Threading on the Node

OpenMP works great if it is implemented correctly

Looking at KNL

- **Lots of cores**
  - While MPI can run across all the cores, there are situations where MPI hits bottlenecks due to the number of cores on the nodes
  - We ALWAYS want to run some MPI tasks on the node – all OpenMP on the node seldom wins
  - Want to identify a MPI sweet spot; that is, when MPI only stops scaling
- **OpenMP**
  - Traditional approach has many short-comings
    - Requires a lot of code modifications
      - Lots of Comment Line directives
      - Does not deal with locality
      - Difficult to dynamically load balance
  - Is there a better way?
    - SPMD OpenMP
      - Fewer code modifications; however, more difficult to do.
      - Requires a better understanding of threads

Why does all-MPI work well on multi/many core architectures?

- **All MPI forces locality**
  - Each MPI task allocated/utilizes memory within the NUMA region that it is running in.
- **All MPI allows tasks to run asynchronously**
  - This allows very good sharing of the memory bandwidth available on the node
- **All MPI is easy – compile and go**
- **One MPI disadvantage is that re-distributing work is difficult and inefficient**
  - Have to move a lot of data

Can we allow threads to work asynchronously?

- **Must minimize synchronization**
  - Calling un-threaded routines
    - Must extend concept to all computational rotuines called within the parallel region
    - You can have replicated computation across the threads
  - Calling library routines
    - Can each thread call a un-threaded library routine?
    - If library routine is threaded, must barrier prior to and after call
  - Calling MPI
    - Consider having each thread do it own message passing
      - Example coming

On multi-core and many-core systems

- **We will want to run as many MPI tasks that gives us a GOOD performance increase**
  - Then we want to add threads to achieve as much parallelism on the node as possible
  - Ideally if we use at least one MPI task on each NUMA region, the threading within the NUMA region will not incur decreased efficiency due to threading across disparate NUMA regions
  - Still efficient threading is required
    - Numerous parallel regions are expensive, low level, low granularity loops will not return much performance
    - High level parallel loops are preferred
      - More difficult to scope
        - Probably want to find out about Reveal

Lets examine the idea of high level OpenMP

- When employing OpenMP, one must understand the usual loop iteration counts of the loops involved
- When employing OpenMP, you should also clean up obvious inefficiencies
- When using high level OpenMP, it is okay to have threads replicate work – its better than parking the threads and then starting them up again.

Analysis of BGW Kernels

A study of higher level OpenMP

Introduction

- BGW test has three kernels
- Reveal does very well on one
- Optimization improved other two significantly

Profiling the BGW test

Table 2: Inclusive and Exclusive Time in Loops (from -hprofile_generate)

Loop Incl Time%	Loop Incl Time	Time (Loop Adj.)	Loop Hit	Loop Trips Avg	Loop Trips Min	Loop Trips Max	Function=/.LOOP[.]
94.5%	55.048452	0.003011	1	96.0	96	96	hackakernel_.LOOP.04.li.121
91.5%	53.298164	0.015635	96	240.0	240	240	hackakernel_.LOOP.08.li.166
91.5%	53.282529	0.055862	23,040	100.0	100	100	hackakernel_.LOOP.09.li.170
91.4%	53.226667	53.226667	2,304,000	8,000.0	8,000	8,000	hackakernel_.LOOP.10.li.177
2.8%	1.658162	0.713467	96	240.0	240	240	hackakernel_.LOOP.11.li.192
1.6%	0.944695	0.944695	22,944	6,000.0	6,000	6,000	hackakernel_.LOOP.12.li.215
0.1%	0.081882	0.000093	96	100.0	100	100	hackakernel_.LOOP.06.li.138
0.1%	0.081789	0.081789	9,600	8,000.0	8,000	8,000	hackakernel_.LOOP.07.li.145
0.0%	0.005393	0.005393	96	6,000.0	6,000	6,000	hackakernel_.LOOP.13.li.243
0.0%	0.000204	0.000204	96	6,000.0	6,000	6,000	hackakernel_.LOOP.05.li.129
0.0%	0.000000	0.000000	1	240.0	240	240	hackakernel_.LOOP.02.li.79
0.0%	0.000000	0.000000	1	240.0	240	240	hackakernel_.LOOP.03.li.85
0.0%	0.000000	0.000000	1	96.0	96	96	hackakernel_.LOOP.01.li.60

Loop B

Loop C

Loop A

Kernel A loop

```
138.  + 1 2--------<          do my_igp = 1, ngpown        (100)
139.    1 2                    if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle
140.    1 2
141.    1 2                    igmax=ncouls
142.    1 2
143.    1 2                    mygpvar1 = CONJG(leftvector(my_igp,n1))
144.    1 2
145.    1 2 Vr2----<           do ig = 1, igmax             (800)
146.    1 2 Vr2                  matngmatmgpD(ig,my_igp) = rightvector(ig,n1) * mygpvar1
147.    1 2 Vr2---->           enddo
148.    1 2-------->          enddo
```

Which of these two loop should we parallelize?
Are there some inefficencies?

Kernel B loop

```
165.     1 A-------<>          schdt_array = 0D0
166.   + 1 b--------<           do ifreq=1,nFreq                    (240)
167.     1 b
168.     1 b                       schDt = (0D0,0D0)
169.     1 b
170.   + 1 b b------<             do my_igp = 1, ngpown              (100)
171.     1 b b
172.     1 b b                       if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle
173.     1 b b
174.     1 b b                       igmax=ncouls
175.     1 b b
176.     1 b b                       schDtt = (0D0,0D0)
177.     1 b b Vr3--<                do ig = 1, igmax                  (8000)
178.     1 b b Vr3                     I_epsRggp_int = I_epsR_array(ig,my_igp,ifreq)
179.     1 b b Vr3                     I_epsAggp_int = I_epsA_array(ig,my_igp,ifreq)
180.     1 b b Vr3                     schD=I_epsRggp_int-I_epsAggp_int
181.     1 b b Vr3                     schDtt = schDtt + matngmatmgpD(ig,my_igp)*schD
182.     1 b b Vr3-->                enddo
183.     1 b b                       schdt_array(ifreq) = schdt_array(ifreq) + schDtt
184.     1 b b------>              enddo
185.     1 b
186.     1 b-------->          enddo
187.     1
188.   + 1                     call timget(endtime_ch)
189.     1                     time_b = time_b + endtime_ch - starttime_ch
190.   + 1                     call timget(starttime_ch)
```

