m 781m  18m R 99.9   1.2   6:42.73  3 my_application
 32494 martinh   20   0 1091m 790m  20m R 99.9   1.2   6:43.14  5 my_application
 32498 martinh   20   0 1090m 788m  18m R 99.9   1.2   6:43.21  9 my_application
 32501 martinh   20   0 1089m 785m  17m R 99.8   1.2   6:43.65 12 my_application
 32504 martinh   20   0 1082m 781m  16m R 99.8   1.2   6:44.40 14 my_application
 32505 martinh   20   0 1082m 777m  15m R 99.8   1.2   6:44.22 15 my_application
```

My application is 100% busy on all cores, but is it doing any useful work?

Observation

I am using a leading CSM application, proved to scale to 1000s of cores

- Iterative solver process
- Solver is processed in core
- Parallelized with MPI

"I am seeing 100% CPU utilization, so my application is using my system efficiently!"

Actually…

```
  PID USER        PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+   P COMMAND
32500 martinh    20   0 1090m 787m  17m R 100.0  1.2   6:43.74 11 my_application
```

CAN be doing (this is a debugger backtrace):

```
#4  opal_event_base_loop (base=0xe6b0ff0, flags=<value optimized out>)
    at event.c:855
#5  0x00007fcfd8623909 in opal_progress () at runtime/opal_progress.c:189
#6  0x00007fcfd8571f75 in opal_condition_wait (count=2,
    requests=0x7fff54a2ad70, statuses=0x7fff54a2ad40)
    at ../opal/threads/condition.h:99
#7  ompi_request_default_wait_all (count=2, requests=0x7fff54a2ad70,
    statuses=0x7fff54a2ad40) at request/req_wait.c:263
#8  0x00007fcfd45ae65e in ompi_coll_tuned_sendrecv_actual (sendbuf=0x0,
    scount=0, sdatatype=0x5e98fc0, dest=27, stag=-16,
    recvbuf=<value optimized out>, rcount=0, rdatatype=0x5e98fc0, source=27,
    rtag=-16, comm=0xe7945f0, status=0x0) at coll_tuned_util.c:54
#9  0x00007fcfd45b6a6e in ompi_coll_tuned_barrier_intra_recursivedoubling (
    comm=0xe7945f0, module=<value optimized out>) at coll_tuned_barrier.c:172
#10 0x00007fcfd857f282 in PMPI_Barrier (comm=0xe7945f0) at pbarrier.c:70
