Cray and the Quad-Core Experience

HECToR Town User Meeting

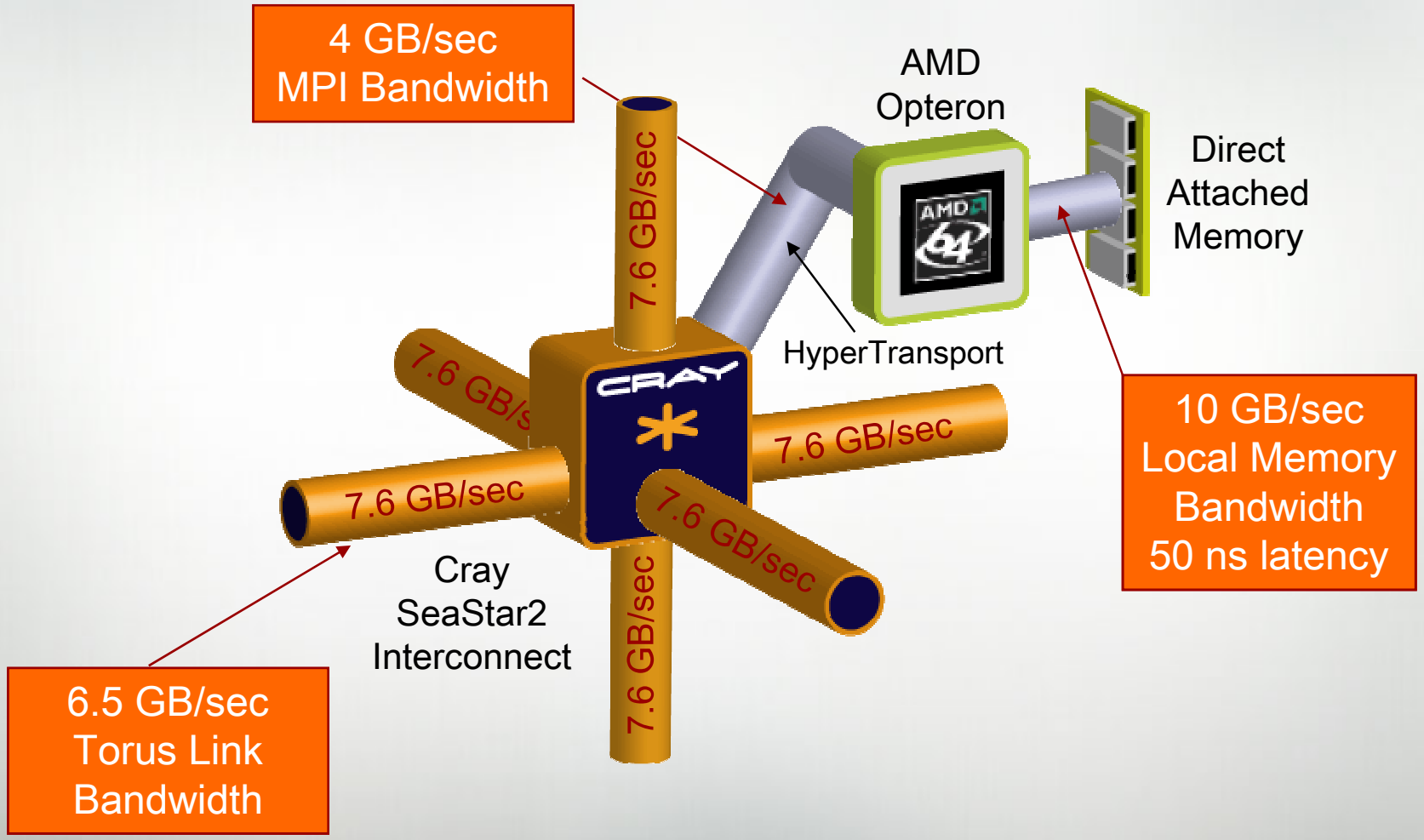
London

22 April 2009

Talk outline

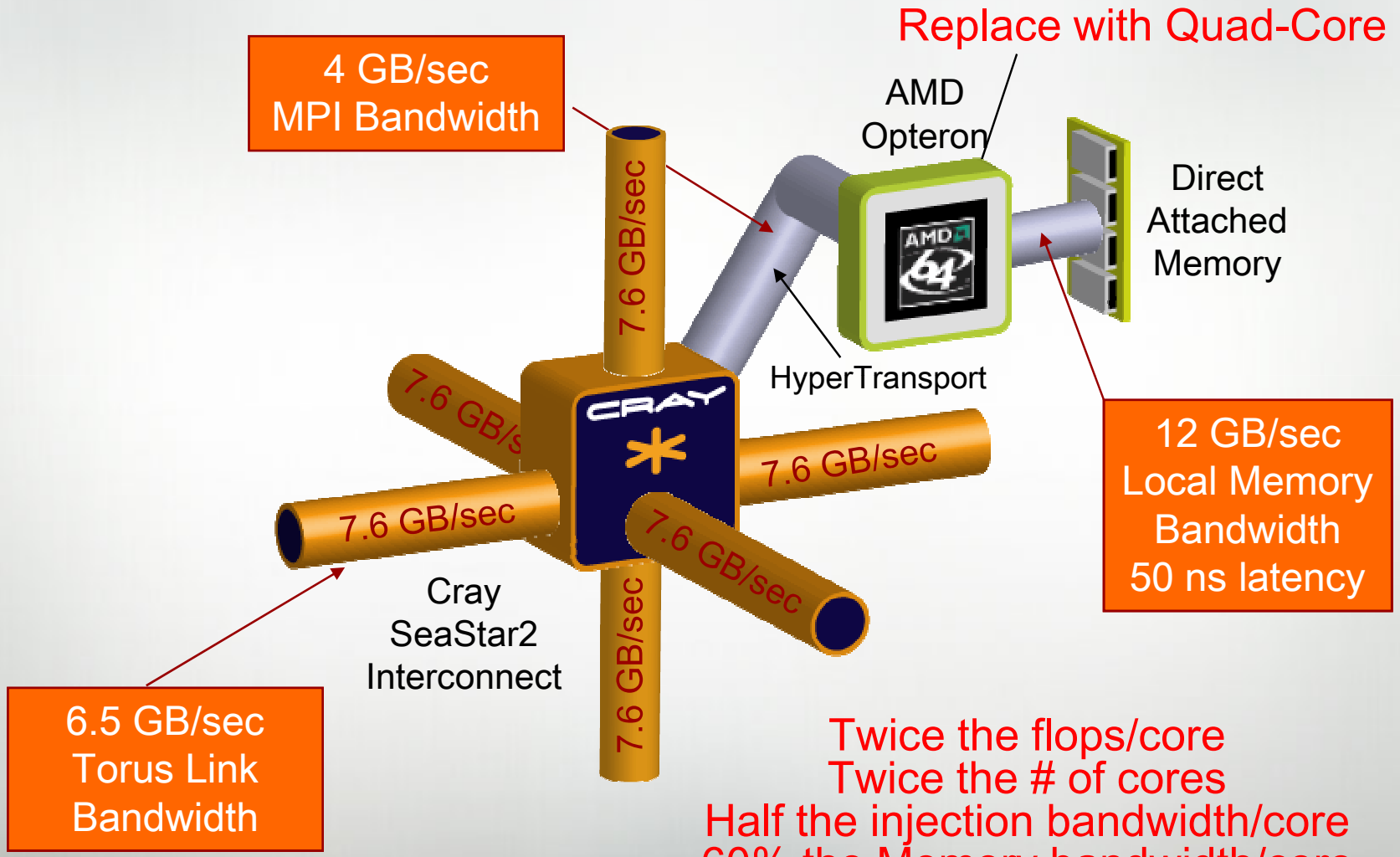
- Upgrade details
  - Dual-Core versus Quad-Core nodes
  - Dual-Core versus Quad-Core
  - System details
- Dual-Core to Quad-Core upgrades at other sites
  - ORNL
  - NERSC
- Application issues
  - What are the issues?
  - Selected HECToR results
  - What can be done?