Once again which loop is best to target for parallelization

Kernel C loop

```
192.  + 1 2--------<         do ifreq=1,nFreq                    (240)
193.    1 2
194.    1 2                      schDt = schDt_array(ifreq)
195.    1 2
196.    1 2                      cedifft_zb = dFreqGrid(ifreq)
197.    1 2                      cedifft_coh = CMPLX(cedifft_zb,0D0)- dFreqBrd(ifreq)
198.    1 2
199.    1 2                      if (ifreq .ne. 1) then
200.    1 2                        cedifft_zb_right = cedifft_zb
201.    1 2                        cedifft_zb_left = dFreqGrid(ifreq-1)
202.    1 2                        schDt_right = schDt
203.    1 2                        schDt_left = schDt_array(ifreq-1)
204.    1 2                        schDt_avg = 0.5D0 * ( schDt_right + schDt_left )
205.    1 2                        schDt_lin = schDt_right - schDt_left
206.    1 2                        schDt_lin2 = schDt_lin/(cedifft_zb_right-cedifft_zb_left)
207.    1 2                      endif
208.    1 2
209.    1 2           ! The below two lines are for sigma1 and sigma3
210.    1 2                      if (ifreq .ne. nFreq) then
211.    1 2 fVr2--<>               schDi(:) = schDi(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedifft_coh)
212.    1 2 f-----<>               schDi_corb(:) = schDi_corb(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedif
213.    1 2                      endif
214.    1 2                      if (ifreq .ne. 1) then
215.    1 2 V------<               do iw = 1, nfreqeval              (6000)
```

Once again which loop is best to target for parallelization

Array Assignment

Kernel C loop -continued

```
216.    1 2 V        !These lines are for sigma2
217.    1 2 V                        intfact=abs((wxi(iw)-cedifft_zb_right)/(wxi(iw)-cedifft_zb_left))
218.    1 2 V                        if (intfact .lt. 1d-4) intfact = 1d-4
219.    1 2 V                        if (intfact .gt. 1d4) intfact = 1d4
220.    1 2 V                        intfact = -log(intfact)
221.    1 2 V                        sch2Di(iw) = sch2Di(iw) - CMPLX(0.d0,prefactor) * schDt_avg * intfact
222.    1 2 V        !These lines are for sigma4
223.    1 2 V                        if (flag_occ) then
224.    1 2 V                          intfact=abs((wxi(iw)+cedifft_zb_right)/(wxi(iw)+cedifft_zb_left))
225.    1 2 V                          if (intfact .lt. 1d-4) intfact = 1d-4
226.    1 2 V                          if (intfact .gt. 1d4) intfact = 1d4
227.    1 2 V                          intfact = log(intfact)
228.    1 2 V                          schDt_lin3 = (schDt_left + schDt_lin2*(-wxi(iw)-cedifft_zb_left))*intfact
229.    1 2 V                        else
230.    1 2 V                          schDt_lin3 = (schDt_left + schDt_lin2*(wxi(iw)-cedifft_zb_left))*intfact
231.    1 2 V                        endif
232.    1 2 V                        schDt_lin3 = schDt_lin3 + schDt_lin
233.    1 2 V                        schDi_cor(iw) = schDi_cor(iw) - CMPLX(0.d0,prefactor) * schDt_lin3
234.    1 2 V------>               enddo
235.    1 2                      endif
236.    1 2-------->         enddo
```

Kernel A

```
 do my_igp = 1, ngpown
  if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle

  igmax=ncouls

  mygpvar1 = CONJG(leftvector(my_igp,n1))

  do ig = 1, igmax
    matngmatmgpD(ig,my_igp) = rightvector(ig,n1) * mygpvar1
  enddo
enddo
```

Kernel A

Automatically inserted by Reveal

```
138.     1              ! Directive inserted by Cray Reveal.  May be incomplete.
139.     1 M----------< !$OMP  parallel do default(none)                                &
140.     1 M            !$OMP&   private (ig,igmax,mygpvar1,my_igp)                    &
141.     1 M            !$OMP&   shared  (leftvector,matngmatmgpd,n1,ncouls,ngpown,rightvector)
142.   + 1 M m--------<          do my_igp = 1, ngpown
143.     1 M m                    if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle
144.     1 M m
145.     1 M m                    igmax=ncouls
146.     1 M m
147.     1 M m                    mygpvar1 = CONJG(leftvector(my_igp,n1))
148.     1 M m
149.     1 M m Vr2----<           do ig = 1, igmax
150.     1 M m Vr2                  matngmatmgpD(ig,my_igp) = rightvector(ig,n1) * mygpvar1
151.     1 M m Vr2---->           enddo
152.     1 M m------->>         enddo
```

Kernel A

Automatically changed by Levesque

```
137.     1 M----------< !$OMP PARALLEL DO
138.     1 M imV------<          do ig = 1, ncouls
139.   + 1 M imV ir4--<            do my_igp = 1, min(ncouls,ngpown)
140.     1 M imV ir4                 matngmatmgpD(ig,my_igp) = rightvector(ig,n1) * CONJG(leftvector(my_igp,
141.     1 M imV ir4-->              enddo
142.     1 M imV------>>         enddo
```

Kernel B

```
schdt_array = 0D0
do ifreq=1,nFreq

    schDt = (0D0,0D0)

    do my_igp = 1, ngpown

      if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle

      igmax=ncouls

      schDtt = (0D0,0D0)
      do ig = 1, igmax
        I_epsRggp_int = I_epsR_array(ig,my_igp,ifreq)
        I_epsAggp_int = I_epsA_array(ig,my_igp,ifreq)
        schD=I_epsRggp_int-I_epsAggp_int
        schDtt = schDtt + matngmatmgpD(ig,my_igp)*schD
      enddo
      schdt_array(ifreq) = schdt_array(ifreq) + schDtt
    enddo

enddo
```

Kernel B

Automatically inserted by Reveal

```
170.    1               ! Directive inserted by Cray Reveal.  May be incomplete.
171.    1 M-----------< !$OMP  parallel do default(none)                                &
172.    1 M             !$OMP&   private (ifreq,ig,igmax,i_epsaggp_int,i_epsrggp_int,my_igp,     &
173.    1 M             !$OMP&            schd,schdt,schdtt)                           &
174.    1 M             !$OMP&   shared  (i_epsa_array,i_epsr_array,matngmatmgpd,ncouls,nfreq,    &
175.    1 M             !$OMP&            ngpown,schdt_array)
176.  + 1 M m---------<         do ifreq=1,nFreq
177.    1 M m
178.    1 M m                     schDt = (0D0,0D0)
179.    1 M m
180.  + 1 M m 4-------<           do my_igp = 1, ngpown
181.    1 M m 4
182.    1 M m 4                     if (my_igp .gt. ncouls .or. my_igp .le. 0) cycle
183.    1 M m 4
184.    1 M m 4                     igmax=ncouls
185.    1 M m 4
186.    1 M m 4                     schDtt = (0D0,0D0)
187.    1 M m 4 Vr3--<              do ig = 1, igmax
188.    1 M m 4 Vr3                   I_epsRggp_int = I_epsR_array(ig,my_igp,ifreq)
189.    1 M m 4 Vr3                   I_epsAggp_int = I_epsA_array(ig,my_igp,ifreq)
190.    1 M m 4 Vr3                   schD=I_epsRggp_int-I_epsAggp_int
191.    1 M m 4 Vr3                   schDtt = schDtt + matngmatmgpD(ig,my_igp)*schD
192.    1 M m 4 Vr3-->              enddo
193.    1 M m 4                     schdt_array(ifreq) = schdt_array(ifreq) + schDtt
194.    1 M m 4------>            enddo
195.    1 M m
196.    1 M m-------->>         enddo
```

