Methods of Power Optimization of High Performance Computing Systems

Martin Hilgeman
HPC Consultant EMEA
Disclaimer

Some of this 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 be used in any customer environment (at this time)
- The clock speed plateau
- The power ceiling
- IPC limit

- Industry is applying Moore’s Law by adding more cores
- Meanwhile Amdahl’s Law says that you cannot use them all efficiently

**Moore’s Law vs. Amdahl’s Law**

**Figure:**
- Diminishing returns:
  Tension between the desire to use more processors and the associated “cost”

**Mathematical Equation:**
\[ e = - \left( \frac{1}{p} - n \right) \left( \frac{1}{n} - 1 \right) \]

**Source:** Chuck Moore, “DATA PROCESSING IN EXASCALE-CLASS COMPUTER SYSTEMS”, The Salishan Conference on High Speed Computing, 2011
Moore’s Law vs Amdahl’s Law - “too Many Cooks in the Kitchen”

Industry is applying Moore’s Law by adding more cores

Meanwhile Amdahl’s Law says that you cannot use them all efficiently
What levels do we have*?

- Challenge: Sustain performance trajectory without massive increases in cost, power, real estate, and unreliability
- Solutions: **No single answer**, must **intelligently turn** “Architectural Knobs”

\[
\text{Hardware performance} = (\text{Freq}) \times \left(\frac{\text{cores}}{\text{socket}}\right) \times (\#\text{sockets}) \times \left(\frac{\text{inst or ops}}{\text{core} \times \text{clock}}\right) \times (\text{Efficiency})
\]

*Slide previously from Robert Hormuth
Turning the knobs 1 - 4

1. Frequency is unlikely to change much. Thermal/Power/Leakage challenges.


3. Number of sockets per system is the easiest knob. Challenging for power/density/cooling/networking.

Tuning knobs for performance

Hardware tuning knobs are limited, but there’s far more possible in the software layer.
Monitoring an 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

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

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...

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>P</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>32500</td>
<td>martinh</td>
<td>20</td>
<td>0</td>
<td>1090m</td>
<td>787m</td>
<td>17m</td>
<td>R</td>
<td>100.0</td>
<td>1.2</td>
<td>6:43.74</td>
<td>11</td>
<td>my_application</td>
</tr>
</tbody>
</table>

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 opal_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 opal_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  0x0000000000007f50f8620909 in opal_progress () at runtime/opal_progress.c:189
#5  0x0000000000007f50f857205 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  0x0000000000007f50f859686d in PMPI_Wait (request=0x7fff54a2ac08, status=0x7fff54a2abf0) at pwait.c:70
#9  0x0000000000007f50f859686d in mpi_wait_f (request=0x7fff54a2ac54, status=0xa60a420, ierr=0x7fff54a2ac50) at pwait_f.c:66
#10 0x0000000000007f50f859686d in my_wait__ ()

The bottom line:

“waiting” can be very CPU intensive!
# Waiting is latency

## Latency Numbers Every Programmer Should Know

<table>
<thead>
<tr>
<th>Latency Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ns</td>
<td>L1 cache reference: 0.5 ns</td>
</tr>
<tr>
<td></td>
<td>Branch mispredict: 5 ns</td>
</tr>
<tr>
<td></td>
<td>L2 cache reference: 7 ns</td>
</tr>
<tr>
<td></td>
<td>Mutex lock/unlock: 25 ns</td>
</tr>
<tr>
<td></td>
<td>= 100 ns</td>
</tr>
<tr>
<td>1 µs</td>
<td>Main memory reference: 100 ns</td>
</tr>
<tr>
<td></td>
<td>Compress 1 KB with Zippy: 3 µs</td>
</tr>
<tr>
<td></td>
<td>= 1 µs</td>
</tr>
<tr>
<td>10 µs</td>
<td>Send 1 KB over 1 Gbps network: 10 µs</td>
</tr>
<tr>
<td></td>
<td>SSD random read (100 Gbps SSD): 150 µs</td>
</tr>
<tr>
<td></td>
<td>Read 1 MB sequentially from memory: 250 µs</td>
</tr>
<tr>
<td></td>
<td>Round trip in same datacenter: 500 µs</td>
</tr>
<tr>
<td>1 ms</td>
<td>Read 1 MB sequentially from disk: 20 ms</td>
</tr>
<tr>
<td></td>
<td>Packet roundtrip OR to Netherlands: 150 ms</td>
</tr>
<tr>
<td></td>
<td>Disk seek: 10 ms</td>
</tr>
</tbody>
</table>

Source: https://gist.github.com/2
To put it differently

L1 cache reference .................................................. 0.5 ns  
L2 cache reference .................................................. 7 ns  
Main memory reference ............................................. 100 ns  
Send 2K bytes over 1 Gbps network ...................... 20,000 ns = 20 μs  
SSD random read ..................................................... 150,000 ns = 150 μs  
Read 1 MB sequentially from memory ............. 250,000 ns = 250 μs  
Round trip within same datacenter .............. 500,000 ns = 0.5 ms  
Read 1 MB sequentially from SSD* ............. 1,000,000 ns = 1 ms  
Disk seek ............................................................. 10,000,000 ns = 10 ms  
Read 1 MB sequentially from disk .......... 20,000,000 ns = 20 ms  
Send packet CA->Netherlands->CA .... 150,000,000 ns = 150 ms  

Adapted from: https://gist.github.com/2843375
Waiting as a service

• Providing a service means waiting when there are no "customers"

• How would you like to wait?

or
So what is happening inside my application?