#11 0x00007fcfd88d6373 in mpi_barrier_f (comm=<value optimized out>,
    ierr=0x7fff54a2ae6c) at pbarrier_f.c:66
```

Or…

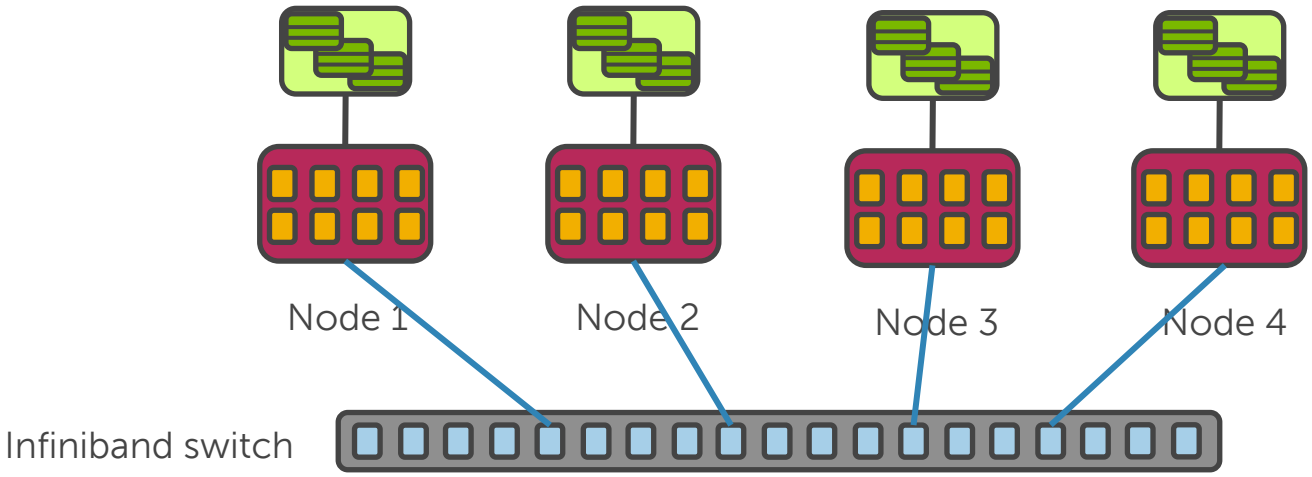
```
#3  opal_event_base_loop (base=0xe6b0ff0, flags=<value optimized out>)
    at event.c:850
#4  0x00007fcfd8623909 in opal_progress () at runtime/opal_progress.c:189
#5  0x00007fcfd8572505 in opal_condition_wait (req_ptr=0x7fff54a2ac08,
    status=0x7fff54a2abf0) at ../opal/threads/condition.h:99
#6  ompi_request_wait_completion (req_ptr=0x7fff54a2ac08,
    status=0x7fff54a2abf0) at ../ompi/request/request.h:377
#7  ompi_request_default_wait (req_ptr=0x7fff54a2ac08, status=0x7fff54a2abf0)
    at request/req_wait.c:38
#8  0x00007fcfd859686d in PMPI_Wait (request=0x7fff54a2ac08,
    status=0x7fff54a2abf0) at pwait.c:70
#9  0x00007fcfd88dcfea in mpi_wait_f (request=0x7fff54a2ac54,
    status=0xa60a420, ierr=0x7fff54a2ac50) at pwait_f.c:66
#10 0x0000000003e6e1e7 in my_wait_ ()
```

The bottom line:

“waiting” **can** be very CPU intensive!

MPI 101

- MPI is a programming interface that allows one to use multiple systems (or nodes) to work on a single problem in parallel
- Each cluster node has its own memory space and is interconnected through a fast, low latency network



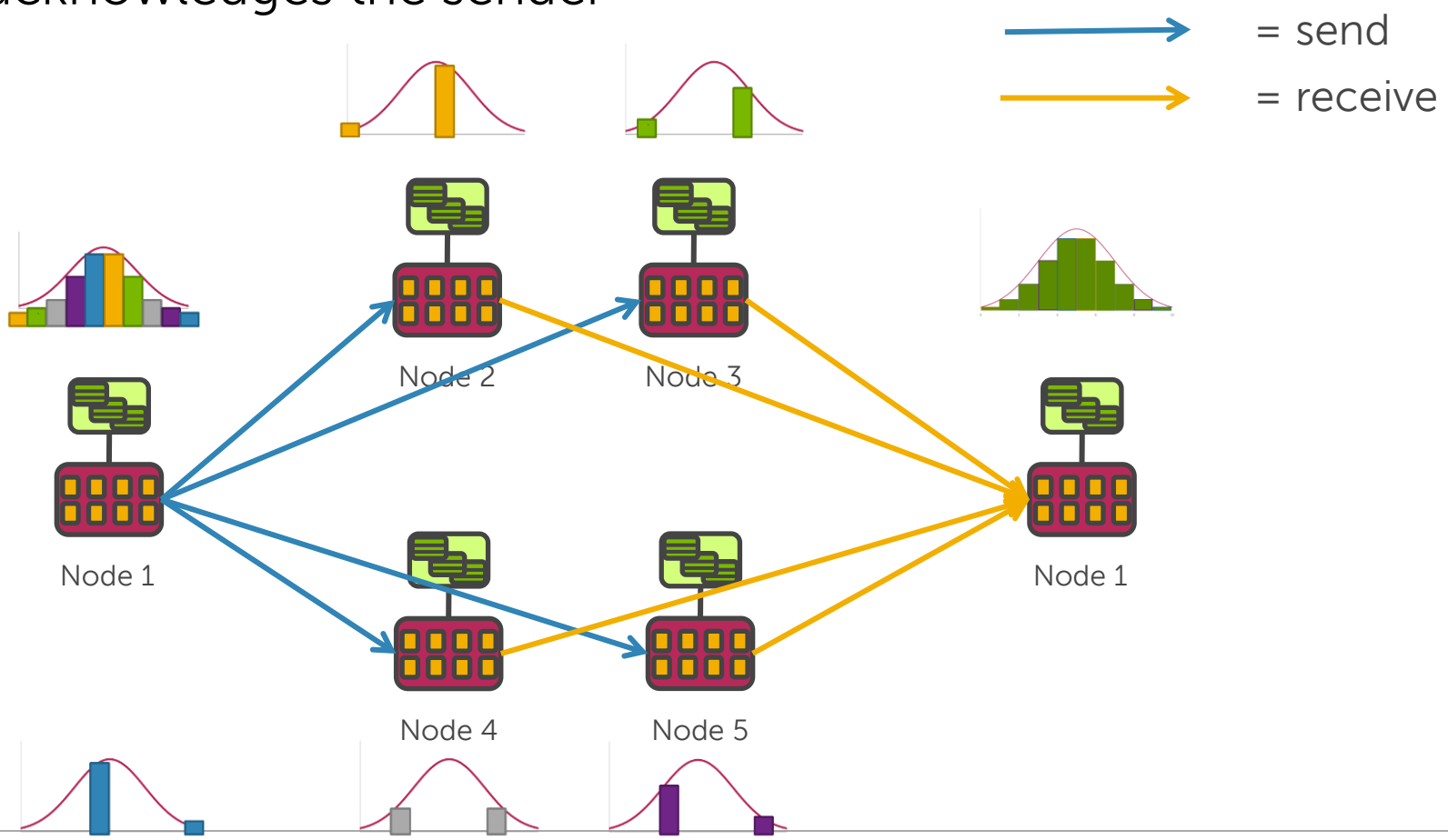
- Data is transferred between nodes as *messages* (with an envelope containing *sender*, *destination* and *reference* tag)

Think of an express service

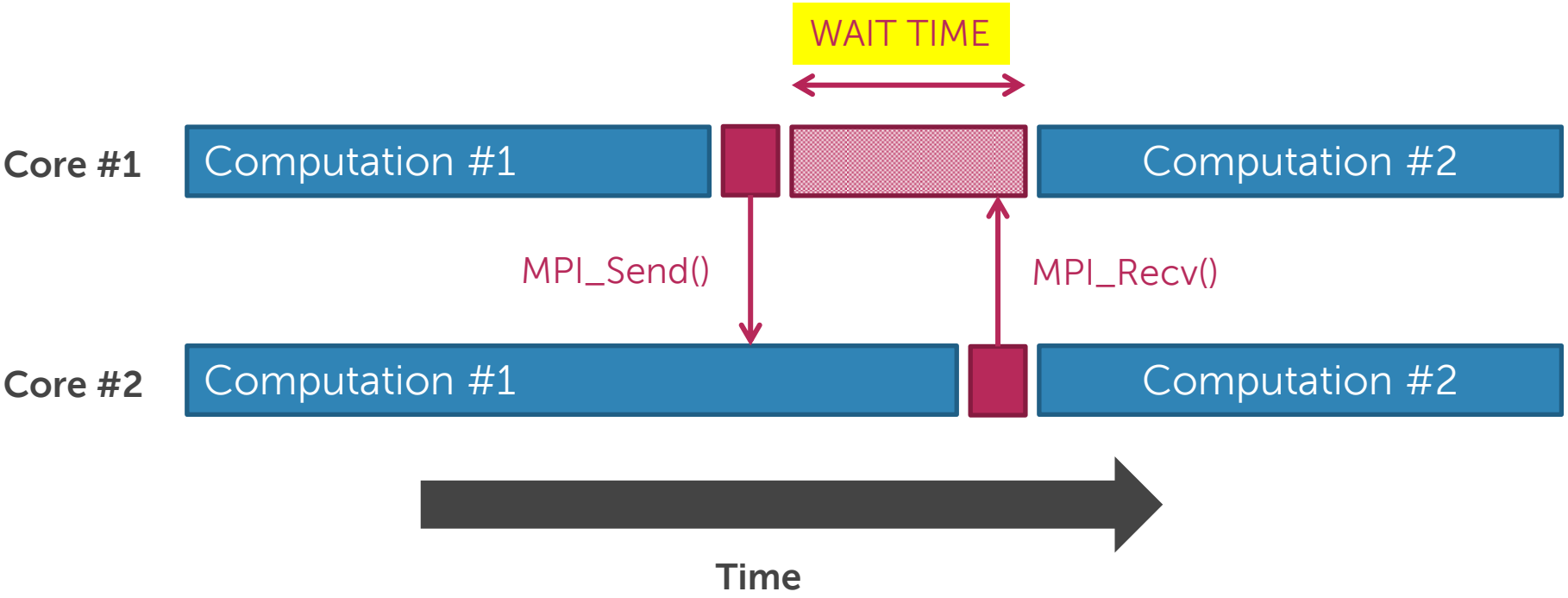
Data is transferred between nodes as *messages* (with an envelope containing *sender*, *destination* and *reference* tag)

MPI 101

- MPI communication is normally **2-way** (the receiver acknowledges the sender



So what is happening?



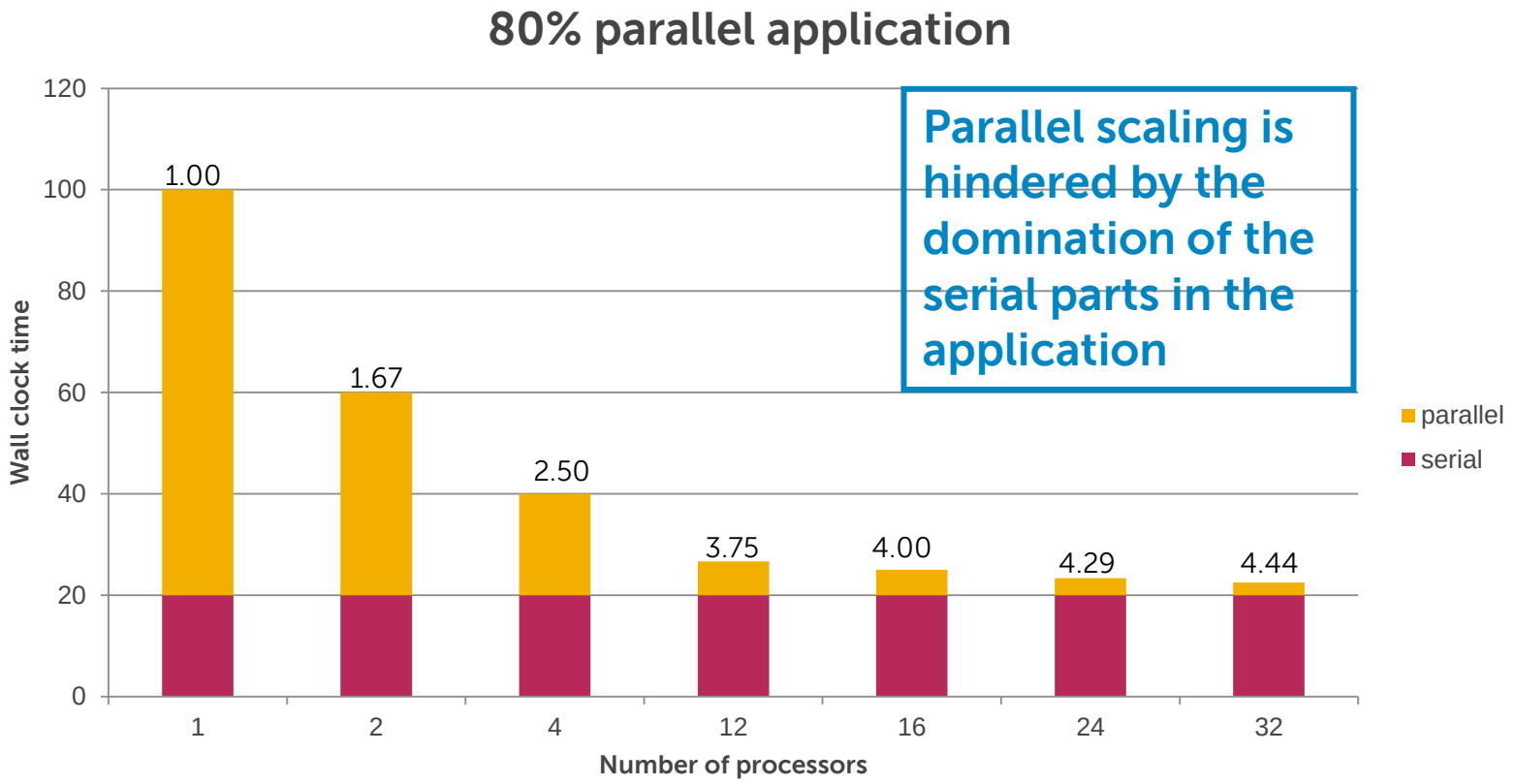
- Core #1 is done with its computation and sends a message to Core #2
- Core #2 is still busy with its computation
- Core #1 has to wait for the receive acknowledgement from Core #2