Kernel C

```
            do ifreq=1,nFreq

            schDt = schDt_array(ifreq)

            cedifft_zb = dFreqGrid(ifreq)
            cedifft_coh = CMPLX(cedifft_zb,0D0)- dFreqBrd(ifreq)

            if (ifreq .ne. 1) then
              cedifft_zb_right = cedifft_zb
              cedifft_zb_left = dFreqGrid(ifreq-1)
              schDt_right = schDt
              schDt_left = schDt_array(ifreq-1)
              schDt_avg = 0.5D0 * ( schDt_right + schDt_left )
              schDt_lin = schDt_right - schDt_left
              schDt_lin2 = schDt_lin/(cedifft_zb_right-cedifft_zb_left)
            endif

! The below two lines are for sigma1 and sigma3
            if (ifreq .ne. nFreq) then
              schDi(:) = schDi(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedifft_coh)
              schDi_corb(:) = schDi_corb(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedifft_cor)
            endif
            if (ifreq .ne. 1) then
              do iw = 1, nfreqeval
!These lines are for sigma2
                intfact=abs((wxi(iw)-cedifft_zb_right)/(wxi(iw)-cedifft_zb_left))
                if (intfact .lt. 1d-4) intfact = 1d-4
                if (intfact .gt. 1d4) intfact = 1d4
                intfact = -log(intfact)
                sch2Di(iw) = sch2Di(iw) - CMPLX(0.d0,prefactor) * schDt_avg * intfact
```

Kernel C

```
!These lines are for sigma4
                if (flag_occ) then
                  intfact=abs((wxi(iw)+cedifft_zb_right)/(wxi(iw)+cedifft_zb_left))
                  if (intfact .lt. 1d-4) intfact = 1d-4
                  if (intfact .gt. 1d4) intfact = 1d4
                  intfact = log(intfact)
                  schDt_lin3 = (schDt_left + schDt_lin2*(-wxi(iw)-cedifft_zb_left))*intfact
                else
                  schDt_lin3 = (schDt_left + schDt_lin2*(wxi(iw)-cedifft_zb_left))*intfact
                endif
                schDt_lin3 = schDt_lin3 + schDt_lin
                schDi_cor(iw) = schDi_cor(iw) - CMPLX(0.d0,prefactor) * schDt_lin3
              enddo
            endif
          enddo
```

Kernel C

```
202.  + 1 2----------<         do ifreq=1,nFreq
203.    1 2
204.    1 2                        schDt = schDt_array(ifreq)
205.    1 2
206.    1 2                        cedifft_zb = dFreqGrid(ifreq)
207.    1 2                        cedifft_coh = CMPLX(cedifft_zb,0D0)- dFreqBrd(ifreq)
208.    1 2
209.    1 2                        if (ifreq .ne. 1) then
210.    1 2                          cedifft_zb_right = cedifft_zb
211.    1 2                          cedifft_zb_left = dFreqGrid(ifreq-1)
212.    1 2                          schDt_right = schDt
213.    1 2                          schDt_left = schDt_array(ifreq-1)
214.    1 2                          schDt_avg = 0.5D0 * ( schDt_right + schDt_left )
215.    1 2                          schDt_lin = schDt_right - schDt_left
216.    1 2                          schDt_lin2 = schDt_lin/(cedifft_zb_right-cedifft_zb_left)
217.    1 2                        endif
218.    1 2
219.    1 2            ! The below two lines are for sigma1 and sigma3
220.    1 2                        if (ifreq .ne. nFreq) then
221.    1 2 fVr2----<>               schDi(:) = schDi(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedifft_coh)
222.    1 2 f-------<>               schDi_corb(:) = schDi_corb(:) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(:)-cedifft_cor)
223.    1 2                        endif
224.    1 2                        if (ifreq .ne. 1) then
```

Kernel C

Automatically inserted by Reveal

```
225.    1 2            ! Directive inserted by Cray Reveal.  May be incomplete.
226.    1 2 M--------< !$OMP  parallel do default(none)                                  &
227.    1 2 M          !$OMP&   private (intfact,iw,schdt_lin3)                          &
228.    1 2 M          !$OMP&   shared  (cedifft_zb_left,cedifft_zb_right,flag_occ,nfreqeval,   &
229.    1 2 M          !$OMP&            prefactor,sch2di,schdi_cor,schdt_avg,schdt_left,       &
230.    1 2 M          !$OMP&            schdt_lin,schdt_lin2,wxi)
231.    1 2 M mV-----<               do iw = 1, nfreqeval
232.    1 2 M mV       !These lines are for sigma2
233.    1 2 M mV                       intfact=abs((wxi(iw)-cedifft_zb_right)/(wxi(iw)-cedifft_zb_left))
234.    1 2 M mV                       if (intfact .lt. 1d-4) intfact = 1d-4
235.    1 2 M mV                       if (intfact .gt. 1d4) intfact = 1d4
236.    1 2 M mV                       intfact = -log(intfact)
237.    1 2 M mV                       sch2Di(iw) = sch2Di(iw) - CMPLX(0.d0,prefactor) * schDt_avg * intfact
238.    1 2 M mV       !These lines are for sigma4
239.    1 2 M mV                       if (flag_occ) then
240.    1 2 M mV                         intfact=abs((wxi(iw)+cedifft_zb_right)/(wxi(iw)+cedifft_zb_left))
241.    1 2 M mV                         if (intfact .lt. 1d-4) intfact = 1d-4
242.    1 2 M mV                         if (intfact .gt. 1d4) intfact = 1d4
243.    1 2 M mV                         intfact = log(intfact)
244.    1 2 M mV                         schDt_lin3 = (schDt_left + schDt_lin2*(-wxi(iw)-cedifft_zb_left))*intfact
245.    1 2 M mV                       else
246.    1 2 M mV                         schDt_lin3 = (schDt_left + schDt_lin2*(wxi(iw)-cedifft_zb_left))*intfact
247.    1 2 M mV                       endif
248.    1 2 M mV                       schDt_lin3 = schDt_lin3 + schDt_lin
249.    1 2 M mV                       schDi_cor(iw) = schDi_cor(iw) - CMPLX(0.d0,prefactor) * schDt_lin3
250.    1 2 M mV---->>               enddo
251.    1 2                        endif
252.    1 2---------->           enddo
```

