Power Savings - the quest for green MPI

MVAPICH2-X case study

Martin Hilgeman

HPC Consultant EMEA

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)

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
```

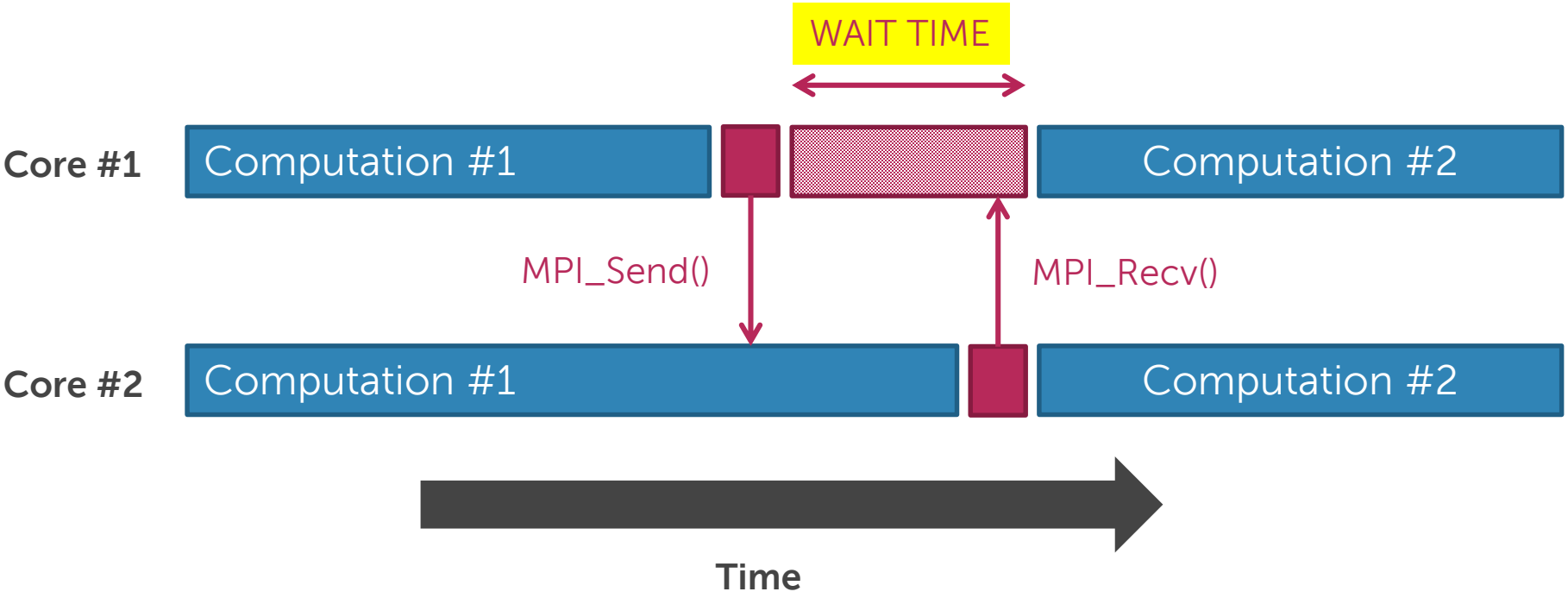
Or…

```
#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!

So what is happening?