- 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)

![80% parallel application diagram]

Parallel scaling is hindered by the domination of the serial parts in the application.
The reality is that: #2

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

![Graph showing CPU core number vs. Wall clock time (s) with colored bars for mpi and user showing some cores do more work than others.](image)
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.

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency: Small turbo boost</td>
<td>Frequency: Larger turbo boost</td>
<td>Frequency: Largest turbo boost</td>
<td></td>
</tr>
</tbody>
</table>

- All cores active
- Cores 2 and 3 idle
- Only core 0 active
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()
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()`

<table>
<thead>
<tr>
<th>MPI library</th>
<th>Spin/Wait type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open MPI</td>
<td>Polls every 10 ms, then is able to yield the CPU</td>
</tr>
<tr>
<td>Intel MPI</td>
<td>Spins 250 times, then yields the CPU</td>
</tr>
<tr>
<td>Platform MPI</td>
<td>Spins 10,000 times, then yields the CPU</td>
</tr>
<tr>
<td>MVAPICH</td>
<td>Spins 2,000 times, then is able to yield the CPU</td>
</tr>
</tbody>
</table>

• 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**

<table>
<thead>
<tr>
<th>From 2.7 GHz to...</th>
<th>Min (μs)</th>
<th>Max (μs)</th>
<th>Average (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6 GHz</td>
<td>4.29</td>
<td>17.40</td>
<td>5.87</td>
</tr>
<tr>
<td>2.2 GHz</td>
<td>4.53</td>
<td>18.12</td>
<td>6.87</td>
</tr>
<tr>
<td>1.6 GHz</td>
<td>4.29</td>
<td>25.03</td>
<td>7.40</td>
</tr>
<tr>
<td>1.2 GHz</td>
<td>4.53</td>
<td>23.13</td>
<td>8.24</td>
</tr>
</tbody>
</table>

- **E5-2670 v3 12C 2.5 GHz**

<table>
<thead>
<tr>
<th>From 2.5 GHz to...</th>
<th>Min (μs)</th>
<th>Max (μs)</th>
<th>Average (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4 GHz</td>
<td>4.05</td>
<td>19.31</td>
<td>5.18</td>
</tr>
<tr>
<td>2.1 GHz</td>
<td>4.05</td>
<td>15.73</td>
<td>5.07</td>
</tr>
<tr>
<td>1.5 GHz</td>
<td>4.05</td>
<td>9.53</td>
<td>4.32</td>
</tr>
<tr>
<td>1.2 GHz</td>
<td>4.05</td>
<td>9.29</td>
<td>4.27</td>
</tr>
</tbody>
</table>
Collective MPI function latency

Latency (us) vs Size (bytes)

- Red: Alltoall
- Orange: Allreduce
- Green: Allgather

Dell Research Computing
## NAS parallel benchmark type D results

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

```c
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.
How does it work?

pe = shmem_getmem(buf, &image[istart], len * sizeof(unsigned char), pe);
To recap

1. Allocate a symmetric memory segment for the data
2. Do a first touch of the array so that it becomes “local” (think NUMA)
3. Calculate the offsets, so that you know your neighbors
4. Distribute the data over all the nodes
5. Call the compute kernel (the actual application)
6. Cleanup and exit
Next step

All steps are independent, which means that:

5. Call the compute kernel (the actual application)

Can be done with any device:

- CPU
- GPU
- Co-processor
- FPGA
How about performance

The goal was to label 8M textures in half an hour

<table>
<thead>
<tr>
<th>#MPI tasks</th>
<th># OMP threads</th>
<th># number of cores</th>
<th>Label time (s)</th>
<th>Time to solution (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>85.698</td>
<td>7935</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>38.175</td>
<td>3535</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>128</td>
<td>0.632</td>
<td>59</td>
</tr>
<tr>
<td>64</td>
<td>4</td>
<td>256</td>
<td>0.177</td>
<td>16</td>
</tr>
<tr>
<td>128</td>
<td>4</td>
<td>512</td>
<td>0.076</td>
<td>7</td>
</tr>
<tr>
<td>256</td>
<td>4</td>
<td>1024</td>
<td>0.027</td>
<td>3</td>
</tr>
<tr>
<td>512</td>
<td>4</td>
<td>2048</td>
<td>0.027</td>
<td>3</td>
</tr>
<tr>
<td>1024</td>
<td>4</td>
<td>4096</td>
<td>0.027</td>
<td>3</td>
</tr>
</tbody>
</table>
How about accelerators?

- NVIDIA and Nallatech were approached to write a kernel that offloads to an accelerator
- Both met the goal of 8 million cells in half an hour
  - NVIDIA used 438x K80 cards (300 W each)
  - Nallatech used 162x Stratix V A7 FPGA cards (25 W each)
NVIDIA solution – 4 racks

Power consumption: 168 kW, nominal load 116 kW
Nallatech solution – 2 racks

Power consumption: 13.7 kW
Dell Research Computing

The power to do more