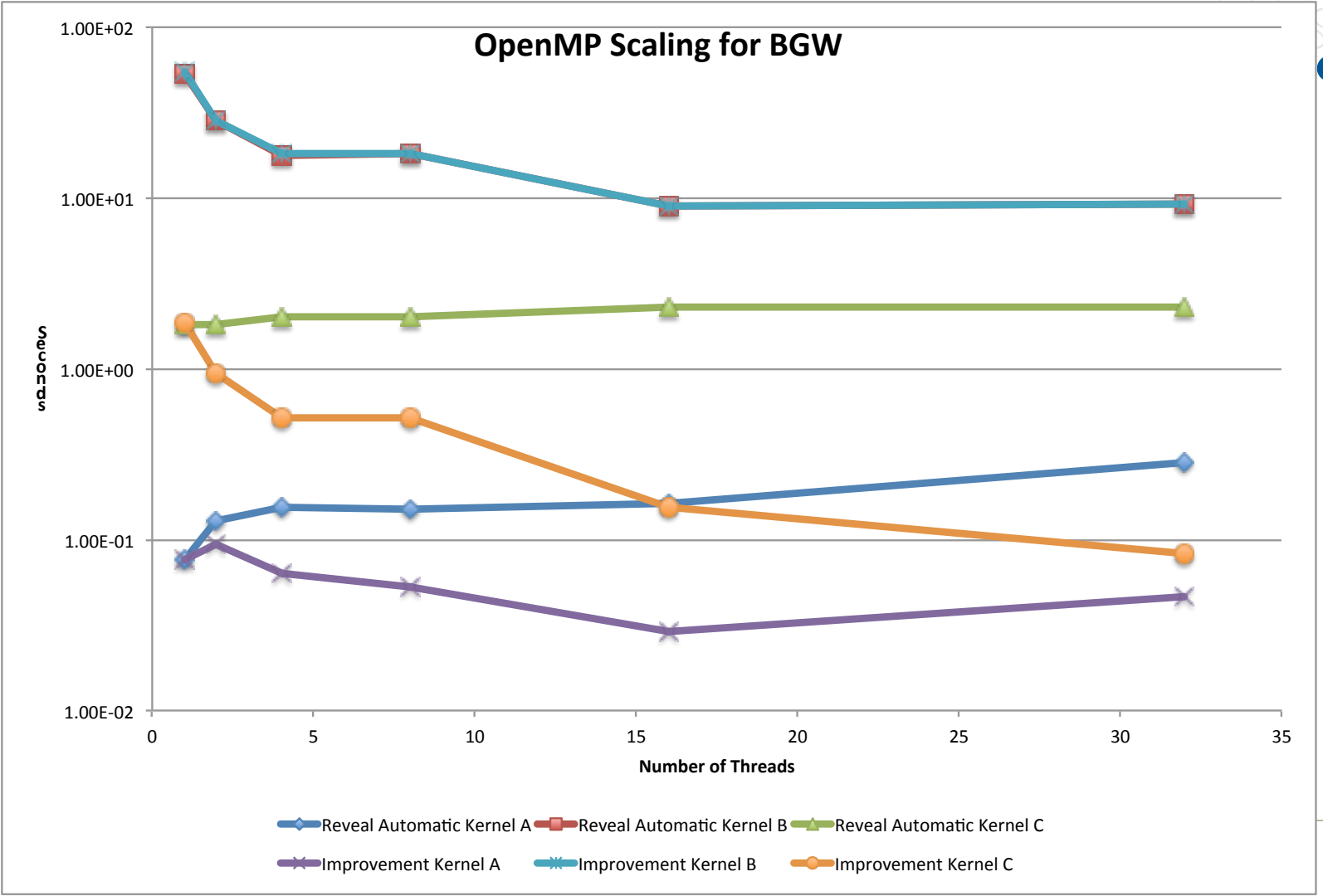
Kernel C

Automatically modified by Levesque

```
192.    1              ! Directive inserted by Cray Reveal.  May be incomplete.
193.    1 M----------< !$OMP  parallel do default(none)                                &
194.    1 M            !$OMP&   private (cedifft_coh,cedifft_zb,cedifft_zb_left,      &
195.    1 M            !$OMP&            cedifft_zb_right,ifreq,intfact,iw,schdt,schdt_avg,     &
196.    1 M            !$OMP&            schdt_left,schdt_lin,schdt_lin2,schdt_lin3,  &
197.    1 M            !$OMP&            schdt_right)                                &
198.    1 M            !$OMP&   shared  (cedifft_cor,dfreqbrd,dfreqgrid,flag_occ,nfreq,       &
199.    1 M            !$OMP&            nfreqeval,pref,prefactor,sch2di,schdi,schdi_cor,      &
200.    1 M            !$OMP&            schdi_corb,schdt_array,wxi)
201.    1 M mV-------<         do iw = 1, nfreqeval
202.    1 M mV 4-----<           do ifreq=1,nFreq
203.    1 M mV 4
204.    1 M mV 4                   schDt = schDt_array(ifreq)
205.    1 M mV 4
206.    1 M mV 4                   cedifft_zb = dFreqGrid(ifreq)
207.    1 M mV 4                   cedifft_coh = CMPLX(cedifft_zb,0D0)- dFreqBrd(ifreq)
208.    1 M mV 4
209.    1 M mV 4                   if (ifreq .ne. 1) then
210.    1 M mV 4                     cedifft_zb_right = cedifft_zb
211.    1 M mV 4                     cedifft_zb_left = dFreqGrid(ifreq-1)
212.    1 M mV 4                     schDt_right = schDt
213.    1 M mV 4                     schDt_left = schDt_array(ifreq-1)
214.    1 M mV 4                     schDt_avg = 0.5D0 * ( schDt_right + schDt_left )
215.    1 M mV 4                     schDt_lin = schDt_right - schDt_left
216.    1 M mV 4                     schDt_lin2 = schDt_lin/(cedifft_zb_right-cedifft_zb_left)
217.    1 M mV 4                   endif
218.    1 M mV 4
219.    1 M mV 4       ! The below two lines are for sigma1 and sigma3
220.    1 M mV 4                   if (ifreq .ne. nFreq) then
221.    1 M mV 4                     schDi(iw) = schDi(iw) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(iw)-cedifft_coh)
222.    1 M mV 4                     schDi_corb(iw) = schDi_corb(iw) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(iw)-cedifft_cor)
223.    1 M mV 4                   endif
224.    1 M mV 4                   if(ifreq.ne.1)then
```