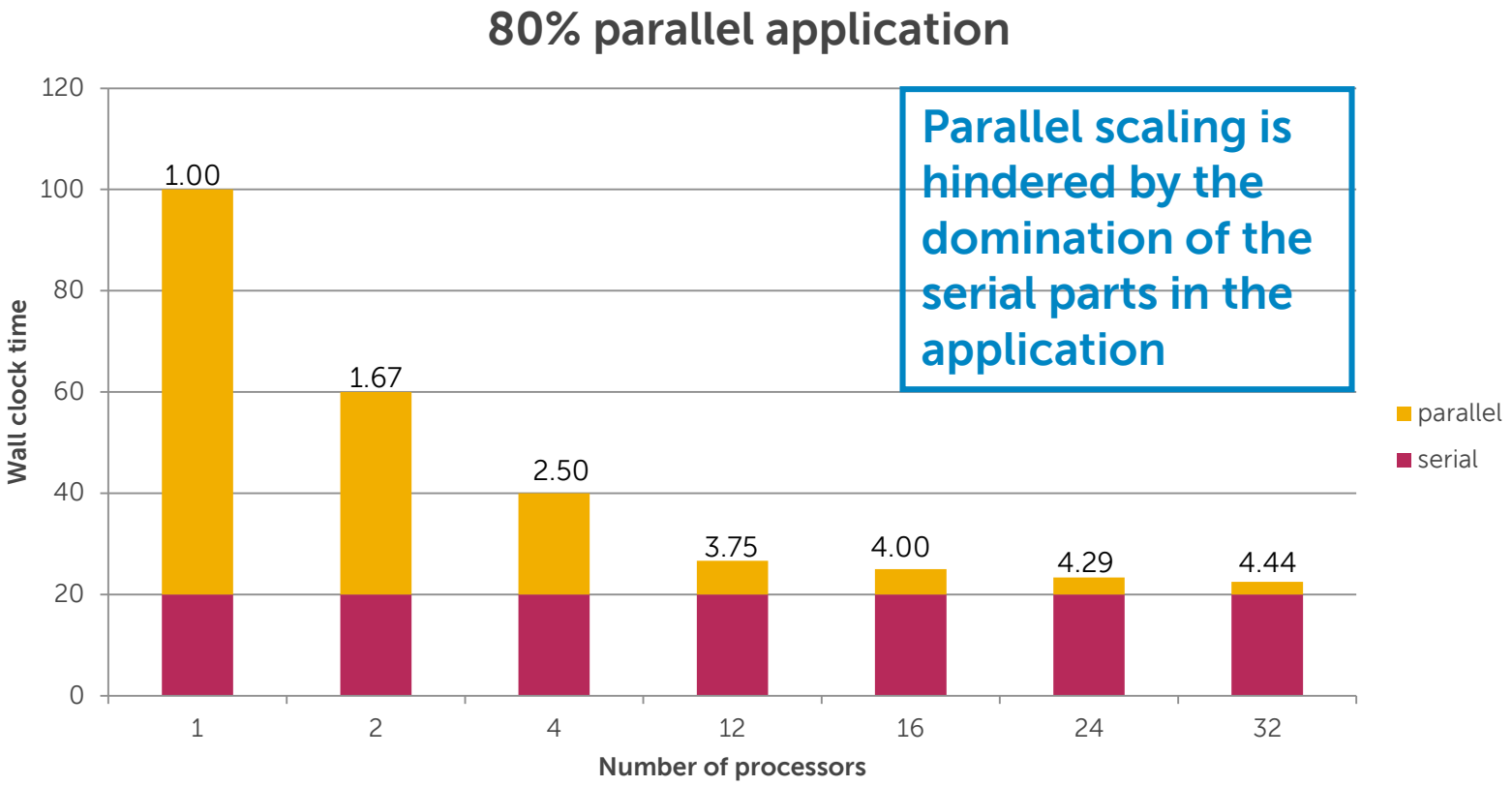
- Core #1 is done with its computation and sends a message to Core #2
- Core #2 is still busy with its computation
- Core #1 has to wait for the receive acknowledgement from Core #2
- While doing that, Core #1 is waiting and keeps polling the network interface for incoming messages