The Cray XT4 Processing Element: Providing a bandwidth-rich environment



208 TF Upgrade

The Cray XT4 Processing Element: Providing a bandwidth-rich environment



Twice the flops/core
Twice the # of cores
Half the injection bandwidth/core
60% the Memory bandwidth/core

Let's Review: Dual-Core v. Quad-Core

Dual-Core

- Core
  - 2.8Ghz clock frequency
  - SSE SIMD FPU (2flops/cycle = 5.6GF peak)
- Cache Hierarchy
  - L1 Dcache/Icache: 64k/core
  - L2 D/I cache: 1M/core
  - SW Prefetch and loads to L1
  - Evictions and HW prefetch to L2
- Memory
  - Dual Channel DDR2
  - 10GB/s peak @ 667MHz
  - 8GB/s nominal STREAMs

Quad-Core

- Core
  - 2.3Ghz clock frequency
  - SSE SIMD FPU (4flops/cycle = 9.2GF peak)
- Cache Hierarchy
  - L1 Dcache/Icache: 64k/core
  - L2 D/I cache: 512 KB/core
  - L3 Shared cache 2MB/Socket
  - SW Prefetch and loads to L1,L2,L3
  - Evictions and HW prefetch to L1,L2,L3
- Memory
  - Dual Channel DDR2
  - 12GB/s peak @ 800MHz
  - 10GB/s nominal STREAMs

What will HECToR look like?

- Cray XT4 Dual-Core - Today
  - 63 TFlops
  - 5664 nodes
  - 11328 cores
  - 2-way SMP on the node
  - 6GB/node
  - 667 MHz memory
- Cray XT4 Quad-Core - Tomorrow
  - 208 TFlops
  - 5664 nodes
  - 22656 cores
  - 4-way SMP on the node
  - 8GB/node
  - 800 MHz memory

Quad-Core upgrades at other sites

Jaguar – Cray XT4
Upgraded to 263 TeraFlops

- We have upgraded Jaguar from single-core to Dual-Core to Quad-Core Opteron processors
  - Replaced 7,832 processors and added 15,664 2GB DIMMS

Quad-Core Processors	7,832
Memory / Core	2 GB
System Memory	62 TB
Disk Bandwidth	44 GB/s
Disk Space	900 TB
Node Size	4-core, 35 GF

NERSC's Cray XT4

- "Franklin" (NERSC-5)
  - 102 Cabinets in 17 rows
  - 9,660 nodes (19,320 cores)
  - 39.5 TBs Aggregate Memory (4 x 1GB DIMMs per node)
- Sustained performance: discussed later
- Interconnect: Cray SeaStar2, 3D Torus
  - >6 TB/s Bisection Bandwidth
  - >7 GB/s Link Bandwidth
- Shared Disk: 400+ TBs
- Network Connections
  - 24 x 10 Gbps + 16 x 1 Gbps
  - 60 x 4 Gbps Fibre Channel

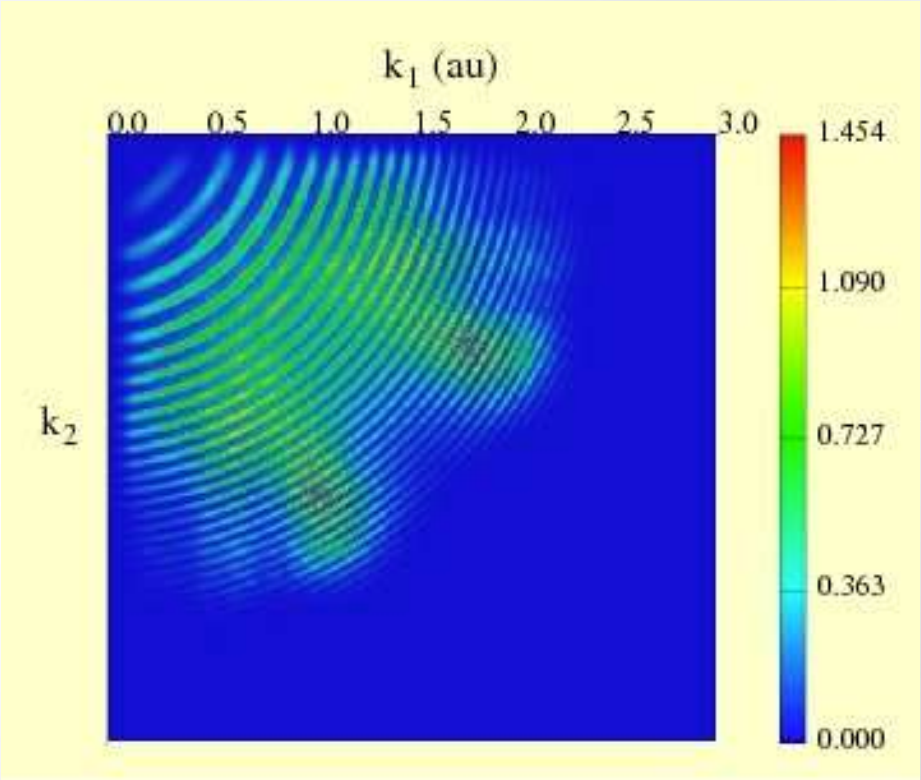
Franklin Quad-Core Upgrade

- In-place, no-interruption upgrade taking place between July and October, 2008.
- All 9,672 nodes change from 2.6-GHz AMD64 to 2.3-GHz Barcelona-64.
- QC nodes have 8 GB memory, same average GB/core as on DC Franklin.
- Memory from 667 MHz to 800 MHz.