- While doing that, Core #1 is waiting and keeps polling the network interface for incoming messages

Why has it been designed like this?

- The polling for incoming data is keeping the CPU at 100% utilization
- All major MPI libraries do it by default
  - Intel MPI
  - HP MPI / Platform MPI
  - Open MPI
  - MPICH/MVAPICH
- The polling *can* be switched off in MPI libraries, but degrades your performance greatly
- The aim for any MPI library is to get the lowest latency and maximum bandwidth for data exchange between CPU cores
- HPC is about keeping the CPUs 100% busy, so this is good practice

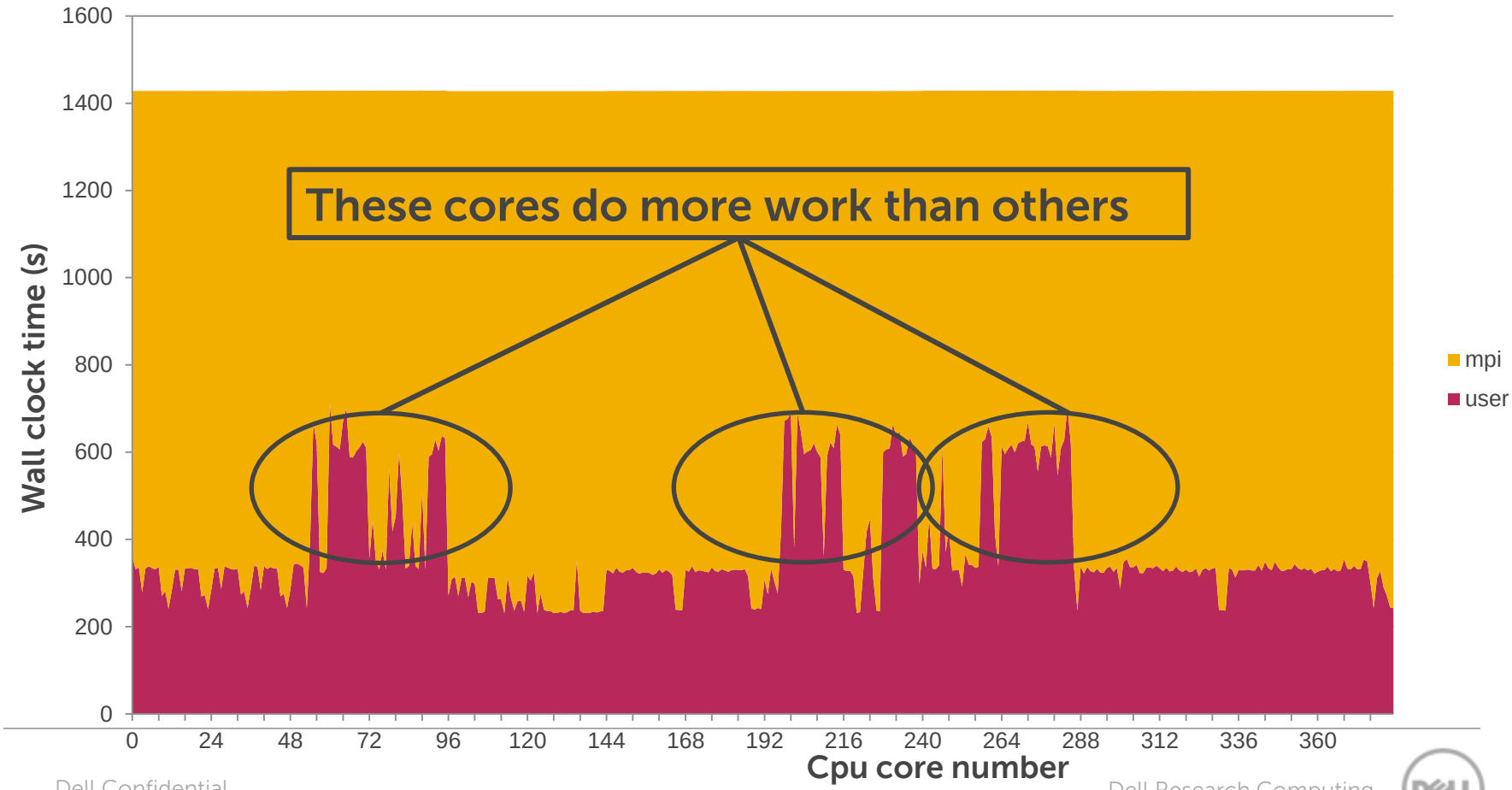
The reality is that: #1

- Not all applications scale linearly and have serial parts (think of Amdahl's Law)



The reality is that: #2

- Not all applications are equally balanced across all the CPU cores



So what if we…

Try to do something smart during these “wait” times?

Keeping the CPU 100% busy while not doing any computation, does it make sense?

Goals

- Can we make the processor use less power during these “wait” times?
- Can we improve performance at the same time?

What hardware features do we have on hand?

Reduce power: Processor P-States

- Use the Processor Performance Power States (P-States)
  - Setting the processor into a higher P-State lowers its power consumption

P0	Processor consumes max power and is at max performance
P1	Processor consumes less power and frequency is lowered
Pn	Processor is at lowest power state (n<16)

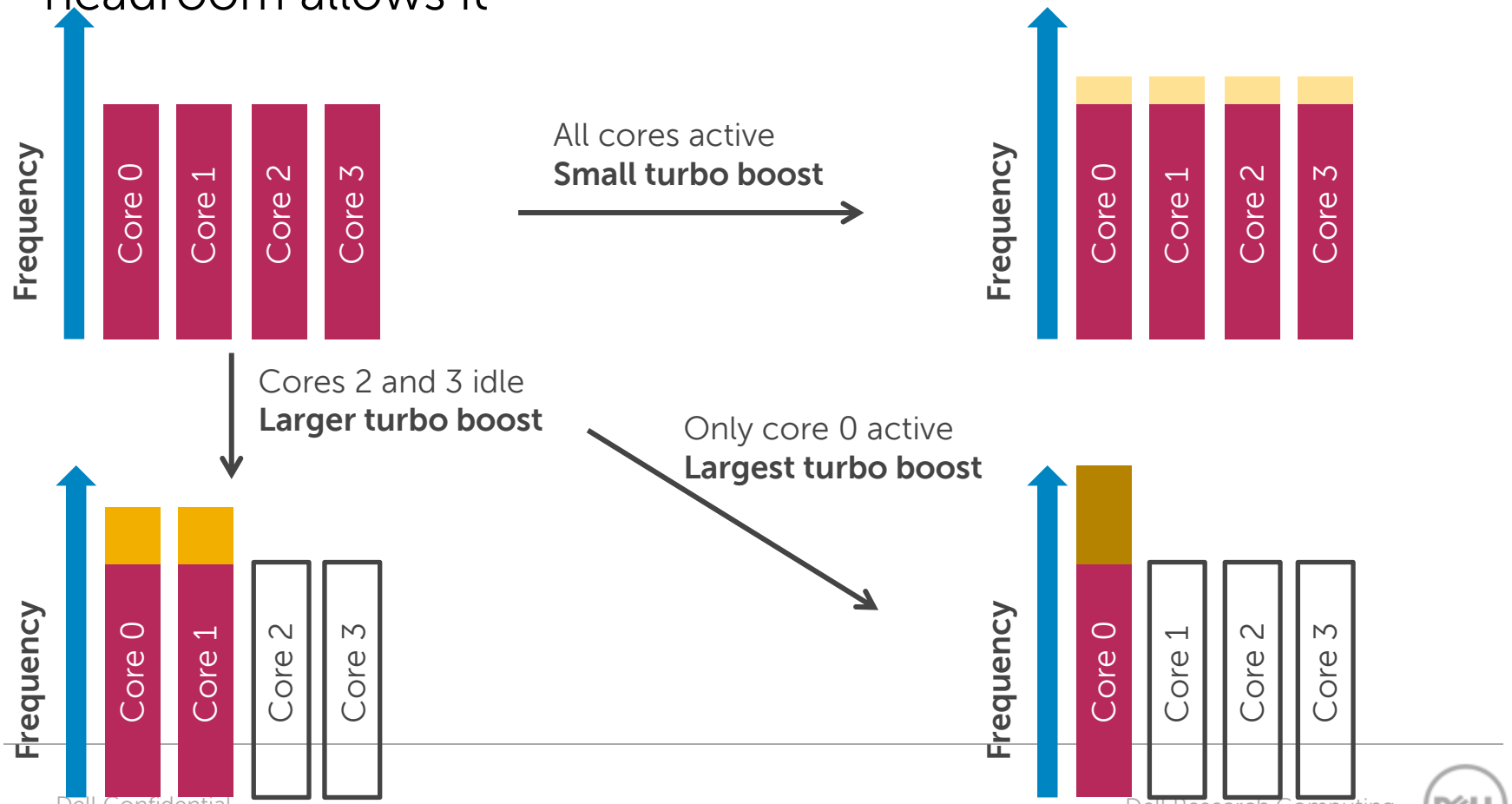
P-States in practice

- Both AMD and Intel do support P-States
  - Intel Enhanced SpeedStep® for Intel Sandy Bridge processors
  - AMD PowerNow!TM for Opteron 6200/6300 series processors
- P-States are part of the ACPI 4.0 (Advanced Configuration and Power Interface) specification
  - An industry standard to define system motherboard device configuration and power management
  - Supported both by Microsoft Windows and Linux operating systems