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

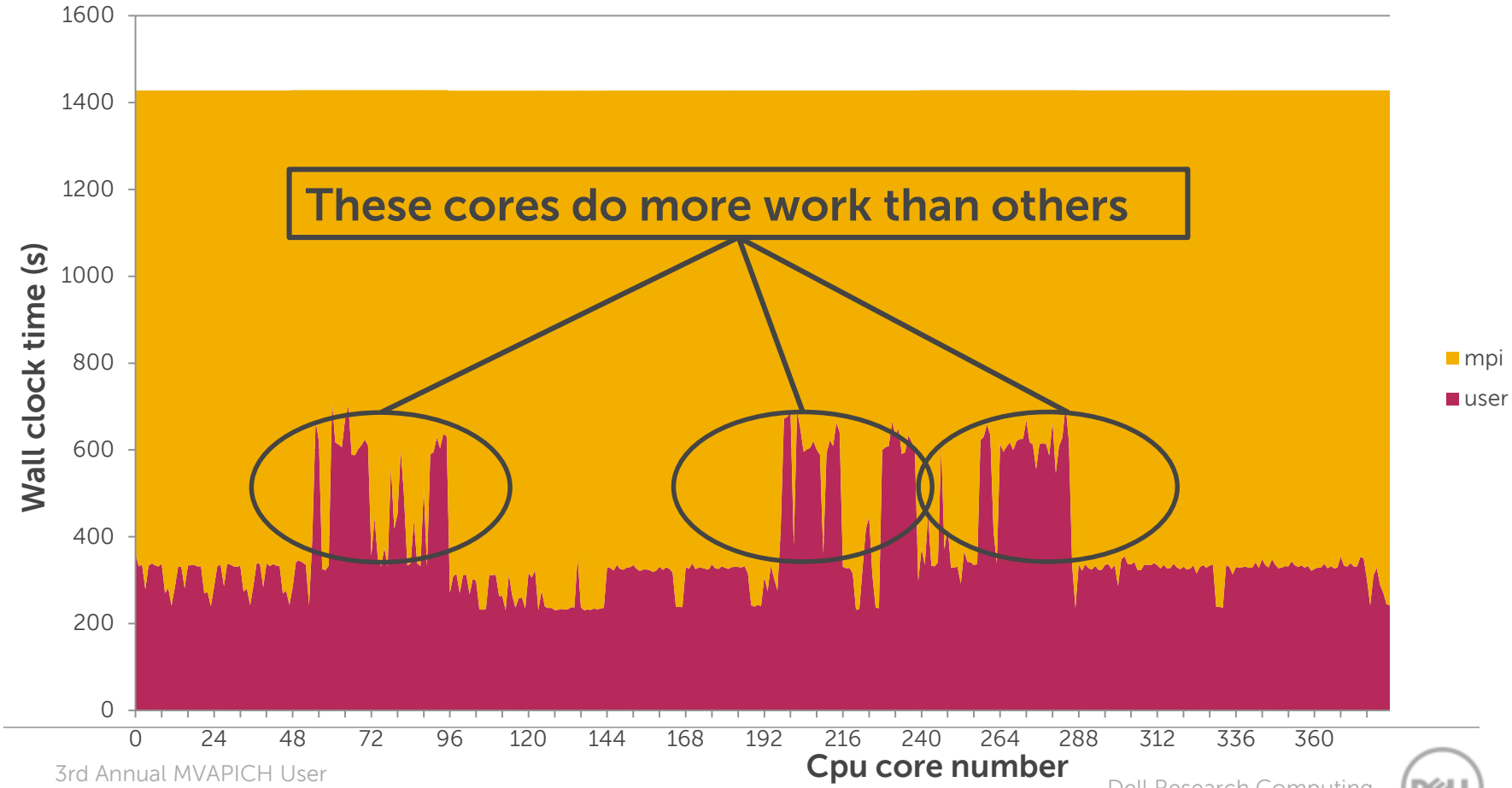
The reality is that: #1

- 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



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?

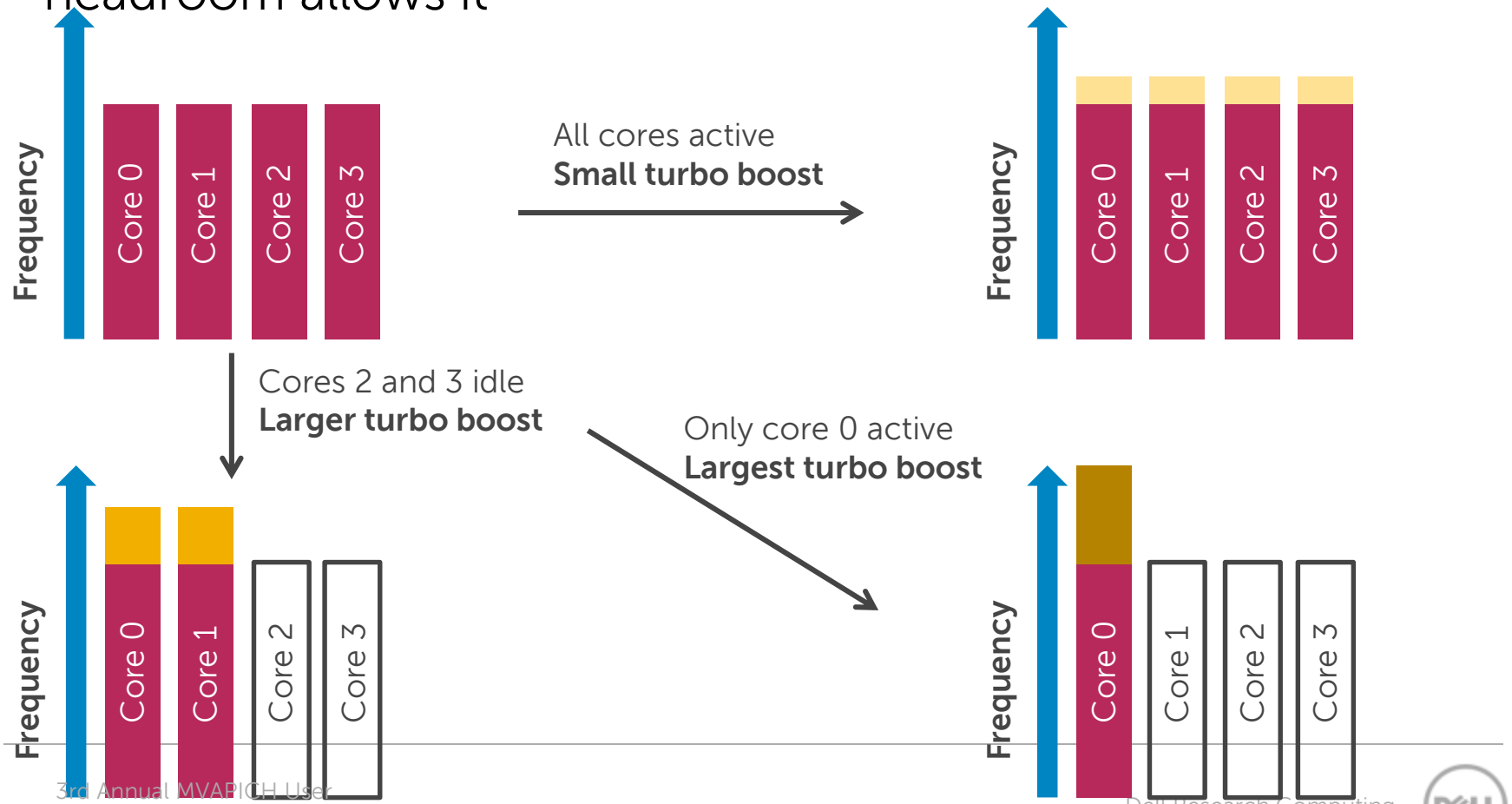
Reduce power: Processor P-States

- Setting the processor into a higher P-State lowers its power consumption
- 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