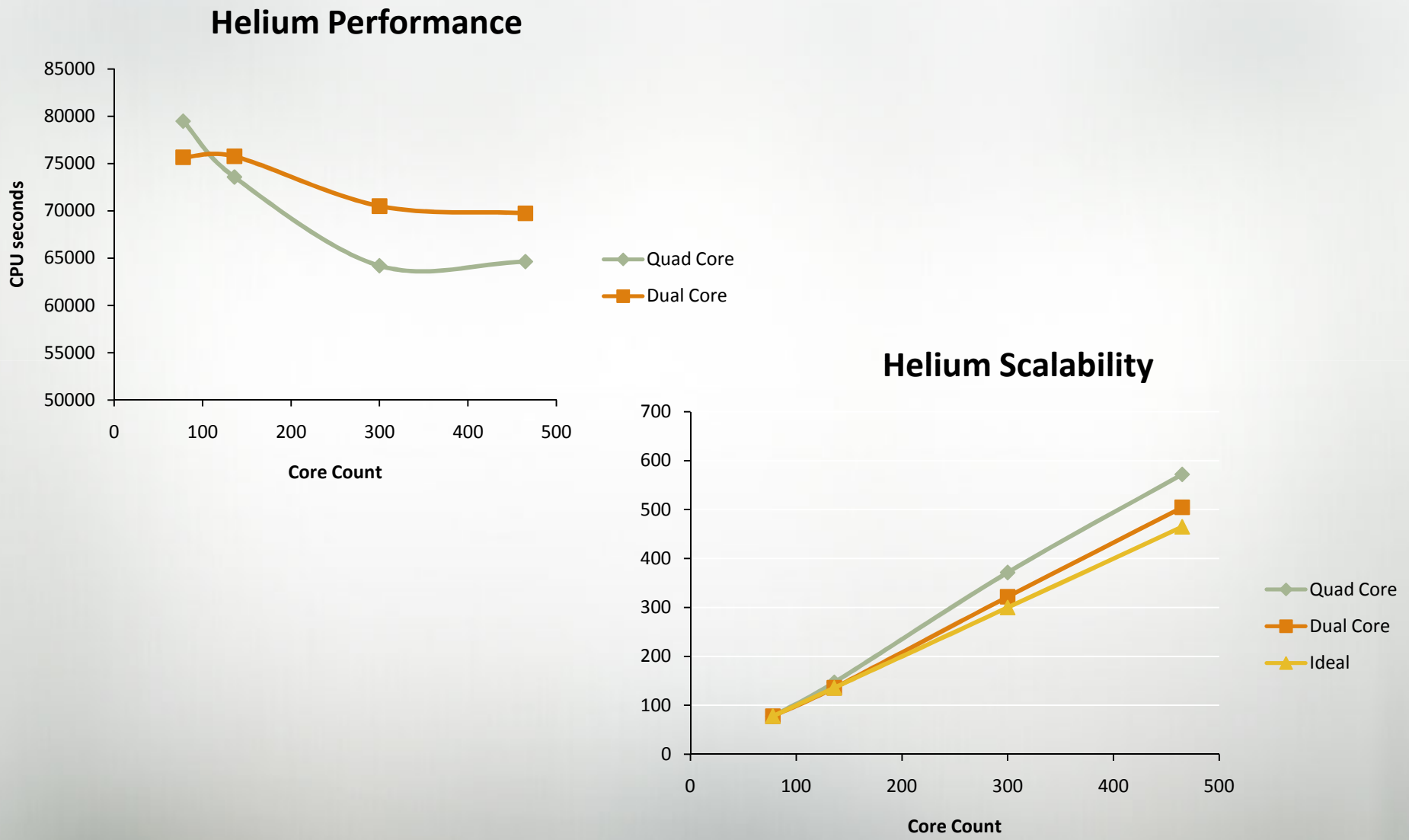
Initial HECToR Application Results

Helium

- Solves time-dependent Schrodinger equation in full dimensionality
- Used to model interaction between an intense linearly polarized laser light and the Helium atom
- Highly optimized for HPCx
  - Six months were spent re-engineering the code specifically for this platform
- Largest problem on HPCx – 1200 processors
  - 50% of time is spent on communication
- Initial simulations on HECToR – 2048 processors
  - 5% of time is spent on communication
- The code authors are now preparing to do simulations at the 800nm wavelength, which are not possible on HPCx