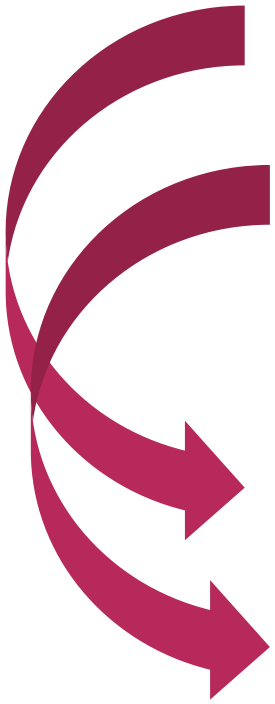
The Linux kernel has performance governors that (allow) control of the P-States on a per core level

Improve performance :Turbo Boost/Turbo Core mode

- Both AMD and Intel have capabilities to allow CPU cores to operate above their nominal frequency if the thermal headroom allows it



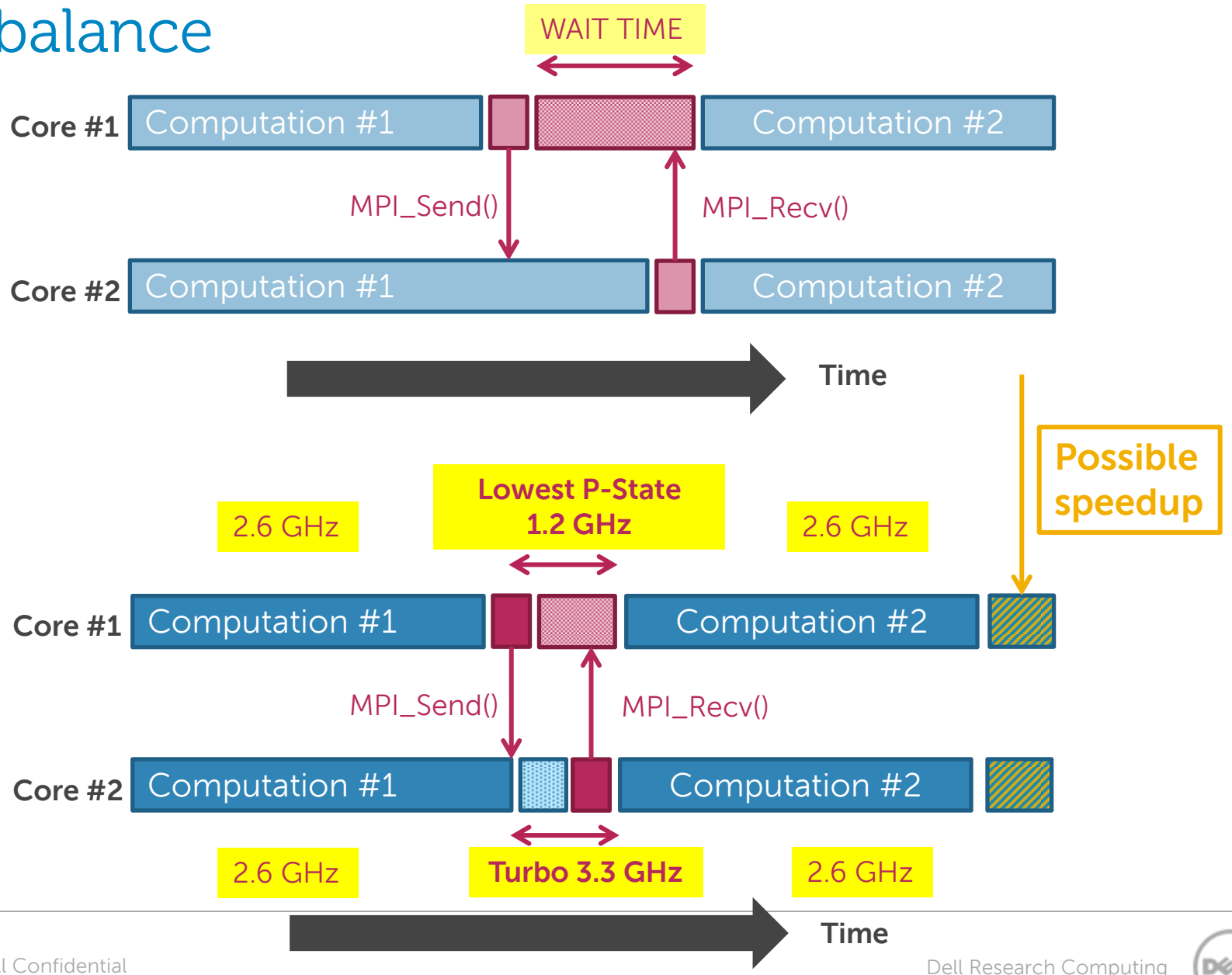
Goals revisited



- Can we make the processor use less power during these “wait” times?
- Can we improve performance at the same time?

- Put the processor core in a lower P-State during these wait times. Lower frequency means less power
- Other cores who are still doing calculations could use their Turbo mode capability to a much greater level to reduce load imbalance

The effect of turbo mode could help load imbalance



Implementation – Step 1

- Identify the MPI functions that are waiting for incoming traffic
  - These are "blocking" functions (i.e. the CPU core cannot do anything else in the mean time)
  - The important MPI functions that do this are:
    - MPI_Send() MPI_Recv()
    - MPI_Barrier() MPI_Wait()
    - MPI_Sendrecv() MPI_Probe()
- Itentify the MPI functions that do collective communication
  - These functions have an implicit barrier and a lot of data movement is involved
  - The important MPI functions that do this are:
    - MPI_Allreduce() MPI_Alltoall() MPI_Allgatherv()
    - MPI_Allgather() MPI_Alltoallv()

Implementation – Step 2

- Wrap these functions in a library so that:
  1. On entry, the CPU core is clocked to a lower frequency. The polling can easily be done at ~ 1.2 GHz **without** sacrificing performance!
  2. The normal MPI function call is executed
  3. On exit, the CPU core is reset to its normal speed (2.6 GHz for an E5-2670)

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status *status)
{
1   …
    set_cpu_freq(task.cpu);
2
    ret = PMPI_Recv(buf, count, datatype, source, tag, comm, status);
3
    reset_cpu_freq(task.cpu);

    return ret;
}
```