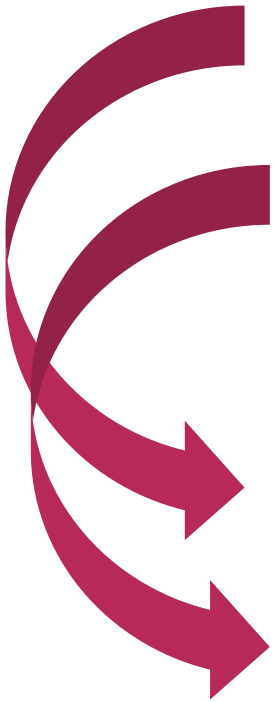
The Linux kernel has performance governors that (allow) control of the P-States on a per core level

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



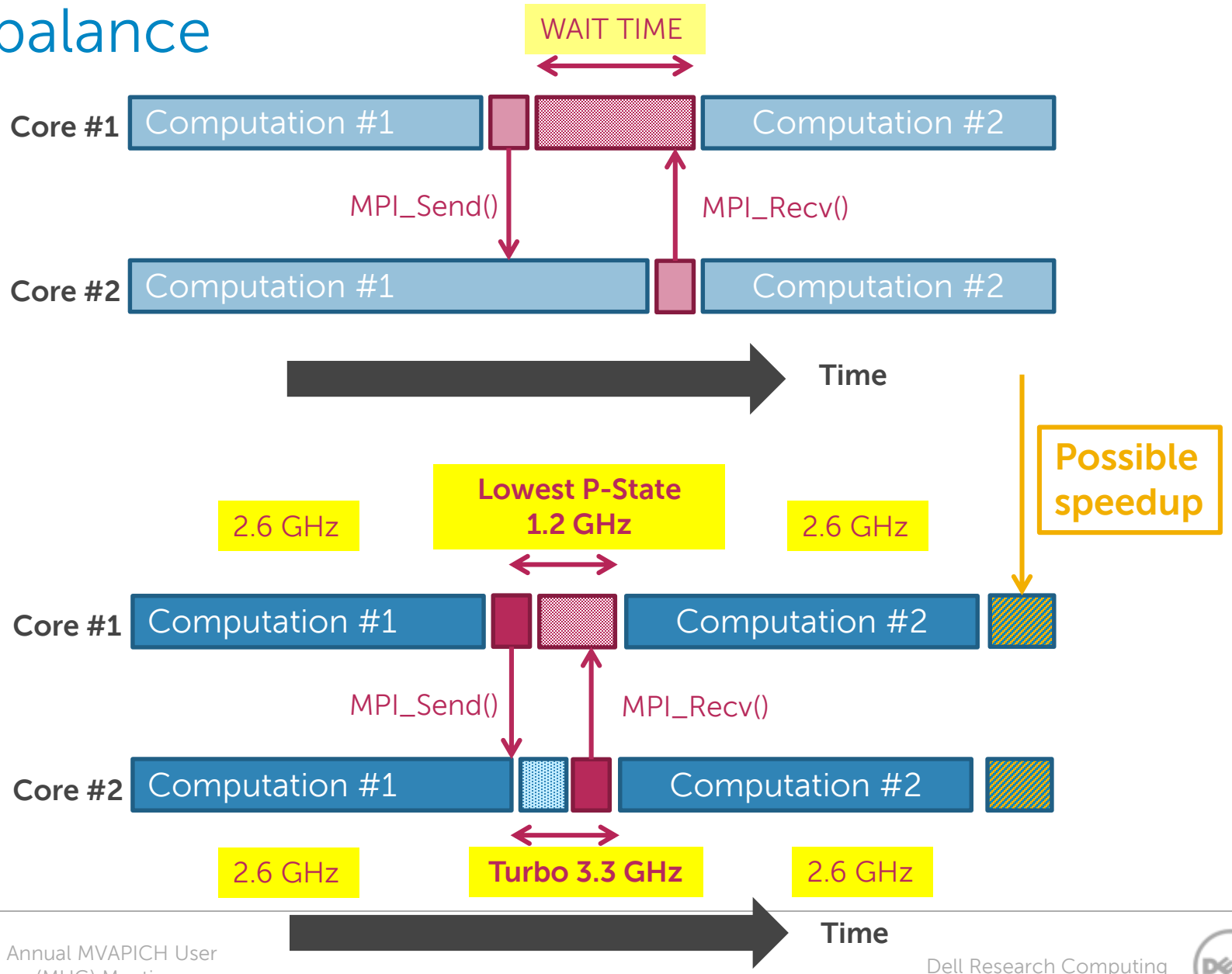
Goals revisited



- 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: **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 ☺