Kernel C

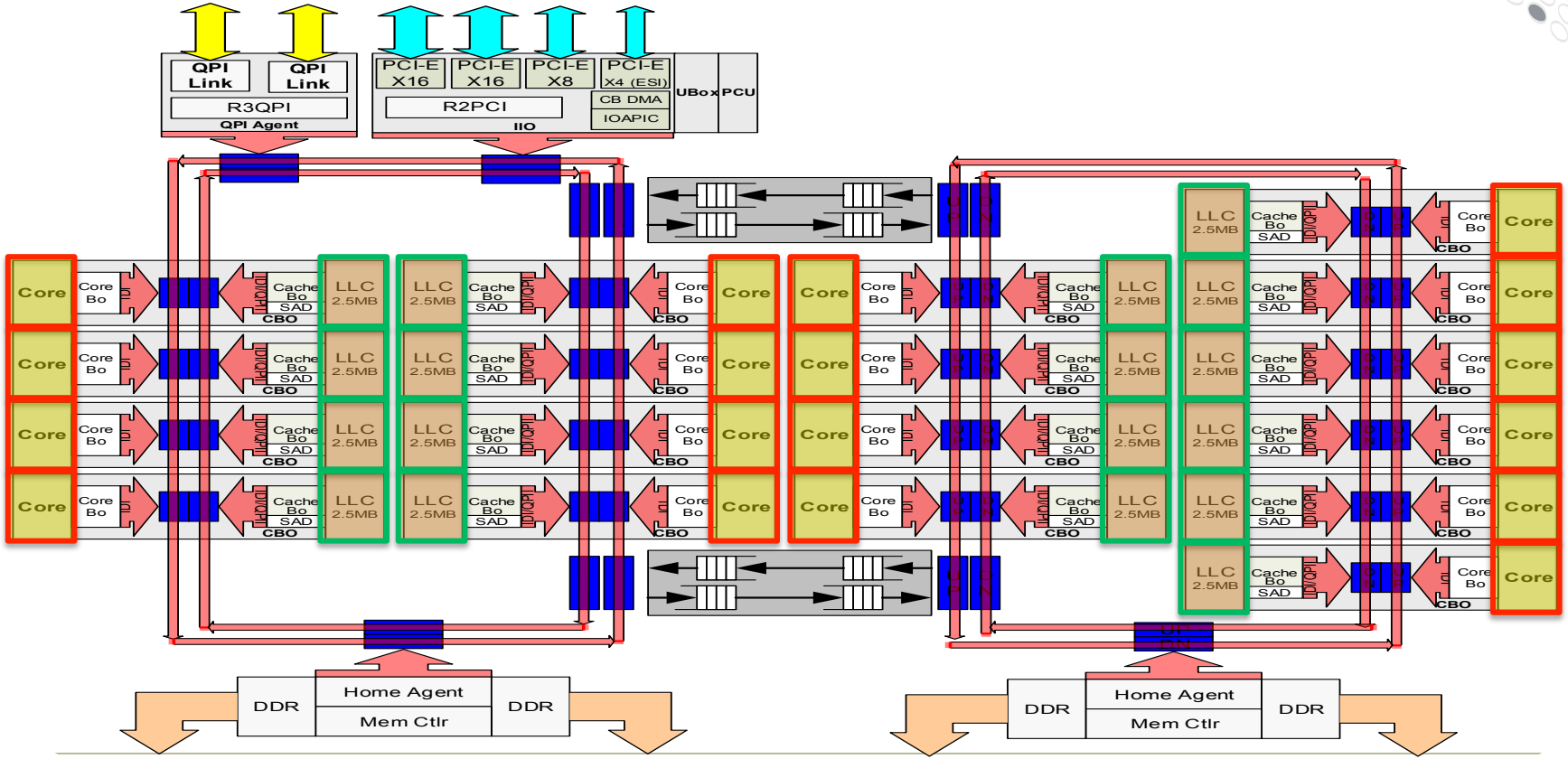
```
219.    1 M mV 4       ! The below two lines are for sigma1 and sigma3
220.    1 M mV 4                    if (ifreq .ne. nFreq) then
221.    1 M mV 4                      schDi(iw) = schDi(iw) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(iw)-cedifft_coh)
222.    1 M mV 4                      schDi_corb(iw) = schDi_corb(iw) - CMPLX(0.d0,pref(ifreq)) * schDt / ( wxi(iw)-cedifft_cor)
223.    1 M mV 4                    endif
224.    1 M mV 4                    if(ifreq.ne.1)then
225.    1 M mV 4       !These lines are for sigma2
226.    1 M mV 4                       intfact=abs((wxi(iw)-cedifft_zb_right)/(wxi(iw)-cedifft_zb_left))
227.    1 M mV 4                       if (intfact .lt. 1d-4) intfact = 1d-4
228.    1 M mV 4                       if (intfact .gt. 1d4) intfact = 1d4
229.    1 M mV 4                       intfact = -log(intfact)
230.    1 M mV 4                       sch2Di(iw) = sch2Di(iw) - CMPLX(0.d0,prefactor) * schDt_avg * intfact
231.    1 M mV 4       !These lines are for sigma4
232.    1 M mV 4                       if (flag_occ) then
233.    1 M mV 4                         intfact=abs((wxi(iw)+cedifft_zb_right)/(wxi(iw)+cedifft_zb_left))
234.    1 M mV 4                         if (intfact .lt. 1d-4) intfact = 1d-4
235.    1 M mV 4                         if (intfact .gt. 1d4) intfact = 1d4
236.    1 M mV 4                         intfact = log(intfact)
237.    1 M mV 4                         schDt_lin3 = (schDt_left + schDt_lin2*(-wxi(iw)-cedifft_zb_left))*intfact
238.    1 M mV 4                       else
239.    1 M mV 4                         schDt_lin3 = (schDt_left + schDt_lin2*(wxi(iw)-cedifft_zb_left))*intfact
240.    1 M mV 4                       endif
241.    1 M mV 4                       schDt_lin3 = schDt_lin3 + schDt_lin
242.    1 M mV 4                       schDi_cor(iw) = schDi_cor(iw) - CMPLX(0.d0,prefactor) * schDt_lin3
243.    1 M mV 4                    endif
244.    1 M mV 4----->           enddo
245.    1 M mV------>>         enddo
```


OpenMP Scaling for BGW
1.00E+02
1.00E+01
1.00E+00
1.00E-01
1.00E-02
Seconds
0
5
10
15
20
25
30
35
Number of Threads
Reveal Automatic Kernel A
Reveal Automatic Kernel B
Reveal Automatic Kernel C
Improvement Kernel A
Improvement Kernel B
Improvement Kernel C

Can we take clues from all-MPI advantages and disadvantages?