Implementation: **dell_toolbox.so**

- The library intercepts the MPI function calls, **no** changes to the call itself!
- Real MPI function is called, so it works with any MPI library (Intel MPI, Open MPI, MPICH2, MVAPICH2, Platform MPI, etc)
- Applications do **not** need to be modified

```
mpirun -np 128 -e LD_PRELOAD=dell_toolbox.so dell_affinity.exe <application>
```

- dell_affinity.exe program is needed to bind the MPI ranks to a core
  - Otherwise the library does not know which CPU core to clock down ☺

Polling implementation in MPI libraries

- The MPI libraries go into a tight spin/wait loop to check for incoming messages
  - It uses system calls that check for file descriptors holding data
  - The most common system calls are `select()`, `poll()`, `epoll()` and `kqueue()`

MPI library	Spin/Wait type
Open MPI	Polls every 10 ms, then is able to yield the CPU
Intel MPI	Spins 250 times, then yields the CPU
Platform MPI	Spins 10,000 times, then yields the CPU
MVAPICH	Spins 2,000 times, then is able to yield the CPU

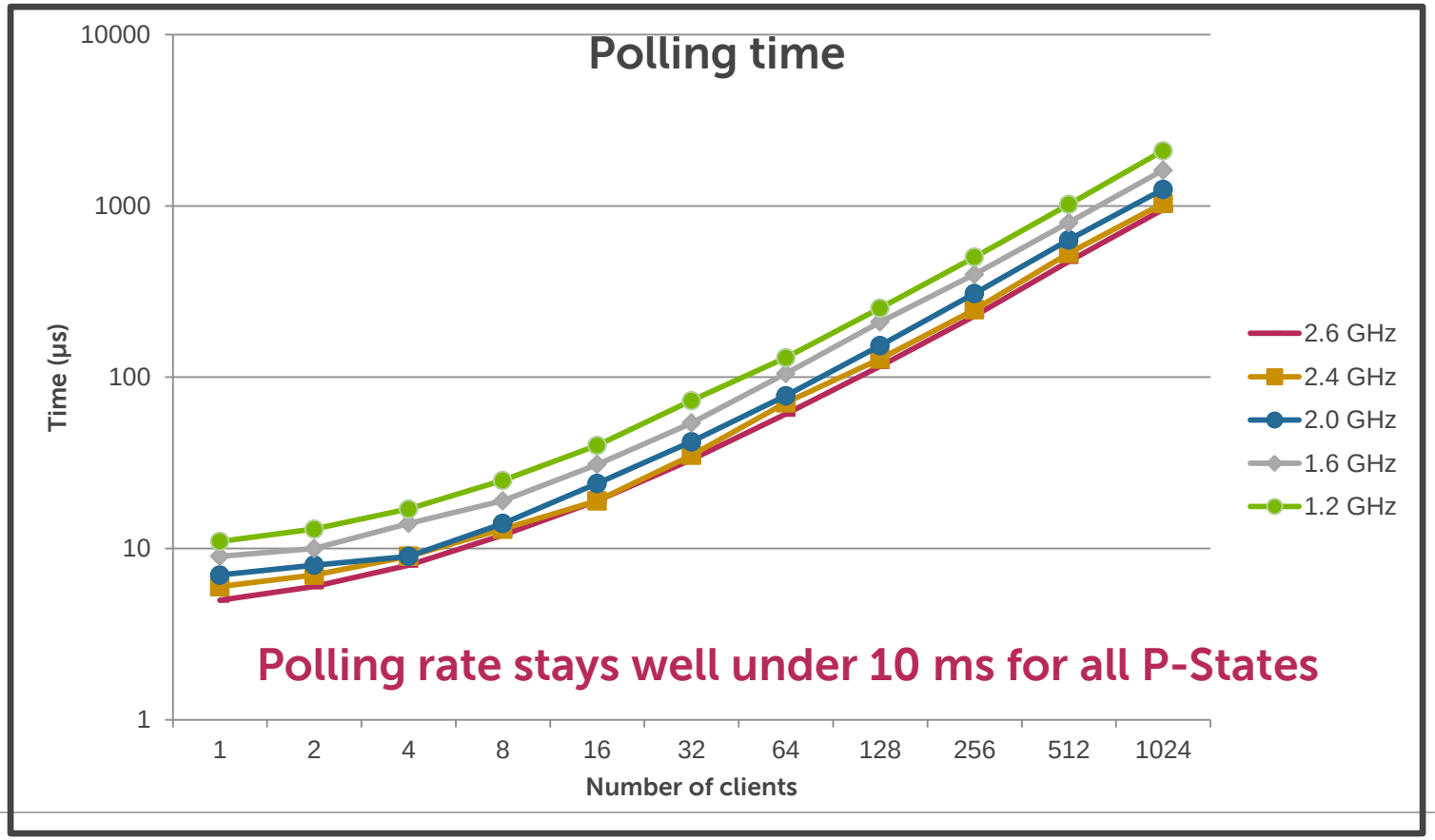
P-state switching latency for an Intel E5-2670 processor

- Switching from one P-state to another involves a latency
  - Unfortunately this is not a uniform number

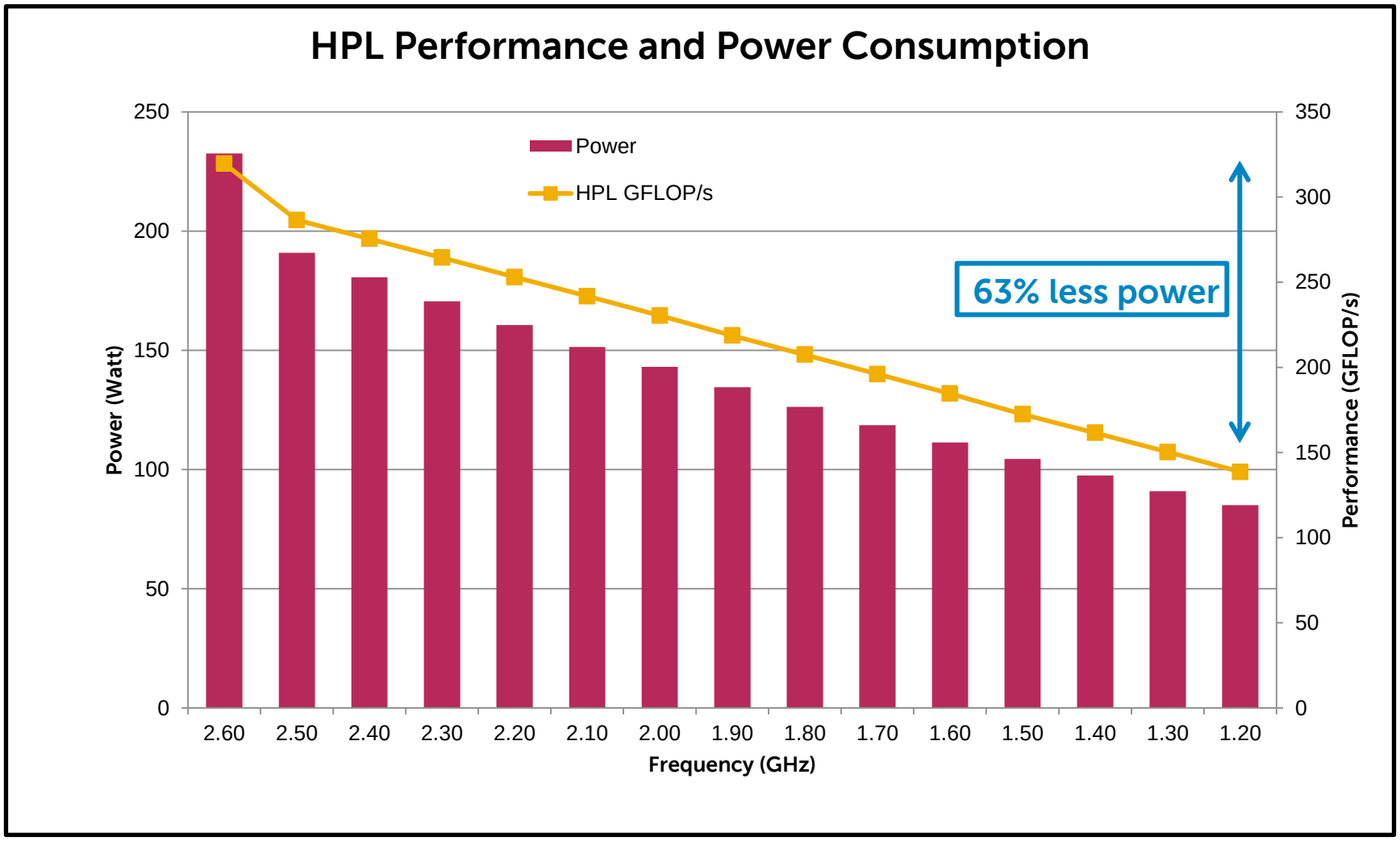
From 2.7 GHz to…	Min (µs)	Max (µs)	Average (µs)
2.6 GHz	4.29	17.40	5.87
2.2 GHz	4.53	18.12	6.87
1.6 GHz	4.29	25.03	7.40
1.2 GHz	4.53	23.13	8.24

Does the lowest P-State degrade performance?

Remember: “waiting” is CPU intensive! The lowest P-state should not make the waiting itself longer



Does the lowest P-State save power?