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()
- Identify 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;
}
```

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

- **Yielding the CPU only makes sense when there are other eligible tasks to run, not very common in HPC**

P-state switching latency

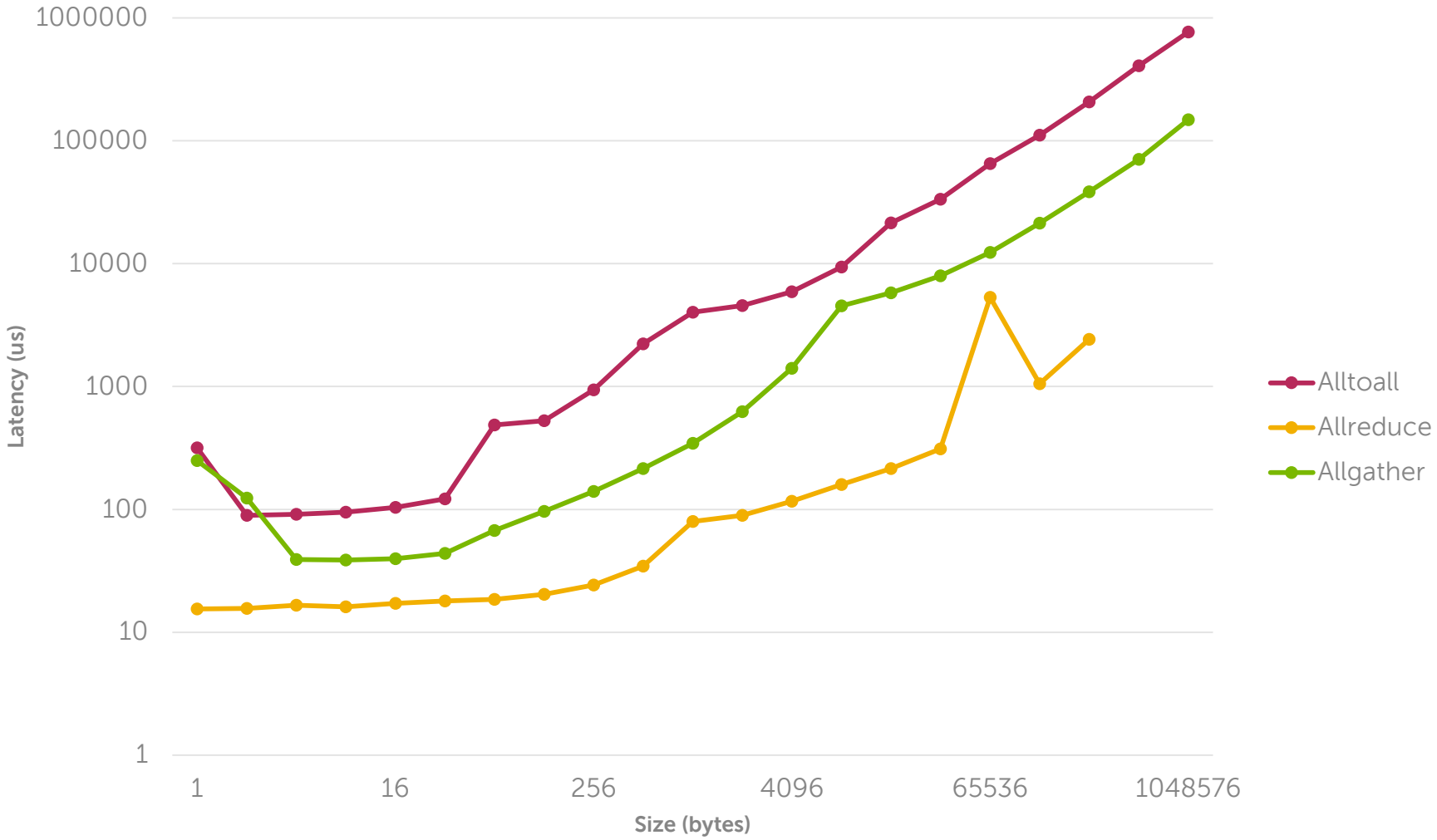
- Switching from one P-state to another involves a latency
  - E5-2670 8C 2.7 GHz

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

  - E5-2670 v3 12C 2.5 GHz

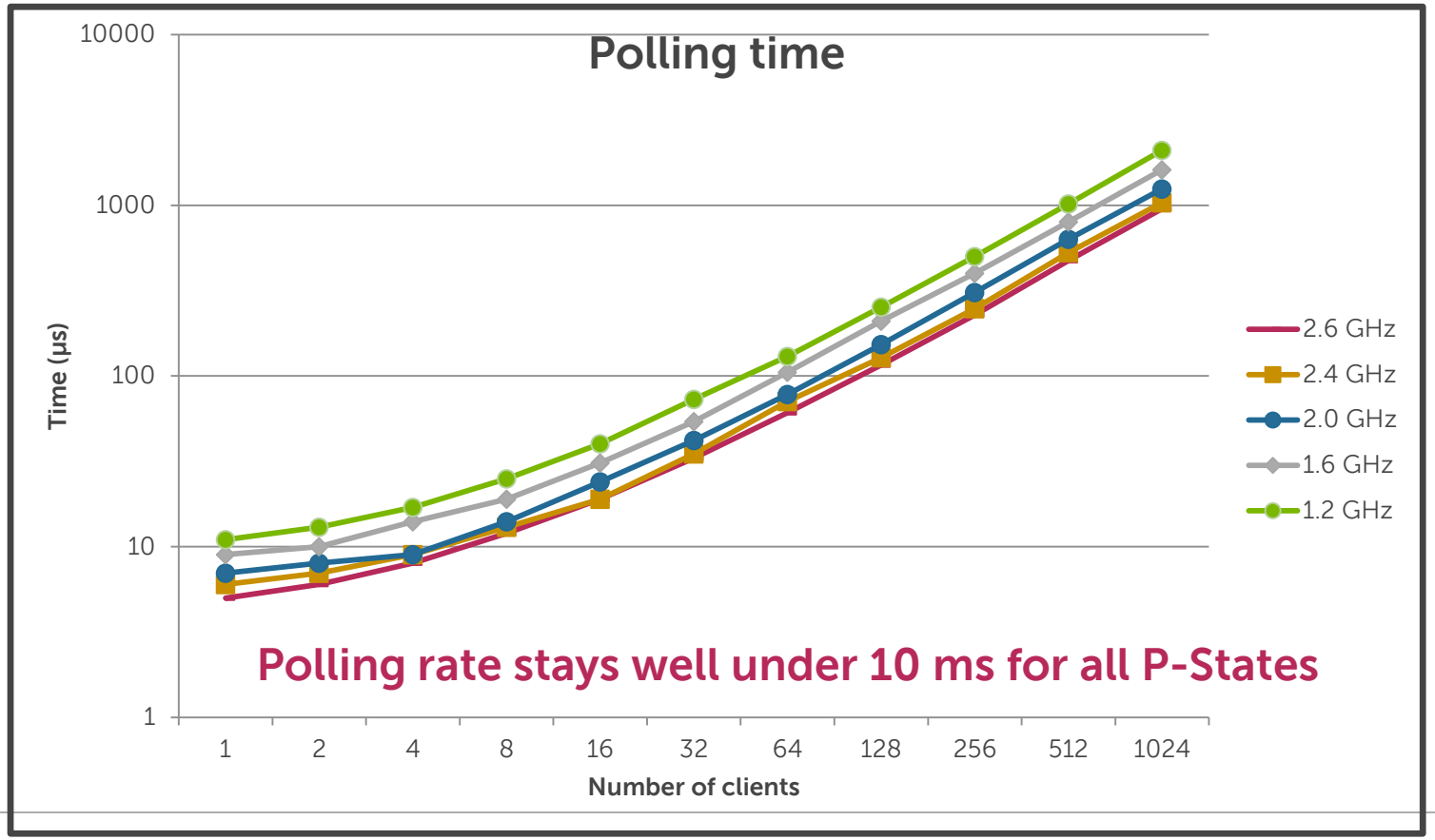
From 2.5 GHz to...	Min (µs)	Max (µs)	Average (µs)
2.4 GHz	4.05	19.31	5.18
2.1 GHz	4.05	15.73	5.07
1.5 GHz	4.05	9.53	4.32
1.2 GHz	4.05	9.29	4.27

Collective MPI function latency



Does the lowest P-State degrade performance?