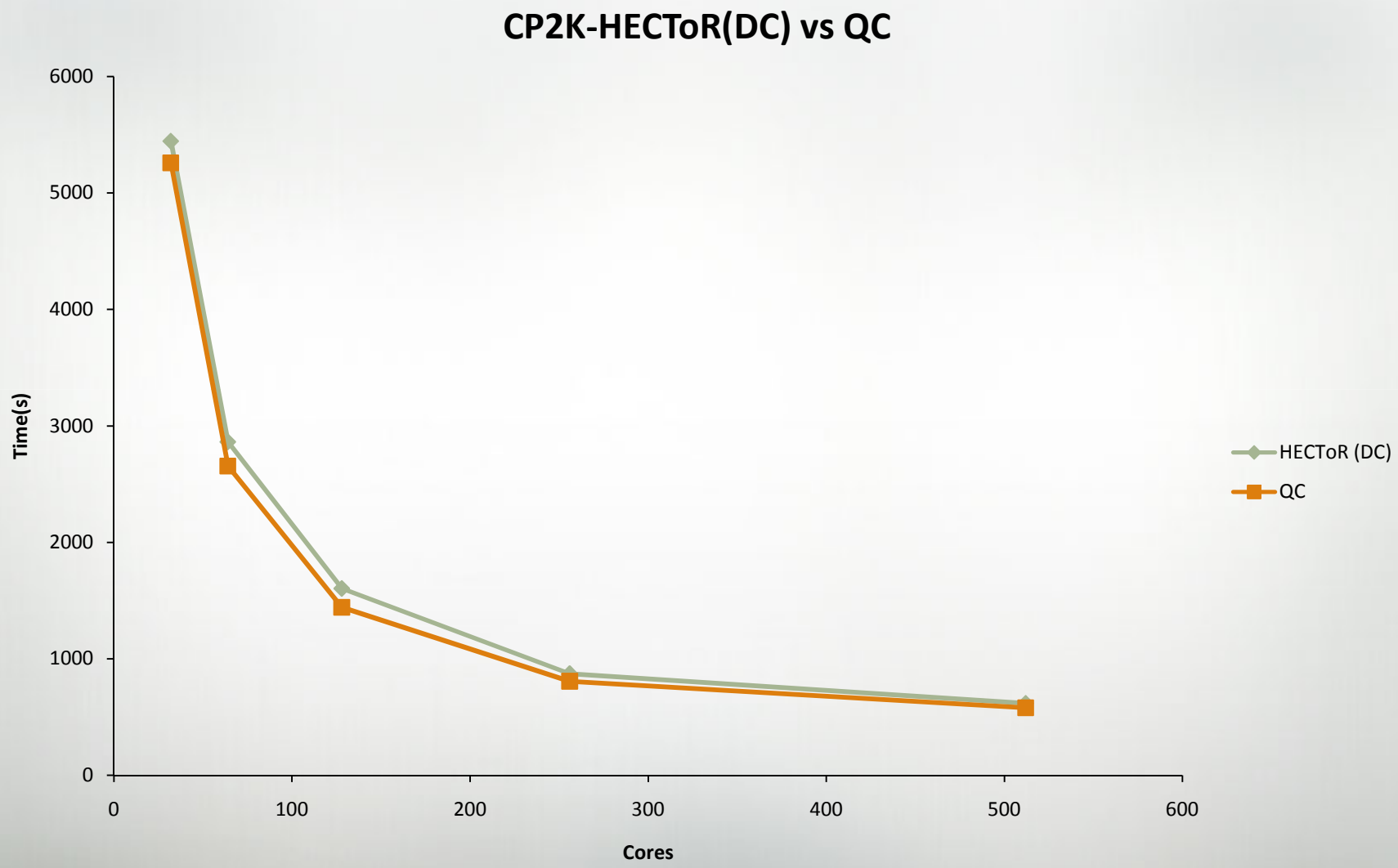
Performance Data



CP2K

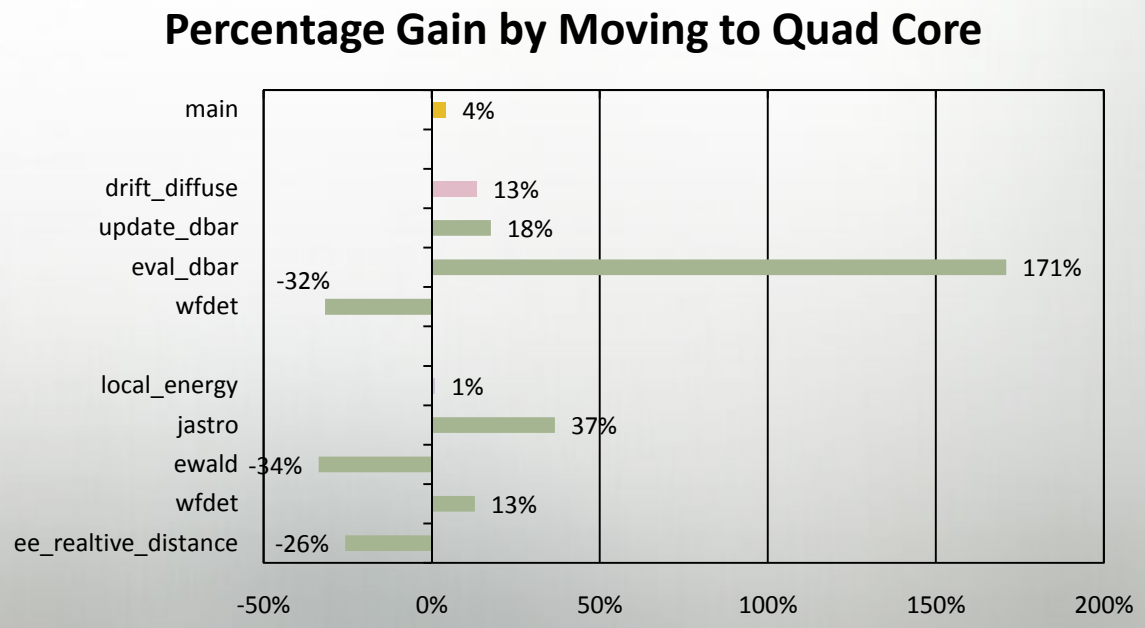
- Atomistic and molecular simulations of solid state, liquid and molecular systems
- Using the freely available version of the code from
  - http://cp2k.berlios.de/
- Using vendor benchmark input
  - Exercises a large number of kernels
  - Good balance of compute and communication
- Makes extensive use of numerical libraries
  - BLAS
  - ScaLAPACK
  - FFTW

CP2K

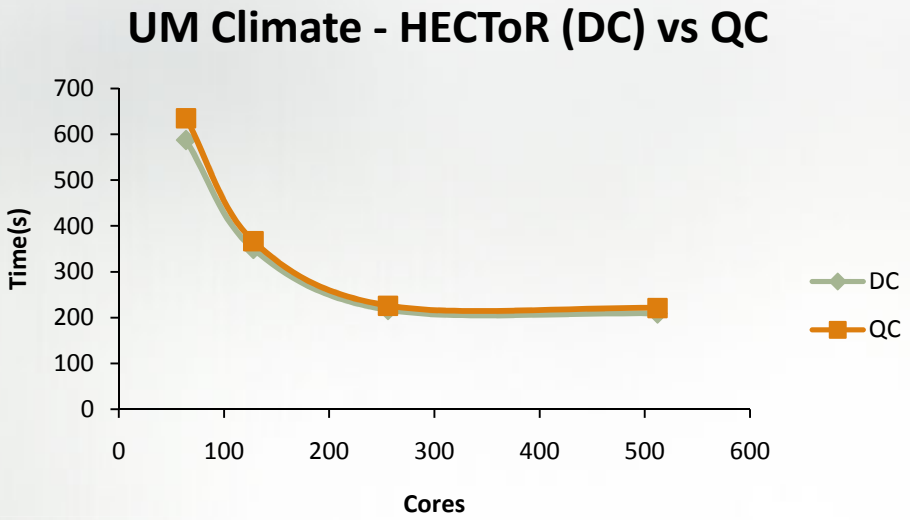

CP2K-HECToR(DC) vs QC
Time(s)
6000
5000
4000
3000
2000
1000
0
0
100
200
300
400
500
600
Cores
HECToR (DC)
QC

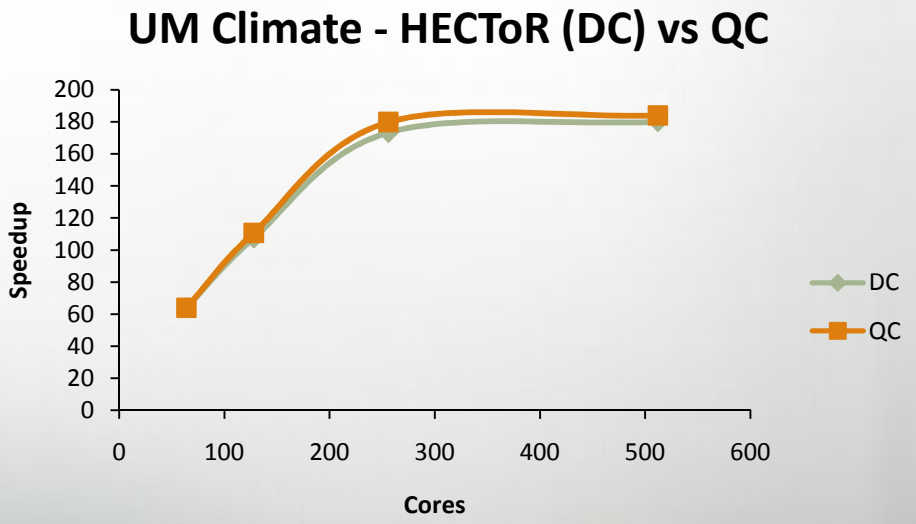
Casino

- Quantum Monte Carlo code used extensively on HECToR by Prof. Dario Alfe of UCL
- Approved for dCSE funding and supported by Lucian Anton, NAG.



UM


UM Climate - HECToR (DC) vs QC
Time(s)
700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
Cores
DC
QC


UM Climate - HECToR (DC) vs QC
Speedup
200
180
160
140
120
100
80
60
40
20
0
0
100
200
300
400
500
600
Cores
DC
QC

Quad-Core, More to the Point

- Doubled flops/clock, only if you use SSE128
  - Very-Short-Vector Instructions
- Clock reduced from 2.8GHz – 2.3 GHz
  - 18% clock speed reduction
- L2 Cache size has been reduced per-core, but shared L3 has been added
  - Essentially, more cache visible to the cores
- DDR2-800 memory has replaced DDR2-667
  - 10.6GB/s -> 12.8GB/s (21% improvement)
  - More total memory bandwidth and 2X as many memory controllers, also 2X as many cores to use the BW.
  - Symmetrical memory – 8GB/node provides full memory interleaving, whereas 6GB/node does not

What can be done?

- MPI is optimized for intra-node communication; however, message off the node will contend for bandwidth requirements off the node
- OpenMP across the cores on the node will help
  - Shared Cache is designed to help OpenMP reduce the applications memory requirements
  - Reduces the message traffic off the node
- Watch out for Libraries – are they Quad-Core enabled?

What about those SSE instructions

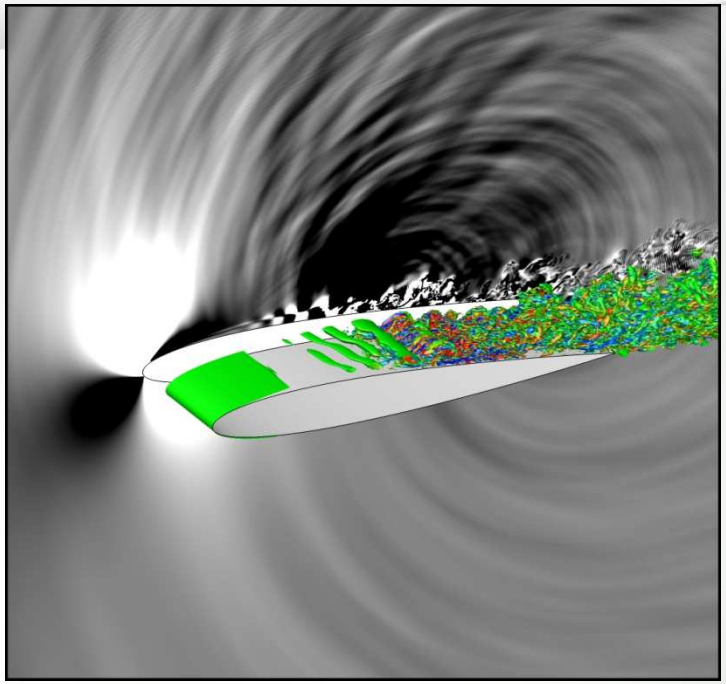
- The Quad-Core is capable of generating 4 flops/clock in 64 bit mode and 8 flops/clock for 32 bit mode
  - Assembler must contain SSE instructions
  - Compilers only generate SSE instructions when it can vectorize the DO loops
  - Libraries must be Quad-Core enabled