Startup messages

```
libtoolbox: process 103788 is pinned to host c6220-5 CPU 0
libtoolbox: host c6220-5 CPU 0: current frequency is 2600000
libtoolbox: host c6220-5 CPU 0: minimum frequency is 1200000
libtoolbox: host c6220-5 CPU 0: maximum frequency is 2601000
libtoolbox: process 96912 is pinned to host c6220-7 CPU 15
libtoolbox: host c6220-7 CPU 15: current frequency is 2600000
libtoolbox: host c6220-7 CPU 15: minimum frequency is 1200000
libtoolbox: host c6220-7 CPU 15: maximum frequency is 2601000
```

- The library reports that the current frequency is 2.6 GHz (it is an Intel E5-2670 processor core)
- The lowest P-State is 1.2 GHz, which is what we are going to use for the functions that are intercepted

Termination messages (end of run)

Wall clock time was 669 seconds on 64 cores

```
libtoolbox: host c6220-5 CPU 0: total MPI wait time: 269.122 s. Total time
spent in frequency scaling: 1.25995398 s.  Number of frequency state
changes: 2207131
libtoolbox: host c6220-7 CPU 15: total MPI wait time: 205.105 s. Total
time spent in frequency scaling: 4.57049227 s.  Number of frequency state
changes: 3655628
```

NAS parallel benchmark type D results

Kernel	Ncores	As is Walltime (s)	As is Power (W)	P-state Walltime (s)	P-state Power (s)	Walltime delta (%)	Power Delta (%)
BT	64	405	221	406	220	0.3	-0.3
BT	144	187	218	188	217	0.5	-0.5
CG	64	180	211	182	204	1.3	-3.4
EP	64	36	179	31	167	-15.2	-6.7
EP	144	17	169	17	147	-0.1	-12.8
FT	64	159	224	165	179	3.7	-20.1
IS	64	22	179	23	162	3.8	-9.4
LU	64	280	222	281	218	0.4	-1.8
LU	144	126	216	127	208	1.2	-3.7
MG	64	34	213	35	210	3.0	-1.5
SP	64	610	212	611	211	0.2	-0.5
SP	144	241	211	245	209	1.8	-1.1

Implementation status (as of September 2013)

- Library works with Intel MPI, Open MPI and MVAPICH2 (tested), should also work with others
- Application testing done with WRF, CP2k, VASP, LS-DYNA and GROMACS
- Performance looks OK-ish, but it is too early to make any judgment (lots of room for tuning)
- Basic power measurements have been done with Intel PCM

To do list (as of September 2013)

- Frequency switching needs super-user privilege, needs to be fixed in some way
- P-state switching can be done in around 5 microseconds, but the sysfs interface has a non-uniform latency
  - This could fits nicely in the time for a MPI collective or blocking message
- Need real benchmark numbers, preferably on a larger system (> 64 cores)
  - The larger the system, the bigger the communication time and the more room for savings
- Real power measurements with an external power meter

References

1. ACPI specification 4.0a - http://www.acpi.info/DOWNLOADS/ACPIspec40a.pdf
2. The Ondemand Governor, Past, Present, and Future - *Venkatesh Pallipadi, Alexey Starikovskiy* - ftp://ftp.kernel.org/pub/linux/kernel/people/lenb/acpi/doc/OLS2006-ondemand-paper.pdf
3. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors - http://support.amd.com/us/Processor_TechDocs/31116.pdf
4. Power management architecture of the 2nd generation Intel® Core™ microarchitecture, formerly codenamed Sandy Bridge - *Efi Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, Eli Weissmann* – Hot Chips august 2011 Conference

Backup

MPI profiling interface

MPI profiling interface

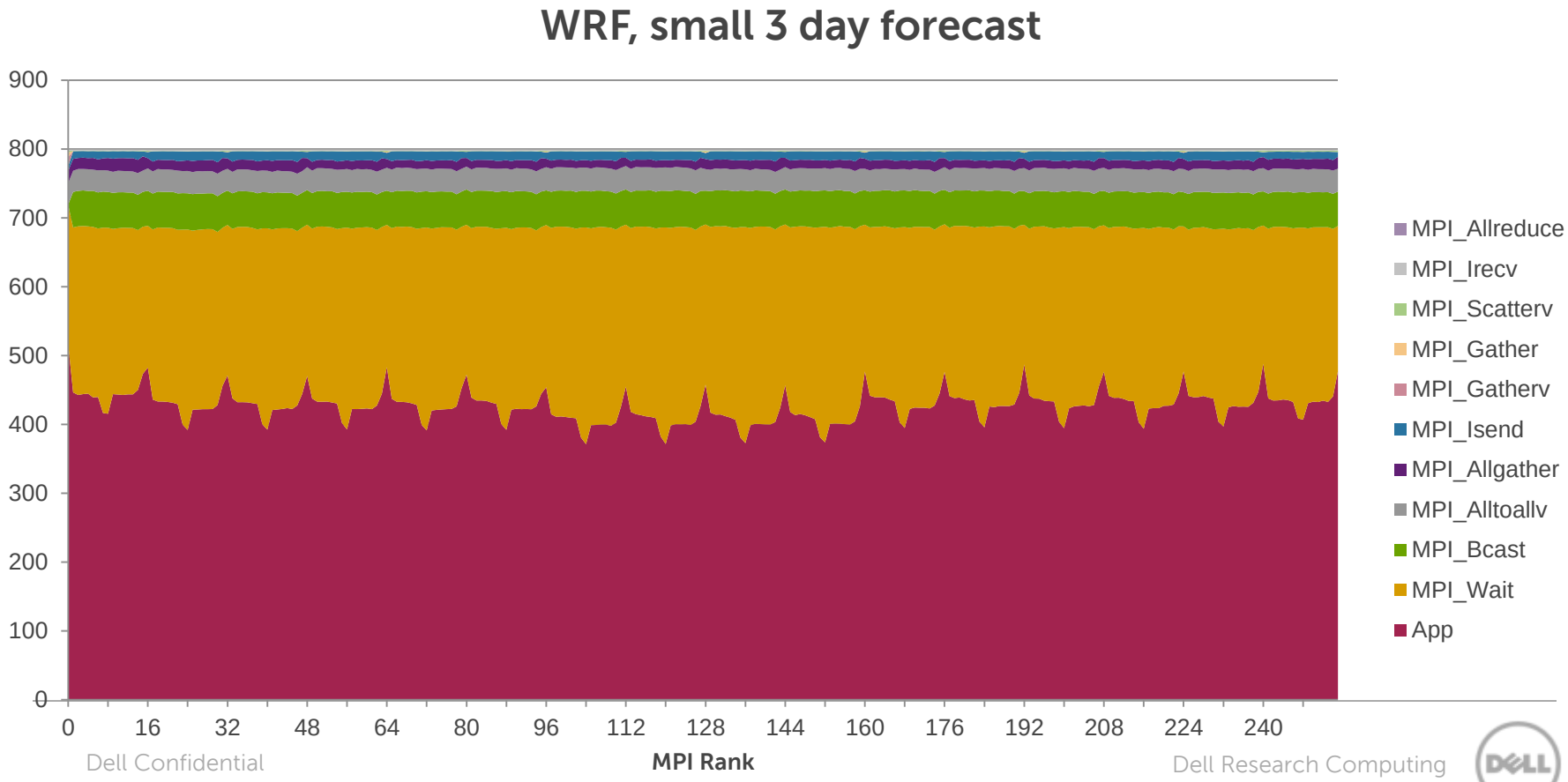
- dell_toolbox.so also contains an MPI profiling interface
  - No need to relink your application, plugin-and-play!
  - Provides insight in computation/computation ratio
  - Allows to dig deeper into MPI internals

- Usage:

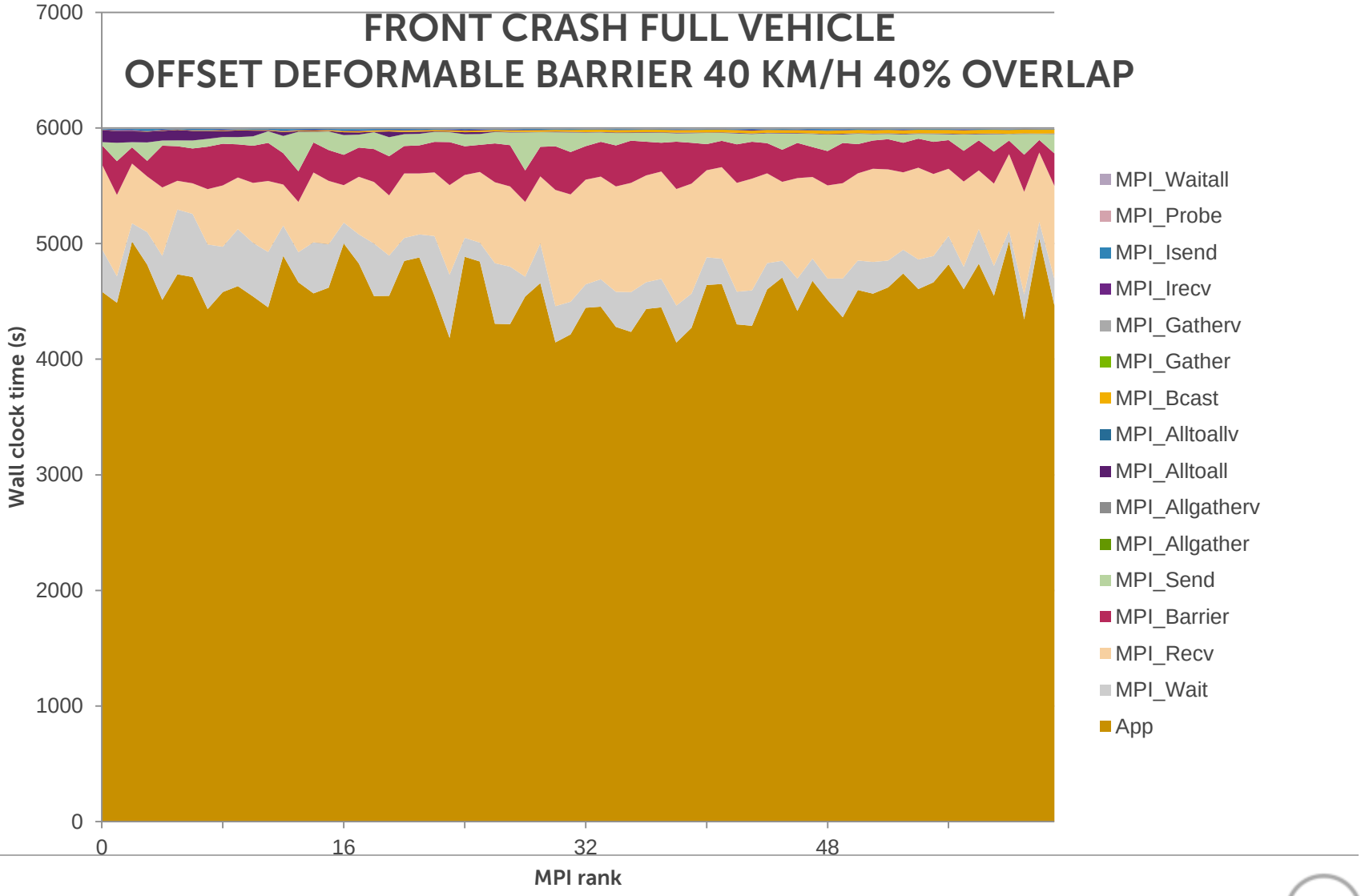
```
% export LIBTOOLBOX="-s"
% mpirun -np 128 -e LD_PRELOAD=dell_toolbox.so dell_affinity.exe <app>
```

MPI profiling interface

- The output generated is a comma separated file (CSV)
  - Can easily be imported in the Microsoft Excel
  - Dell also has a macro available to automatically generate graphs

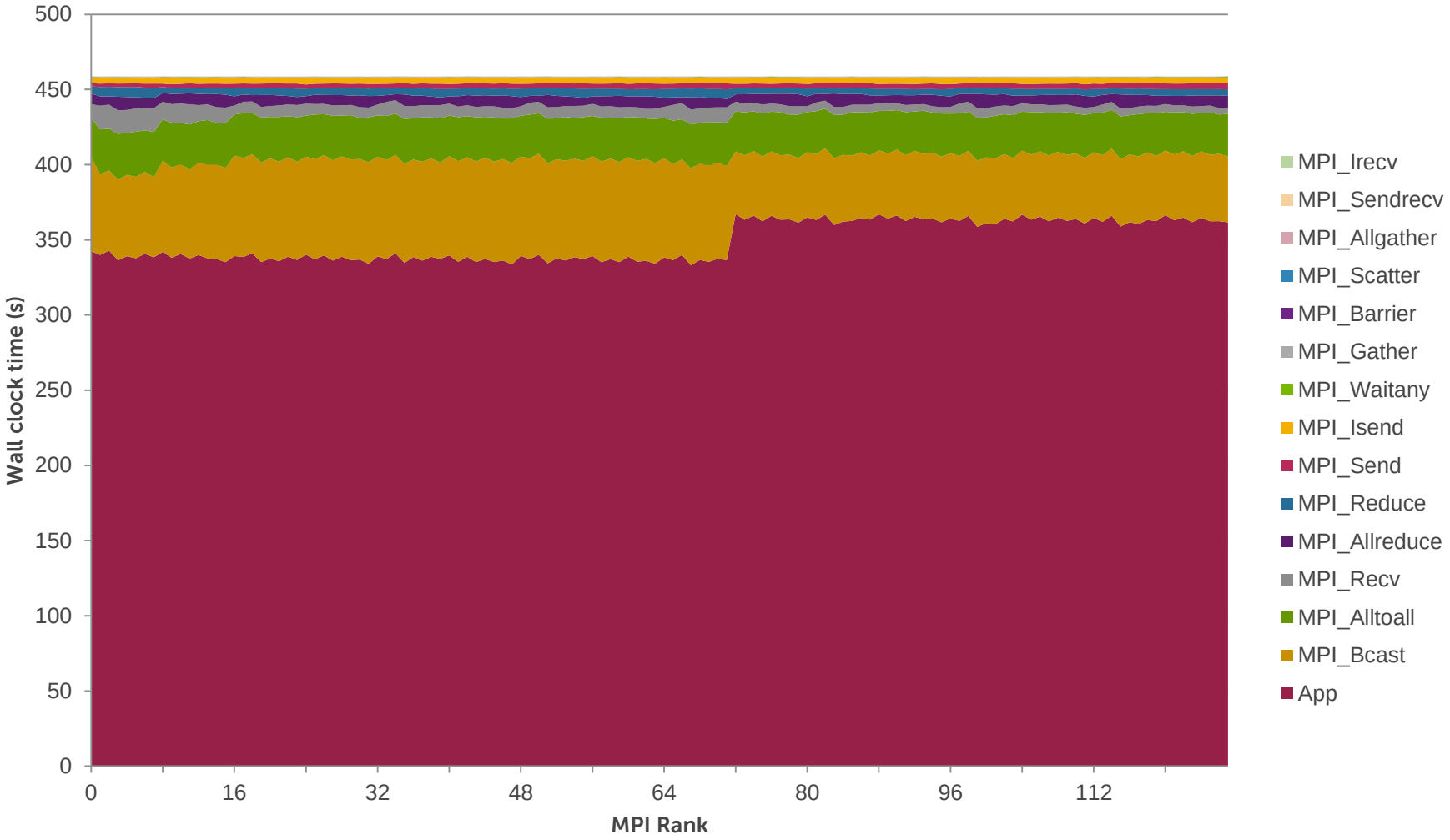


LS-DYNA


FRONT CRASH FULL VEHICLE
OFFSET DEFORMABLE BARRIER 40 KM/H 40% OVERLAP
Wall clock time (s)
7000
6000
5000
4000
3000
2000
1000
0
0
16
32
48
MPI rank
MPI_Waitall
MPI_Probe
MPI_Isend
MPI_Irecv
MPI_Gatherv
MPI_Gather
MPI_Bcast
MPI_Alltoallv
MPI_Alltoall
MPI_Allgatherv
MPI_Allgather
MPI_Send
MPI_Barrier
MPI_Recv
MPI_Wait
App