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



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 August 2015)

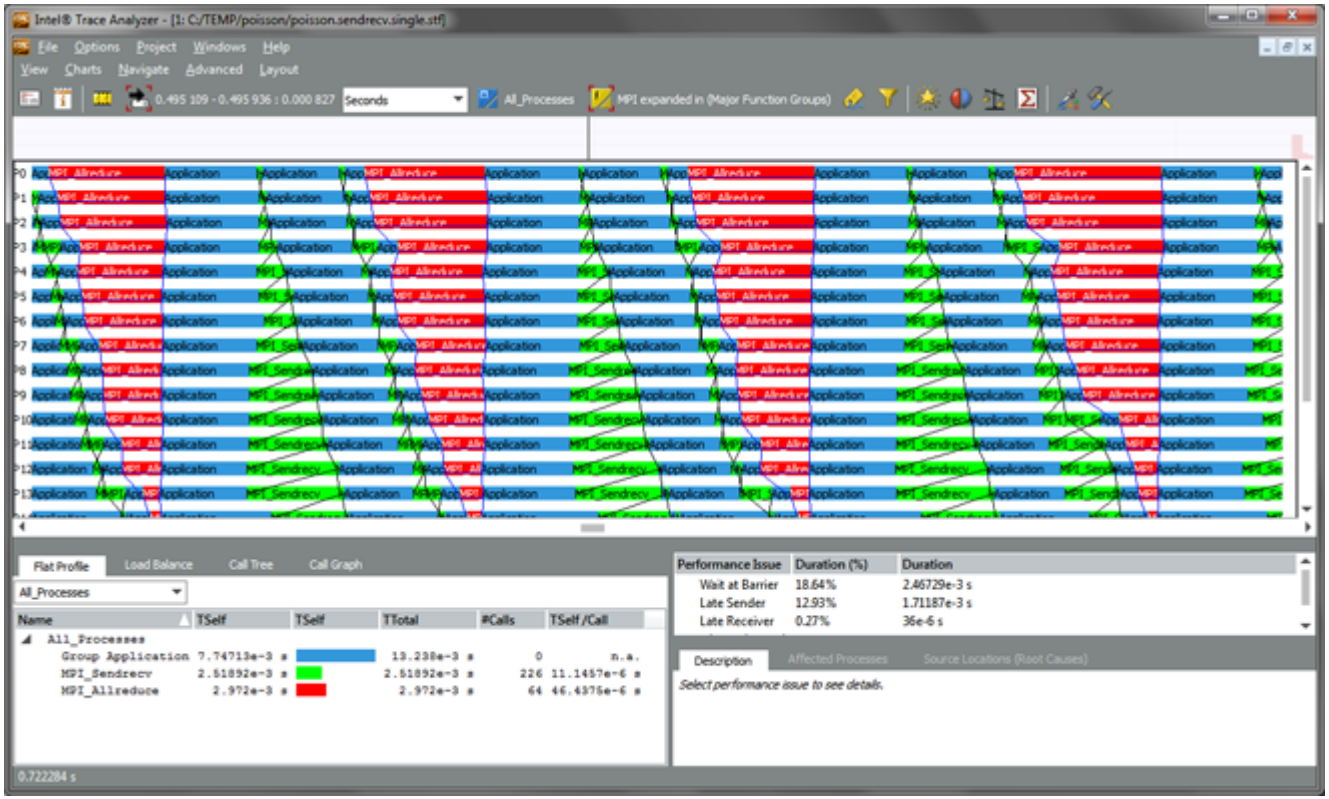
- Library works with MVAPICH2, Intel MPI, Platform MPI, Open MPI and (tested), should also work with others
- Application testing done with WRF, CP2k, VASP, LS-DYNA, STAR-CCM+, ANSYS FLUENT, CP2k, and GROMACS
- Performance looks OK-ish, but it is too early to make any judgment (lots of room for tuning)
- The Fortran Interface to MPI is a bit too CPU intensive, should look into making the wrappers more efficient
- Basic power measurements have been done with Intel PCM

To do list (as of August 2015)

- 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
- Make the library workload adaptive