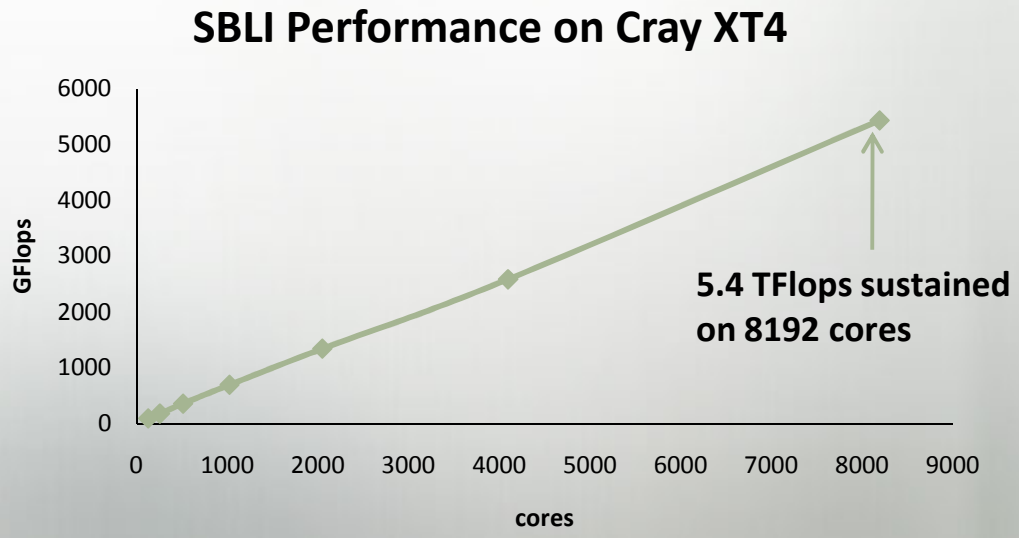
Some Lessons from Quad Core

- Vectorize

SBLI – Direct Numerical Simulation of Turbulence at Scale on Cray XT4

- Finite difference code for turbulent boundary layers
- Higher-order central differencing, shock preserving advection scheme from the TVD family, entropy splitting of the Euler terms
- Application areas include noise production from wing sections - critical in modern aircraft design
- Code scales to largest job queue on HECToR – 8192 cores -> 5.4 TFlops
- HPCx scaling stops at around 1200 processors
- Cray Centre of Excellence for HECToR have improved single CPU performance of this code on HECToR – 20% speedup over original version





APA analysis of code

- module load xt-craypat
- Create the instrumented executable
  - make hector
    - Use compiler listings with –Minfo and -Mneginfo
  - pat_build –O apa pdns3d.x
- Run the experiment
  - Run your simulation – aprun –n $NPROCS ./pdn3d.x+pat
- Create your report
  - pat_report –o sample.txt *.xf

Sample information

```
|  86.2% | 17474 |      -- |     -- |USER
||-----------------------------------------
||  39.2% |  7953 |      -- |     -- |rhs_
3|        |       |         |        | jason/src/sbli_3.5/rhs_3d.f
||||-------------------------------------------
4|||  23.2% |  4704 | 1068.80 |  18.8% |line.413
4|||   3.1% |   629 |   63.98 |   9.4% |line.78
4|||   2.9% |   592 |   55.06 |   8.6% |line.165
4|||   2.7% |   539 |   20.19 |   3.7% |line.190
4|||   2.3% |   476 |   23.73 |   4.8% |line.236
4|||   1.8% |   371 |   24.86 |   6.4% |line.214
4|||   1.8% |   356 |   32.34 |   8.5% |line.260
||||===========================================
```

USER / rhs_

Time%	39.5%		
Time	128.658205		
Imb.Time	19.624435		
Imb.Time%	13.4%		
Calls	150		
DATA_CACHE_MISSES	18.130M/sec	2260790340 misses	
PAPI_TOT_INS	1445.103M/sec	180203670533 instr	
PAPI_L1_DCA	757.287M/sec	94433319644 refs	
PAPI_FP_OPS	842.208M/sec	105022916627 ops	
User time (approx)	124.700 secs	324218781250 cycles	
Cycles	124.700 secs	324218781250 cycles	
User time (approx)	124.700 secs	324218781250 cycles	
Utilization rate	96.9%		
Instr per cycle	0.56 inst/cycle		
HW FP Ops / Cycles	0.32 ops/cycle		
HW FP Ops / User time	842.208M/sec	105022916627 ops	16.2%peak
HW FP Ops / WCT	816.294M/sec		
HW FP Ops / Inst	58.3%		
Computation intensity	1.11 ops/ref		
MIPS	92486.59M/sec		
MFLOPS	53901.30M/sec		
Instructions per LD ST	1.91 inst/ref		
LD & ST per D1 miss	41.77 refs/miss		
D1 cache hit ratio	97.6%		
LD ST per Instructions	52.4%		

Large loop, which is not vectorizing

```
do k=1,nzp
   do j=1,nyp
      do i=1,nxp                413, Loop not vectorized: data dependency
        wy(8)  = wx(i,j,k,8)
        wy(9)  = wx(i,j,k,9)
        wy(11) = wx(i,j,k,11)
        wy(12) = wx(i,j,k,12)
        wy(18) = wx(i,j,k,18)

        dxidx  = dxi_dx(i,j,IK)
        dxidy  = dxi_dy(i,j,IK)
        detadx = deta_dx(i,j,IK)
        detady = deta_dy(i,j,IK)

        q22 =    wy(18)*((wy(11)*dxidx
   $                 -wy(12)*detadx)
   $                +(-wy(8)*dxidy
   $                  +wy(9)*detady))

        etc…
```

Try the Cray (X1/X2) compiler…

```
411.  1------<       do k=1,nzp
412.  1 2----<         do j=1,nyp
413.  1 2 V--<           do i=1,nxp
414.  1 2 V               wy(8)  = wx(i,j,k,8)
415.  1 2 V               wy(9)  = wx(i,j,k,9)
416.  1 2 V               wy(11) = wx(i,j,k,11)
417.  1 2 V               wy(12) = wx(i,j,k,12)
418.  1 2 V               wy(18) = wx(i,j,k,18)
419.  1 2 V
420.  1 2 V               dxidx  = dxi_dx(i,j,k)
421.  1 2 V               dxidy  = dxi_dy(i,j,k)
422.  1 2 V               detadx = deta_dx(i,j,k)
423.  1 2 V               detady = deta_dy(i,j,k
```

ftn-6383 ftn: VECTOR File = rhs_3d.f, Line = 413
  A loop starting at line 413 requires an estimated 56 vector registers at line 941; 24 of these have been preemptively forced to memory.

ftn-6204 ftn: VECTOR File = rhs_3d.f, Line = 413
  A loop starting at line 413 was vectorized.