- **Can we force locality like MPI does?**
  - This requires that the thread that uses the data, allocates the data within its NUMA region
  - Tradition OpenMP has no notion of locality
- **Can we allow threads to work asynchronously?**
  - Tradition OpenMP implies barriers after a parallel region
  - Loop level parallelism forces too much synchronization
  - With SPMD OpenMP, user has complete control over synchronization
- **Can we somehow control scheduling of the threads to enable more dynamic re-distribution of work**
  - With SPMD OpenMP, user can write sophistication thread scheduling routine

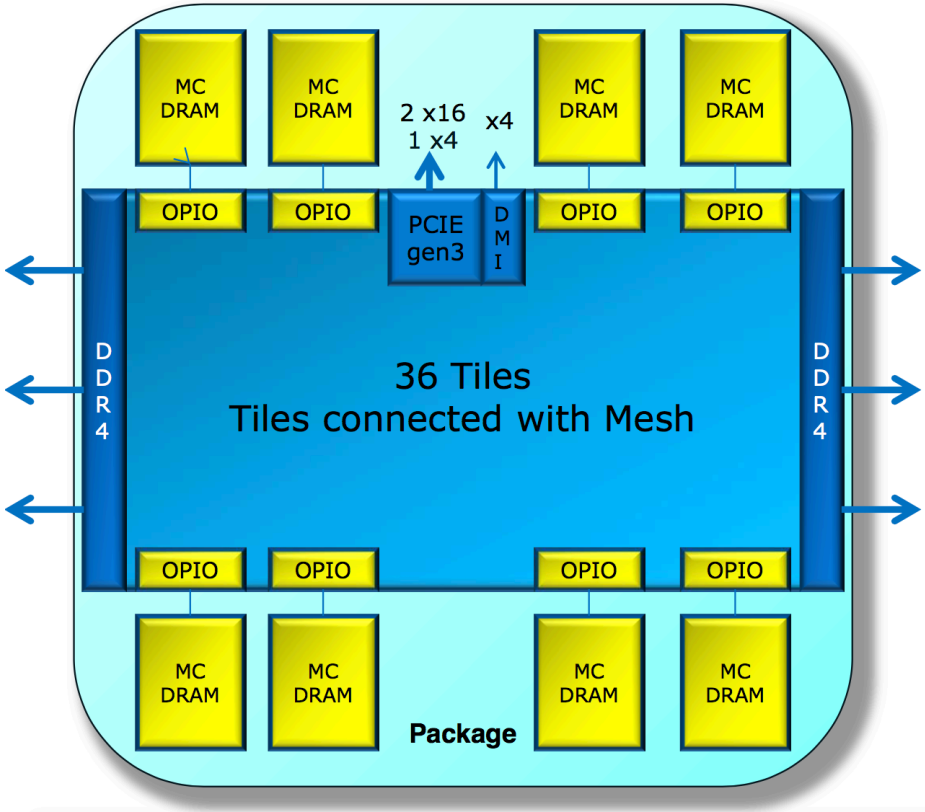
Haswell Uncore Structure



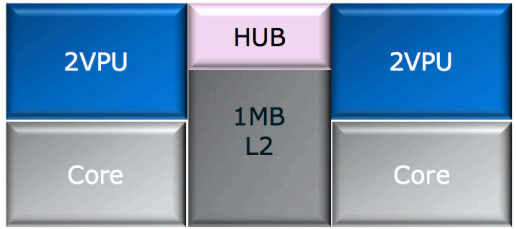
Haswell Node Memory Structure

- **With two socket node**
  - Within a socket
    - Each core has its own L1 cache
    - Two cores share a L2 cache
    - Last Level Cache (L3) is shared by all cores
  - Significant NUMA effects across the two sockets
    - Probably want at least two MPI tasks

Knights Landing Processor Overview



TILE:

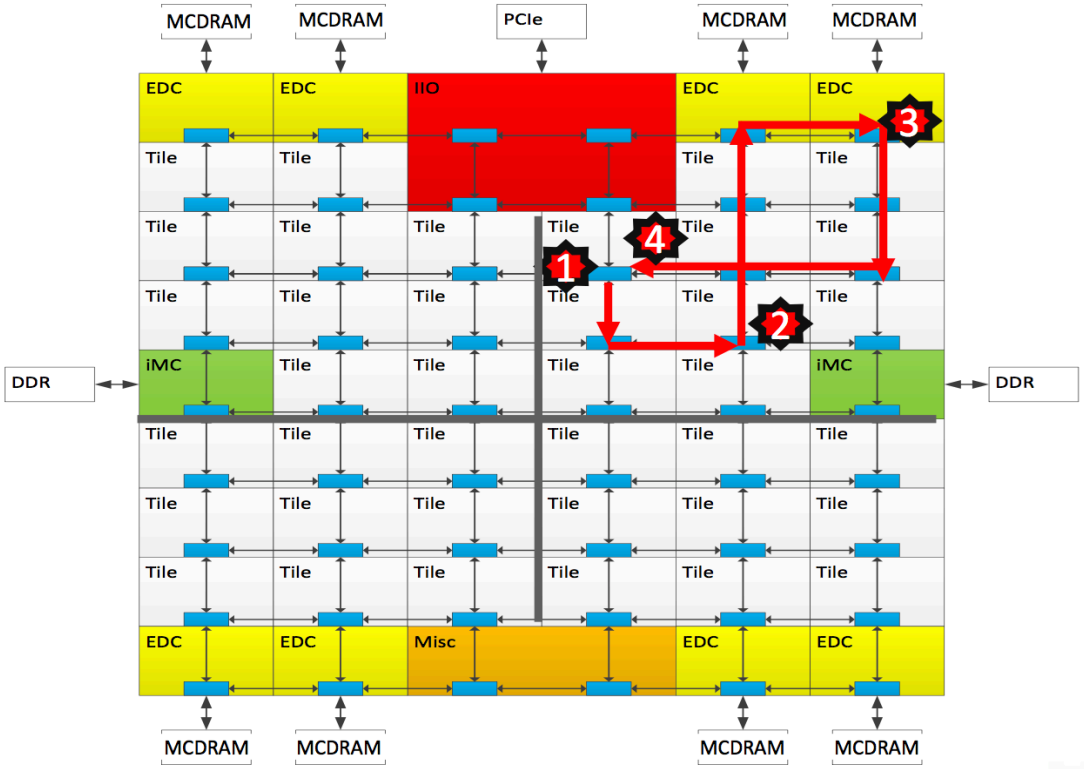


- 72 cores *based on Silvermont core in today's Intel® Atom® processors with significant enhancements for HPC*
  - OOO Execution
  - Back to Back fetch and issue per thread
  - AVX1, AVX2 and AVX512 ISA
- 2 Vector Processing Units per core
- Full Intel® Xeon® processor ISA compatibility (except TSX)
- 6 channels of DDR4 2400 up to 384GB
- 8GB/16GB of high bandwidth on-package memory
- 36 lanes PCIE Gen 3
- ~ 2.5 TF DGEMM
- ~480 GB/s Stream Triad
- ~3x Single thread perf over KNC
- TDP: 200W

KNL node memory structure

- **Very complicated – different clustering modes**
  - All to all
  - Quadrant mode
  - SNC2,SNC4
- **As with Haswell**
  - Each core has L1 Cache
  - Two cores share L2 Cache
  - No L3 Cache
    - Can use MCDRAM as Last Level Cache
  - Due to 2-D mesh and thread placement some NUMA effects

Cluster Mode: Sub-NUMA Clustering (SNC)



Each Quadrant (Cluster) exposed as a separate NUMA domain to OS.