Communication patterns explained

- Almost all HPC applications use some kind of iterative algorithm



MVAPICH2-X example

MVAPICH2-X case study

- In house application to post-process 3D textures for analysis of rock samples
- Sequential example code given to benchmark
- Very large data sizes (>20 GB input file, > 8M textures), runs for **99 years** on a single core till completion
- Code labels 128x128x128 pixel textures in memory and calculates the overlap

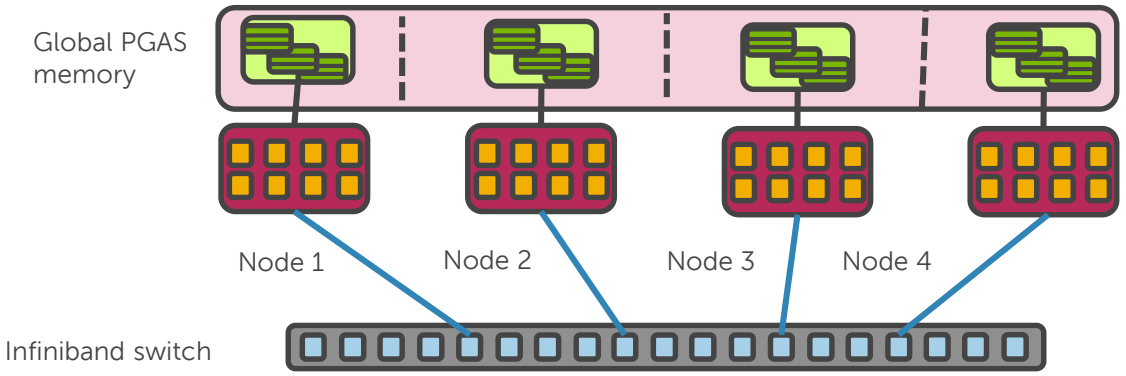
```
for(iz=-tw/2; iz < tw/2 ; iz++) {
    for(iy=-tw/2; iy < tw/2 ; iy++) {
        for(ix=-tw/2; ix < tw/2 ; ix++) {
            /* Copy texture into buffer */
            buf[(iz+tw/2)*tw*tw + (iy+tw/2)*tw + ix + tw/2] = image[(z+iz)*dimx*dimy +
(y+iy)*dimx + (x+ix)];