Why is PGI not vectorizing this loop???

```
do k=1,nzp
    do j=1,nyp
        do i=1,nxp
          wy(8)  = wx(i,j,k,8)
          wy(9)  = wx(i,j,k,9)
          wy(11) = wx(i,j,k,11)
          wy(12) = wx(i,j,k,12)
          wy(18) = wx(i,j,k,18)

          dxidx  = dxi_dx(i,j,IK)
          dxidy  = dxi_dy(i,j,IK)
          detadx = deta_dx(i,j,IK)
          detady = deta_dy(i,j,IK)

          q22 =    wy(18)*((wy(11)*dxidx
    $                   -wy(12)*detadx)
    $                  +(-wy(8)*dxidy
    $                    +wy(9)*detady))

          etc…
```

After much looking around at this, the only unusual thing I could see is the use of the array wy() for loop temporaries.
This was done as a single declaration of wy(50) was more compact then wy1, wy2, wy3, etc…

Sample code to test this…

```
subroutine vect_test
  implicit none

  integer :: i
  real*8  :: wy(4),wx(100,4)
  real*8  :: wy1,wy2,wy3,wy4,q11

  do i=1,100
    wy(1)=wx(i,1)
    wy(2)=wx(i,2)
    wy(3)=wx(i,3)
    wy(4)=wx(i,4)
    q11 = q11+wy(1)*wy(2)*wy(3)*wy(4)
  end do
  do i=1,100
    wy1=wx(i,1)
    wy2=wx(i,2)
    wy3=wx(i,3)
    wy4=wx(i,4)
    q11 = q11+wy1*wy2*wy3*wy4
  end do

end subroutine vect_test
```

```
jason@nid00004:~/src/vect_test> ftn -fast -Minfo -
Mneginfo -c vect_test.f90
/opt/xt-asyncpe/1.0a/bin/ftn: INFO: linux target is being
used
vect_test:
     8, Loop not vectorized: data dependency
    16, Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
```

USER / rhs_

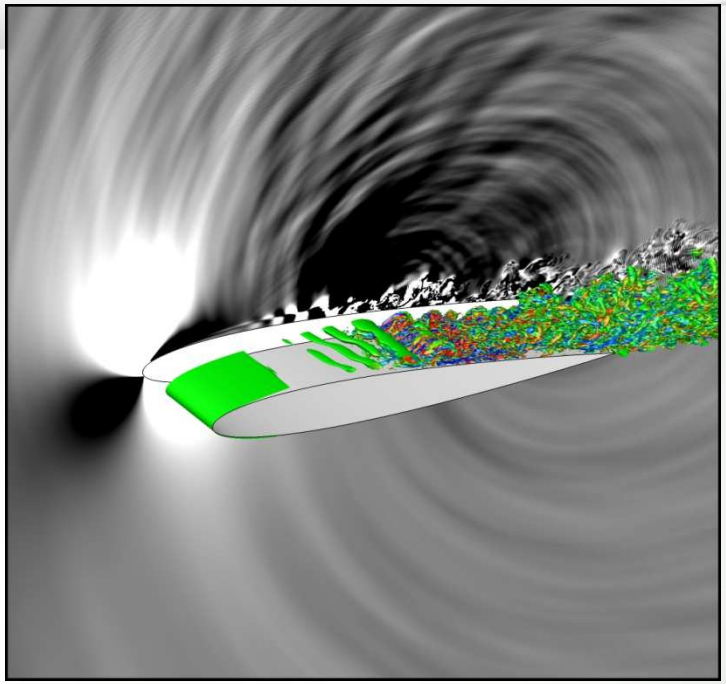
Time%	38.1%		
Time	106.324202		
Imb.Time	11.409116		
Imb.Time%	9.8%		
Calls	150		
DATA_CACHE_MISSES	21.902M/sec	2242750389 misses	
PAPI_TOT_INS	699.101M/sec	71588386163 instr	
PAPI_L1_DCA	537.188M/sec	55008430462 refs	
PAPI_FP_OPS	673.327M/sec	68949134785 ops	
User time (approx)	102.401 secs	266241828125 cycles	
Cycles	102.401 secs	266241828125 cycles	
User time (approx)	102.401 secs	266241828125 cycles	
Utilization rate	96.3%		
Instr per cycle	0.27 inst/cycle		
HW FP Ops / Cycles	0.26 ops/cycle		
HW FP Ops / User time	673.327M/sec	68949134785 ops	12.9%peak
HW FP Ops / WCT	648.480M/sec		
HW FP Ops / Inst	96.3%		
Computation intensity	1.25 ops/ref		
MIPS	44742.43M/sec		
MFLOPS	43092.91M/sec		
Instructions per LD ST	1.30 inst/ref		
LD & ST per D1 miss	24.53 refs/miss		
D1 cache hit ratio	95.9%		
LD ST per Instructions	76.8%		

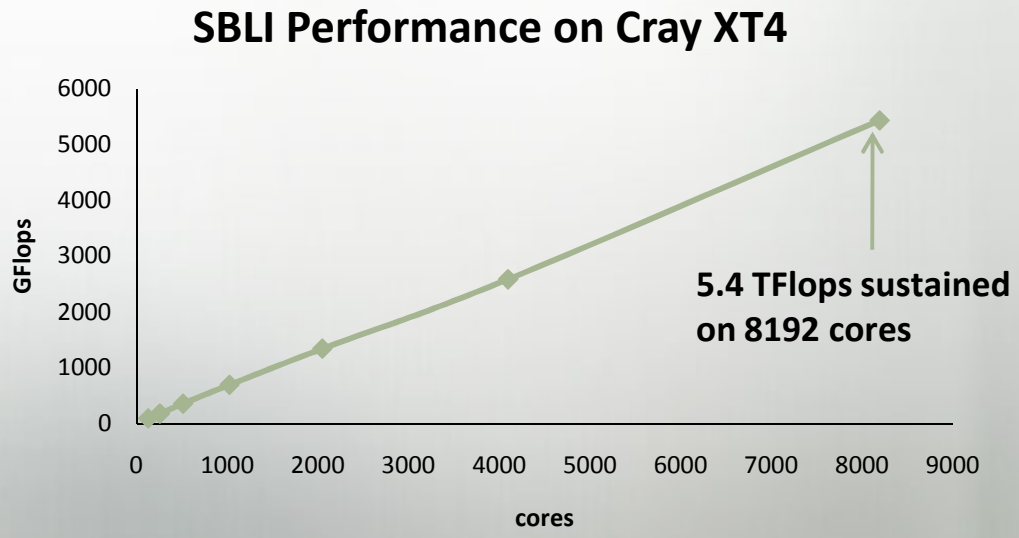
Some Lessons from Quad-Core

- Vectorize
- Cache Block/Make efficient use of cache

SBLI – Direct Numerical Simulation of Turbulence at Scale on Cray XT4

- Finite difference code for turbulent boundary layers
- Higher-order central differencing, shock preserving advection scheme from the TVD family, entropy splitting of the Euler terms
- Application areas include noise production from wing sections - critical in modern aircraft design
- Code scales to largest job queue on HECToR – 8192 cores -> 5.4 TFlops
- HPCx scaling stops at around 1200 processors
- Cray Centre of Excellence for HECToR have improved single CPU performance of this code on HECToR – 20% speedup over original version





APA analysis of code

- module load xt-craypat
- Create the instrumented executable
  - make hector
  - pat_build –O apa pdns3d.x
- Run the experiment
  - Run your simulation – aprun –n $NPROCS ./pdn3d.x+pat
- Create your report
  - pat_report –o sample.txt *.xf