Looks analogous to 4-Socket Xeon

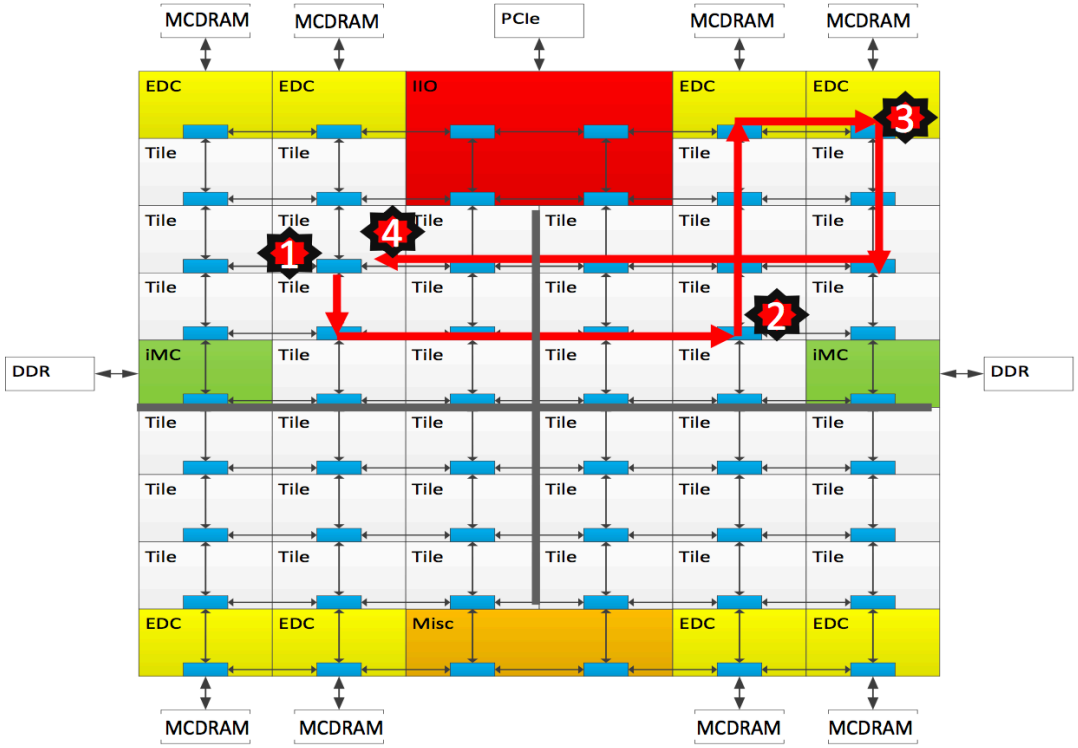
Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

SW needs to NUMA optimize to get benefit.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Cluster Mode: Quadrant



Chip divided into four virtual Quadrants

Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Finding the MPI Sweet Spot

- **The follow chart gives the performance of the Himeno benchmark, which is very memory bandwidth limited**
  - First we only run with MPI across four nodes, keeping the grid size constant (Strong Scaling) we increase the total number of MPI tasks from 4 up to 128
    - Important to assure that equal number of MPI tasks are allocated on each socket – where X is the number of MPI tasks on a node
      - aprun –n 4*X –N X –S X/2
  - Then for each MPI run we add OpenMP threads (the traditional method)
  - The plot gives MFLOPS and higher is better
  - The traditional OpenMP was done extremely well using first touch to have the threads allocate the data within its closest memory

First Touch – Initialize the data with OpenMP

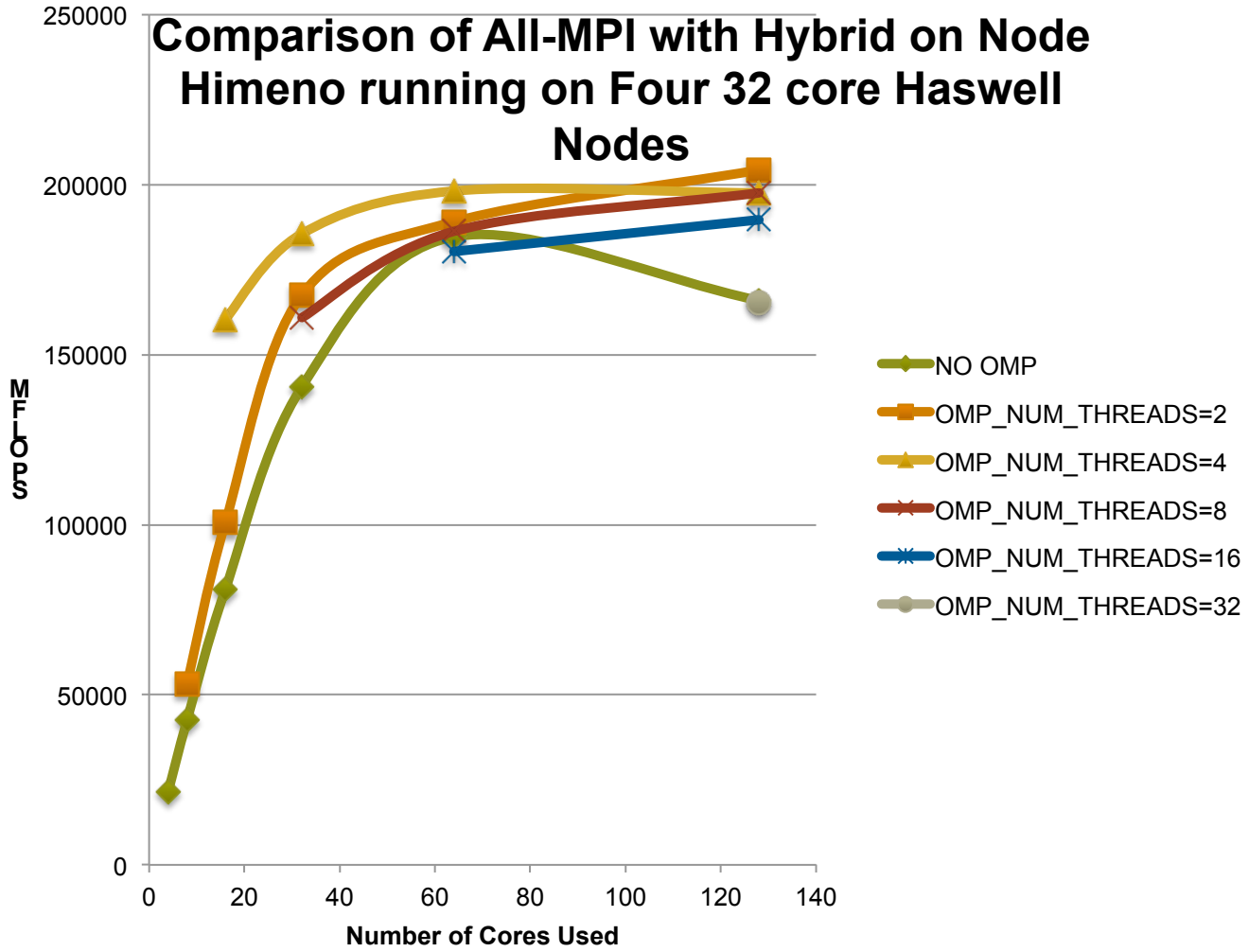
```
151.     M--------------< !$OMP PARALLEL
152.     M                !$OMP DO
153.   + M m------------<       do k=1,mkmax
154.     M m C----------<          do j=1,mjmax
155.     M m C VCr2-----<             do i=1,mimax
156.     M m C VCr2                      a(i,j,k,1)=0.0
157.     M m C VCr2                      a(i,j,k,2)=0.0
158.     M m C VCr2                      a(i,j,k,3)=0.0
159.     M m C VCr2                      a(i,j,k,4)=0.0
160.     M m C VCr2                      b(i,j,k,1)=0.0
161.     M m C VCr2                      b(i,j,k,2)=0.0
162.     M m C VCr2                      b(i,j,k,3)=0.0
163.     M m C VCr2                      c(i,j,k,1)=0.0
164.     M m C VCr2                      c(i,j,k,2)=0.0
165.     M m C VCr2                      c(i,j,k,3)=0.0
166.     M m C VCr2 A--<>                p(i,j,k)=0.0
167.     M m C VCr2 A--<>                wrk1(i,j,k)=0.0
168.     M m C VCr2 A--<>                wrk2(i,j,k)=0.0
169.     M m C VCr2 A--<>                bnd(i,j,k)=0.0
170.     M m C VCr2----->             enddo
171.     M m C---------->          enddo
172.     M m------------>       enddo
```