            /* Label the texture */
            label_texture(buf, tw, tw, tw, val_min, val_max, nl, nb, bodies, matrix_a,
matrix_a_b, matrix_a_c, matrix_a_d);
        }
    }
}
```

Parallel solution

- The sequential problem has to be parallelized to speedup the processing
- The labeling function can be make multi-threaded through OpenMP
- What to do with the textures themselves?

Parallelizing the textures with MVAPICH2-x

- Parallelized the code with MPI+PGAS to load the data structures in globally distributed memory across nodes.



- Challenges
  - MVAPICH2-x was a very young project at the time (1.9b)
  - Sensitive to the exact match of the OFED release installed on the cluster and used for the MVAPICH2-x binaries
  - Luckily there was a version available specifically compiled for Stampede's old OFED (1.5.4)

The solution – allocate symmetric memory and define boundaries

```
/*
 * Allocate the image array in symmetric memory
 */
g_image_size = taille;
lmod = g_image_size % npes;
while (g_image_size % npes != 0)
    g_image_size++;

g_image_block = g_image_size / (long) npes;
image = (unsigned char *) shmalloc(g_image_block * sizeof(unsigned char)) ;

/*
 * Calculate offsets and do first touch of the elements
 */
g_image_start = (long *) malloc(npes * sizeof(long));
g_image_end = (long *) malloc(npes * sizeof(long));
image_block = (long *) malloc(npes * sizeof(long));
for (i = 0; i < npes; i++) {
    g_image_start[i] = i * g_image_block;
    g_image_end[i] = g_image_start[i] + g_image_block - 1;
}
for(ind = g_image_start[me]; ind < g_image_end[me]; ind ++)
    image[ind]=0;

/*
 * Calculate offsets for the actual data blocks
 */
lblock = taille / (long) npes;
lmod = taille % npes;
if (lmod != 0 && lmod >= me + 1) {
    lblock ++;
}
ret = MPI_Allgather(&lblock, 1, MPI_LONG, image_block, 1, MPI_LONG, MPI_COMM_WORLD);
```

The solution – define remote pointers and domain decomposition

```
/*
 * Read the input data into symmetric memory
 */
if (me == 0) {
    for (i = 0; i < npes; i++) {
        ptr = shmem_ptr(&image[g_image_start[i]], i);
        ret = read(fpd, ptr, sizeof(unsigned char) * image_block[i]);
    }
    shmem_quiet();
    close(fpd) ;
}

/*
 * Domain decomposition
 */
gstart = (int *) malloc(npes * sizeof(int));
gend = (int *) malloc(npes * sizeof(int));
giblock = (int *) malloc(npes * sizeof(int));
iblock = tw / npes;
lmod = tw % npes;
if (lmod != 0 && lmod >= me + 1) iblock ++;

ret = MPI_Allgather(&iblock, 1, MPI_INT, giblock, 1, MPI_INT, MPI_COMM_WORLD);
j = -tw2;
gstart[0] = j;
gend[0] = gstart[0] + giblock[0] - 1;
for (i = 1; i < npes; i++) {
    j += giblock[i-1];
    gstart[i] = j;
    gend[i] = gstart[i] + giblock[i] - 1;
}
len = giblock[me] * tw * tw;
```

The solution – find the PE holding the object and label texture

```
for(z = tw2; z < dimz-tw2+1; z += 4) {
    for(y = tw2; y < dimy-tw2+1; y += 4) {
        for(x = tw2; x < dimx-tw2+1; x += 4) {
            istart = (z + gstart[me]) * dimx * dimy + (y + -tw2) * dimx;
            /*
             * Find the PE of the remote data object
             */
            pe = istart / g_image_block;
            shmem_getmem(buf, &image[istart], len * sizeof(unsigned char), pe);
            shmem_quiet();
            label_texture(buf, tw, tw, giblock[me], val_min, val_max,nl, nb, bodies, matrix_a,
matrix_b, matrix_c, matrix_d) ;
        }
    }
}

shmem_barrier_all();
shfree(image);
```

- Run line:

```
% export OOSHM_SYMMETRIC_HEAP_SIZE=3172M
% export OMP_NUM_THREADS=4
% ibrun -np 256 dell_affinity.exe ${EXE}
```

Results

#MPI tasks	# OMP threads	# number of cores	Label time (s)	Speedup
1	1	1	85.698	1.00
1	4	4	38.175	2.24
32	4	128	0.632	135
64	4	256	0.177	484
128	4	512	0.076	1128
256	4	1024	0.027	3174
512	4	2048	0.027	3174
1024	4	4096	0.027	3174

The power to do more