CP2k

CP2k 1024 Water molecules 128 core message profile



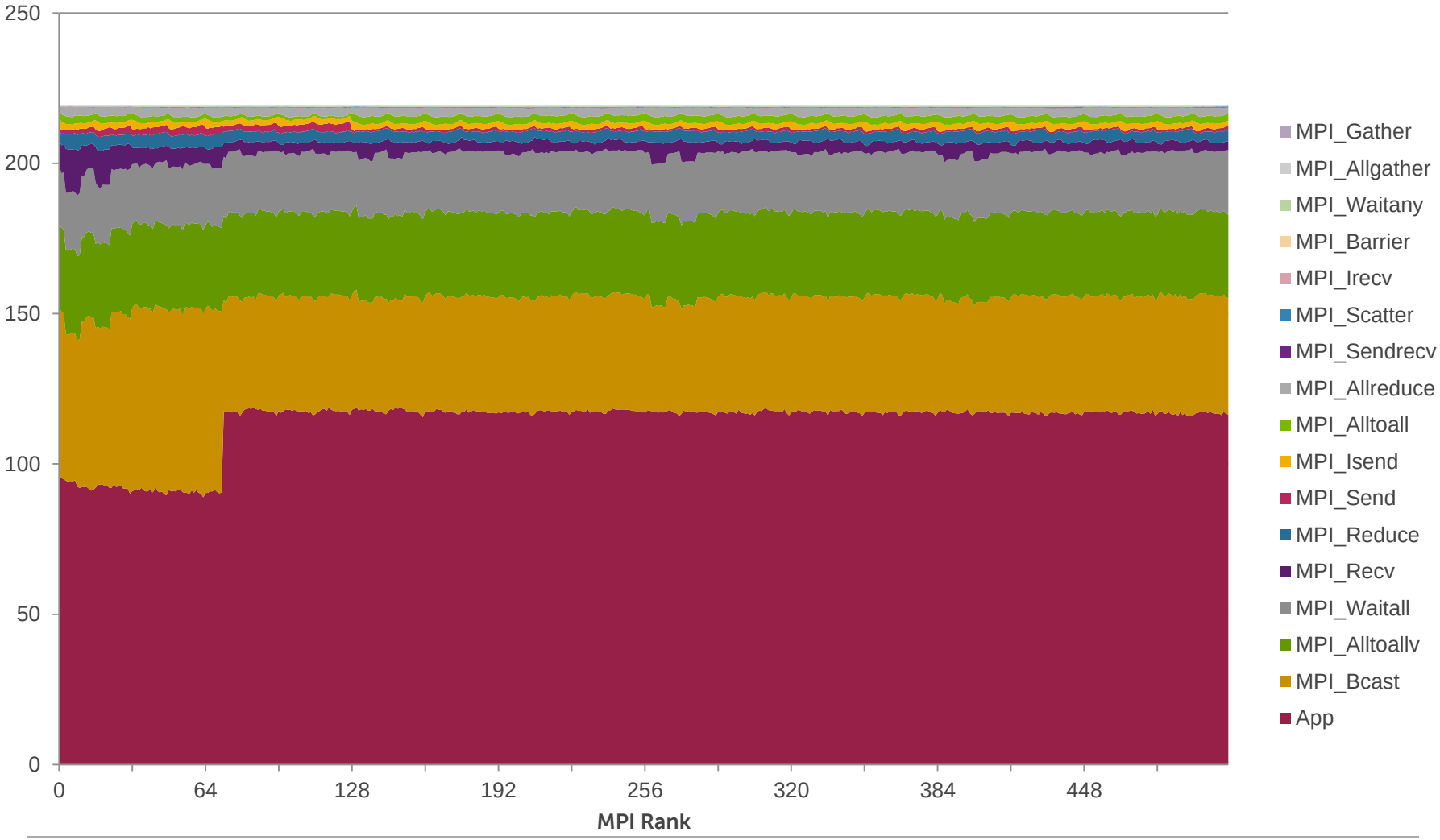
MPI profiling interface

- dell_toolbox.so can also count message sizes
  - Captured into a CSV file
  - Provides insight in network bandwidth usage and communication pattern

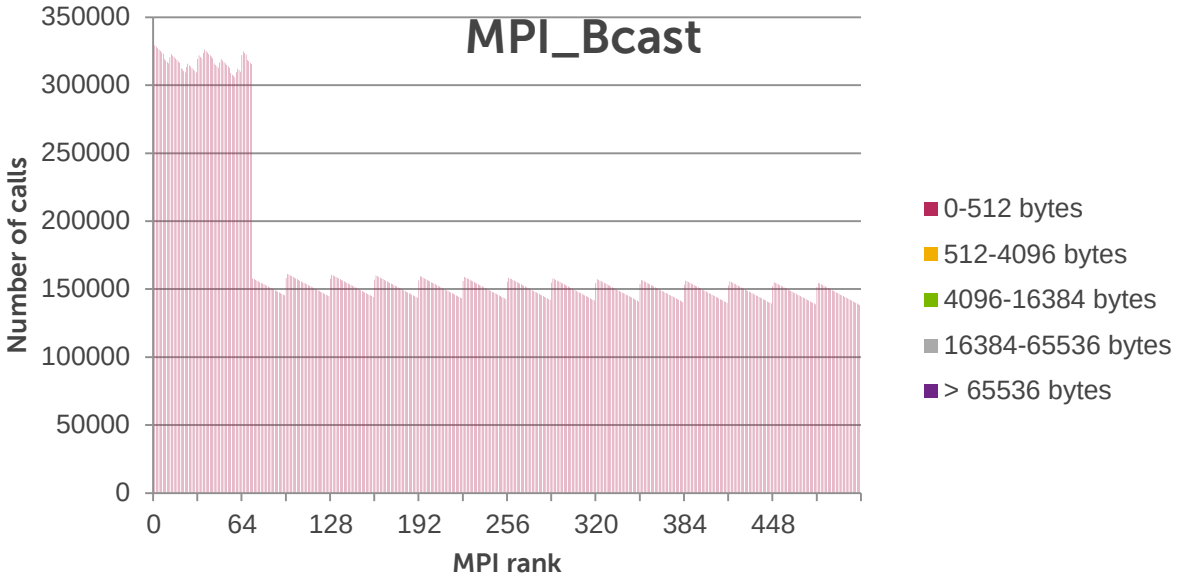
```
% export LIBTOOLBOX="-s"
% mpirun -np 128 -e LD_PRELOAD=dell_toolbox.so dell_affinity.exe <app>

…

 **** **** ******  **  PROGRAM ENDED AT                  2013-09-06 07:35:23.270
 ***** ** ***  *** **   PROGRAM RAN ON        c432-903.stampede.tacc.utexas.edu
 **    ****   ******    PROGRAM RAN BY                                  martinh
 ***** **    ** ** **   PROGRAM PROCESS ID                                98923
  **** **  *******  **  PROGRAM STOPPED IN  /scratch/01760/martinh/cp2k/1657750

libtoolbox: Wrote file libtoolbox_time.98923.csv

libtoolbox: Wrote file libtoolbox_msg_stat.98923.csv
```

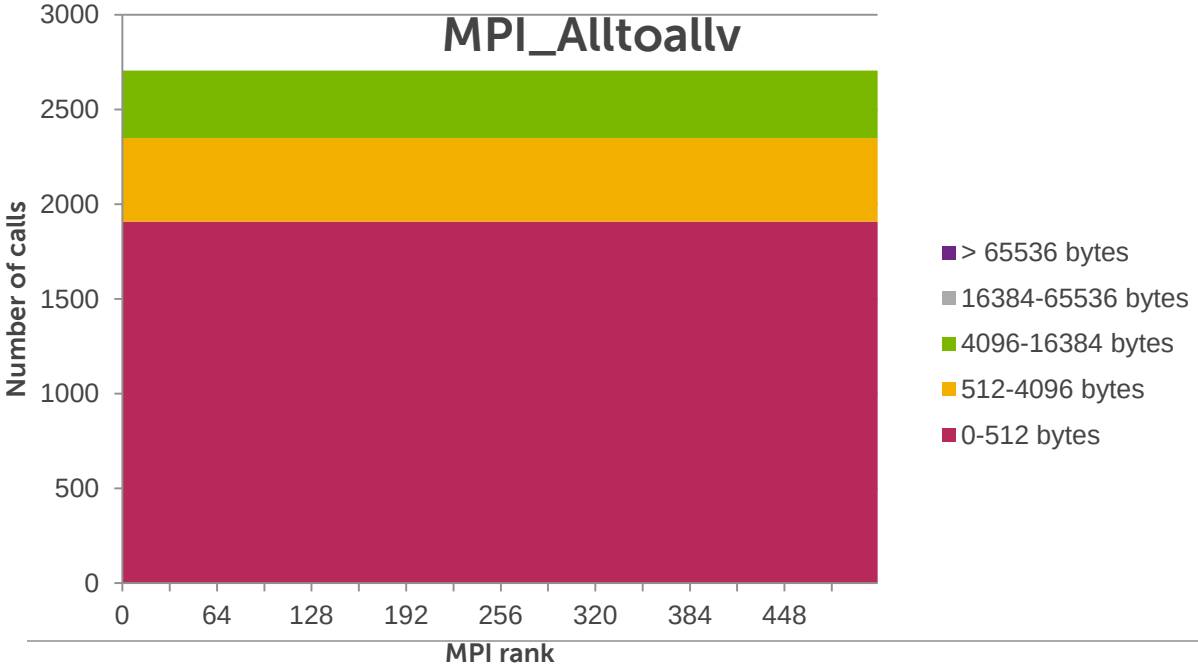
CP2k

CP2k 1024 Water molecules 512 core message profile



CP2k


MPI_Bcast
Number of calls
350000
300000
250000
200000
150000
100000
50000
0
0
64
128
192
256
320
384
448
MPI rank
0-512 bytes
512-4096 bytes
4096-16384 bytes
16384-65536 bytes
> 65536 bytes


MPI_Alltoallv
Number of calls
3000
2500
2000
1500
1000
500
0
0
64
128
192
256
320
384
448
MPI rank
> 65536 bytes
16384-65536 bytes
4096-16384 bytes
512-4096 bytes
0-512 bytes

The power to do more