Sampling experiment output

Table 1: Profile by Group, Function, and Line

```
 Samp % |  Samp |    Imb. |   Imb. |Group
        |       |    Samp | Samp % | Function
        |       |         |        |  Source
        |       |         |        |   Line
        |       |         |        |    PE='HIDE'

 100.0% | 20270 |      -- |     -- |Total
|-----------------------------------------------
|  86.2% | 17474 |      -- |     -- |USER
||----------------------------------------------
||  12.3% |  2488 |      -- |     -- |deriv_d1eta_2_
3|       |       |         |        | jason/src/sbli_3.5/cent2.f
||||--------------------------------------------
4|||  12.1% |  2455 |   90.88 |   3.6% |line.1824
```

- deriv_d1eta_2_:cent2.f:line.1824 – a triple nested loop with a difference calculation

deriv_d1eta_2 original

```
do k=1-zhalo,nzp+zhalo
   do j=1,nyp
     do i=1-xhalo,nxp+xhalo
       dfn(i,j,k) = ( fn(i,j-2,k) - fn(i,j+2,k)
&              +8.0d0*(fn(i,j+1,k) - fn(i,j-1,k)) )*facty
&               * hyr(j)
     end do
   end do
 end do
```

USER / deriv_d1eta_2_

Time%	12.2%	
Time	39.830899	
Imb.Time	2.933972	
Imb.Time%	7.0%	
Calls	2854	
DATA_CACHE_MISSES	46.294M/sec	1508451014 misses
PAPI_TOT_INS	656.987M/sec	21407421197 instr
PAPI_L1_DCA	452.621M/sec	14748311652 refs
PAPI_FP_OPS	440.116M/sec	14340835118 ops
User time (approx)	32.584 secs	84718968750 cycles
Cycles	32.584 secs	84718968750 cycles
User time (approx)	32.584 secs	84718968750 cycles
Utilization rate	81.8%	
Instr per cycle	0.25 inst/cycle	
HW FP Ops / Cycles	0.17 ops/cycle	
HW FP Ops / User time	440.116M/sec	14340835118 ops 8.5%peak
HW FP Ops / WCT	360.043M/sec	
HW FP Ops / Inst	67.0%	
Computation intensity	0.97 ops/ref	
MIPS	42047.19M/sec	
MFLOPS	28167.42M/sec	
Instructions per LD ST	1.45 inst/ref	
LD & ST per D1 miss	9.78 refs/miss	
D1 cache hit ratio	89.8%	
LD ST per Instructions	68.9%	

- Cache behaviour is quite poor

Why was original so bad?

```
do k=1-zhalo,nzp+zhalo
  do j=1,nyp
    do i=1-xhalo,nxp+xhalo
      dfn(i,j,k) = ( fn(i,j-2,k) - fn(i,j+2,k)
&              +8.0d0*(fn(i,j+1,k) - fn(i,j-1,k)) )*facty
&               * hyr(j)
    end do
  end do
 end do
```

- Natural loop order above seems OK to me…
- Let's see what the compiler thinks

```
d1eta_2:
  1820, Interchange produces reordered loop nest:
1822, 1820, 1824
  1824, Generated 4 alternate loops for the inner loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
```

```
do j=1,nyp
  do k=1-zhalo,nzp+zhalo
      do i=1-xhalo,nxp+xhalo
        dfn(i,j,k) = ( fn(i,j-2,k) - fn(i,j+2,k)
  &              +8.0d0*(fn(i,j+1,k) - fn(i,j-1,k)) )*facty
  &               * hyr(j)
      end do
    end do
   end do
```

Loop interchange ruins the cache behaviour – compiler thought it was helping with hyr, but didn't seem to consider fn accesses

Rewrite the loop?

```
d1eta_2:
  1822, Generated 4 alternate loops for the inner loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
        Generated vector sse code for inner loop
        Generated 4 prefetch instructions for this loop
```

```
do k=1-zhalo,nzp+zhalo
  do j=1,nyp
   hyrj = hyr(j)
    do i=1-xhalo,nxp+xhalo
      dfn(i,j,k) = ( fn(i,j-2,k) - fn(i,j+2,k)
&              +8.0d0*(fn(i,j+1,k) - fn(i,j-1,k)) )*facty
&               * hyrj
    end do
  end do
 end do
```

Stopped the loop interchange, and still getting inner loop vectorization

USER / deriv_d1eta_2_

Time%	7.2%	
Time	22.156190	
Imb.Time	2.450521	
Imb.Time%	10.1%	
Calls	2854	
DATA_CACHE_MISSES	36.943M/sec	594335578 misses
PAPI_TOT_INS	1342.373M/sec	21596266950 instr
PAPI_L1_DCA	921.836M/sec	14830620040 refs
PAPI_FP_OPS	892.677M/sec	14361505154 ops
User time (approx)	16.088 secs	41829125000 cycles
Cycles	16.088 secs	41829125000 cycles
User time (approx)	16.088 secs	41829125000 cycles
Utilization rate	72.6%	
Instr per cycle	0.52 inst/cycle	
HW FP Ops / Cycles	0.34 ops/cycle	
HW FP Ops / User time	892.677M/sec	14361505154 ops 17.2%peak
HW FP Ops / WCT	648.194M/sec	
HW FP Ops / Inst	66.5%	
Computation intensity	0.97 ops/ref	
MIPS	85911.88M/sec	
MFLOPS	57131.35M/sec	
Instructions per LD ST	1.46 inst/ref	
LD & ST per D1 miss	24.95 refs/miss	
D1 cache hit ratio	96.0%	
LD ST per Instructions	68.7%	

Some Lessons from Quad-Core

- Vectorize
- Cache Block
- Don’t stride through memory

Bad Striding

```
(    5)       COMMON A(8,8,IIDIM,8),B(8,8,iidim,8)


(   59)       DO 41090 K = KA, KE, -1
(   60)         DO 41090 J = JA, JE
(   61)           DO 41090 I = IA, IE
(   62)             A(K,L,I,J) = A(K,L,I,J) - B(J,1,i,k)*A(K+1,L,I,1)
(   63)      *    - B(J,2,i,k)*A(K+1,L,I,2) - B(J,3,i,k)*A(K+1,L,I,3)
(   64)      *    - B(J,4,i,k)*A(K+1,L,I,4) - B(J,5,i,k)*A(K+1,L,I,5)
(   65) 41090 CONTINUE
(   66)
```

PGI

59, Loop not vectorized: loop count too small

60, Interchange produces reordered loop nest: 61, 60

Loop unrolled 5 times (completely unrolled)

61, Generated vector sse code for inner loop

Pathscale

(lp41090.f:62) Non-contiguous array "A(_BLNK__.0.0)" reference exists. Loop was not vectorized.

(lp41090.f:62) Non-contiguous array "A(_BLNK__.0.0)" reference exists. Loop was not vectorized.

(lp41090.f:62) Non-contiguous array "A(_BLNK__.0.0)" reference exists. Loop was not vectorized.

(lp41090.f:62) Non-contiguous array "A(_BLNK__.0.0)" reference exists. Loop was not vectorized.

Rewrite

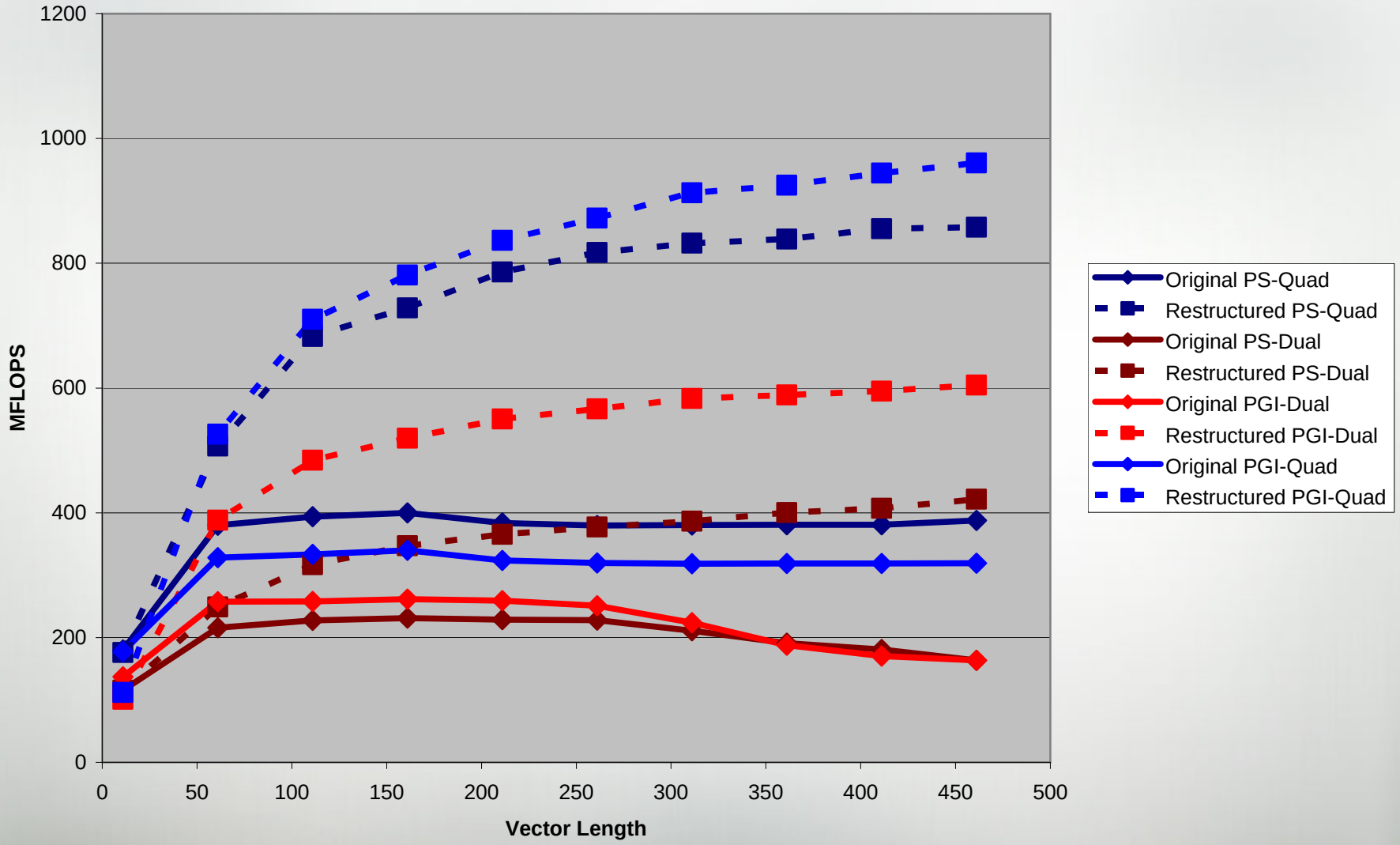
```
(    6)       COMMON AA(IIDIM,8,8,8),BB(IIDIM,8,8,8)


(   95)       DO 41091 K = KA, KE, -1
(   96)         DO 41091 J = JA, JE
(   97)           DO 41091 I = IA, IE
(   98)             AA(I,K,L,J) = AA(I,K,L,J) - BB(I,J,1,K)*AA(I,K+1,L,1)
(   99)      *     - BB(I,J,2,K)*AA(I,K+1,L,2) - BB(I,J,3,K)*AA(I,K+1,L,3)
(  100)      *     - BB(I,J,4,K)*AA(I,K+1,L,4) - BB(I,J,5,K)*AA(I,K+1,L,5)
(  101) 41091 CONTINUE
```

PGI

95, Loop not vectorized: loop count too small
96, Outer loop unrolled 5 times (completely unrolled)
97, Generated 3 alternate loops for the inner loop
Generated vector sse code for inner loop
Generated 8 prefetch instructions for this loop
Generated vector sse code for inner loop
Generated 8 prefetch instructions for this loop
Generated vector sse code for inner loop
Generated 8 prefetch instructions for this loop
Generated vector sse code for inner loop
Generated 8 prefetch instructions for this loop

Pathscale

```
(lp41090.f:99) LOOP WAS VECTORIZED.
```


LP41090
MFLOPS
Vector Length
0
200
400
600
800
1000
1200
0
50
100
150
200
250
300
350
400
450
500
Original PS-Quad
Restructured PS-Quad
Original PS-Dual
Restructured PS-Dual
Original PGI-Dual
Restructured PGI-Dual
Original PGI-Quad
Restructured PGI-Quad

Some Lessons from Quad-Core

- Vectorize
- Cache Block
- Don’t stride through memory
- Use Quad-Core enabled Libraries
  - Loading xtpe-quadcore module ensures you get the Quad-Core enabled versions of libsci, and the appropriate compiler flags to generate code for Quad-Core

Some Lessons from Quad-Core

- Vectorize
- Cache Block
- Don’t stride through memory
- Use Quad-Core enabled Libraries
- Pre-post Receives

Some Lessons from Quad-Core

- Vectorize
- Cache Block
- Don’t stride through memory
- Use Quad-Core enabled Libraries
- Pre-post Receives
- Investigate OpenMP

Some Lessons from Quad-Core

- Vectorize
- Cache Block
- Don’t stride through memory
- Use Quad-Core enabled Libraries
- Pre-post Receives
- Investigate OpenMP
- Memory per core decrease

CASINO

- Quantum Monte Carlo (QMC) electronic structure calculations for finite and periodic systems
- Fortran 90 + MPI
  - Note: No OpenMP
- Over 100,000 lines of code
- One of EPSRC's HECToR benchmark codes
  - Now used heavily in production

Casino – The Problems

- 1. Memory
- User wants to use VERY LARGE wavefunction data sets
  - 2 copies (~4GB each) do not fit on HECToR's 6GB (dual core) nodes
  - Effective performance is HALF (since must run in single core mode)
- Solution:
  - Array is read only (once loaded) so only 1 copy is really needed
  - Use a single SHARED array (between MPI tasks on node)
    - Note: Too big a job to re-implement whole code in OpenMP

Casino – shared memory

- Method
  - Establish configuration (PEs on each node, and Master PE for each node)
  - Use Posix or System V shared memory to allocate large array on each node
  - Map this space onto the users array (Fortran 90 Pointer)
- Shared memory issues
  - Can't use Posix as /dev/shm is not user writeable
  - => Use System V shared memory
  - BUT: System V shared memory uses int (32 bits) for size
  - => Build up full size out of Sys V blocks of 1 GByte each
  - and map successive blocks to address of previous + 1 Gbyte
  - Keep track of all blocks allocated, so they can be 'DE-allocated'
  - Delete all blocks (once mapped on all PEs) so that the shared segments disappear on program termination or failure/crash.

Get help with all of this!

- HECToR webpage – documentation, etc
  - http://www.hector.ac.uk/
- CSE training courses and support
  - http://www.hector.ac.uk/cse/
- Upcoming Cray Centre of Excellence Workshop
  - Specifically targeted at Quad-Core
  - Dates still to be finalized
  - Will be advertised on the HECToR webpage
- Cray – http://www.cray.com/
  - Documentation – http://docs.cray.com/

Concerned about your codes performance on Quad-Core?

- Give it to us!
  - Jason Beech-Brandt – jason@cray.com
  - Kevin Roy – kroy@cray.com
- We can run your code on Quad-Core today to give you some advance notice on how your code will perform – and some pointers for improving that performance
- Source, makefile, and dataset