The major computational loop

```
209.  + 1--------------<       DO loop=1,nn
210.    1                     gosa=0.0
211.    1                     wgosa=0.0
212.  + 1                !$OMP PARALLEL DO PRIVATE(S0,SS) REDUCTION(+:WGOSA)
213.  + 1 m------------<       DO K=2,kmax-1
214.  + 1 m 3----------<          DO J=2,jmax-1
215.    1 m 3 Vr3------<             DO I=2,imax-1
216.    1 m 3 Vr3                      S0=a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K)
217.    1 m 3 Vr3            1              +a(I,J,K,3)*p(I,J,K+1)
218.    1 m 3 Vr3            2              +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K)
219.    1 m 3 Vr3            3              -p(I-1,J+1,K)+p(I-1,J-1,K))
220.    1 m 3 Vr3            4              +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1)
221.    1 m 3 Vr3            5              -p(I,J+1,K-1)+p(I,J-1,K-1))
222.    1 m 3 Vr3            6              +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1)
223.    1 m 3 Vr3            7              -p(I+1,J,K-1)+p(I-1,J,K-1))
224.    1 m 3 Vr3            8              +c(I,J,K,1)*p(I-1,J,K)+c(I,J,K,2)*p(I,J-1,K)
225.    1 m 3 Vr3            9              +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
226.    1 m 3 Vr3                      SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
227.    1 m 3 Vr3                      WGOSA=WGOSA+SS*SS
228.    1 m 3 Vr3                      wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
229.    1 m 3 Vr3------>             enddo
230.    1 m 3---------->          enddo
231.    1 m------------>       enddo
232.    1                   C
```

And then the communication

```
1                C
  233.  + 1                !$OMP PARALLEL DO
  234.  + 1 m------------<          DO K=2,kmax-1
  235.  + 1 m 3----------<             DO J=2,jmax-1
  236.    1 m 3 A--------<                DO I=2,imax-1
  237.    1 m 3 A                           p(I,J,K)=wrk2(I,J,K)
  238.    1 m 3 A-------->                enddo
  239.    1 m 3---------->             enddo
  240.    1 m------------>          enddo
  241.    1                C
  242.  + 1                          call sendp(ndx,ndy,ndz)
  243.    1                C
  244.  + 1                          call mpi_allreduce(wgosa,
  245.    1                     >                       gosa,
  246.    1                     >                       1,
  247.    1                     >                       mpi_real4,
  248.    1                     >                       mpi_sum,
  249.    1                     >                       mpi_comm_world,
  250.    1                     >                       ierr)
  251.    1                C
  252.    1-------------->       enddo
```


Comparison of All-MPI with Hybrid on Node
Himeno running on Four 32 core Haswell Nodes
250000
200000
150000
100000
50000
0
MFLOPS
0
20
40
60
80
100
120
140
Number of Cores Used
NO OMP
OMP_NUM_THREADS=2
OMP_NUM_THREADS=4
OMP_NUM_THREADS=8
OMP_NUM_THREADS=16
OMP_NUM_THREADS=32

Analysis of Results

- **All MPI works very well, until we cross the 16 tasks/node at which point we saturate the memory bandwidth**
- **Four MPI tasks and all OpenMP threads on the node has identical performance**
- **Simply using 16 MPI tasks/node and 2 threads gives use the best performance**
  - This makes sense, given the NUMA architecture of the Haswell
    - Two cores sharing Level 2 cache

On multi-core and many-core systems

- **We will want to run as many MPI tasks that gives us a GOOD performance increase**
  - Then we want to add threads to achieve as much parallelism on the node as possible
  - Ideally if we use at least one MPI task on each NUMA region, the treading within the NUMA region will not suffer from NUMA effects.
  - Still efficient threading is required
    - Numerous parallel regions are expensive, low level, low granularity loops will not return much performance
    - High level parallel loops are preferred
      - More difficult to scope
        - Probably want to find out about Reveal

SPMD OpenMP

- **Less code modifications**
- **Requires a better understanding of array usage**
  - Reveal will not help
- **Basically**
  - Treat threads like MPI tasks
    - User must assure synchronization is correct
      - Required when threads need to communication with each other
    - User must distribute work to threads
      - This is a good thing, effective way of dealing with load imbalance

SPMD OpenMP

- **Introduce a high level !$OMP PARALLEL region**
  - Down the call chain the user is responsible for managing threads
    - Assuring shared data is shared as in the Fortran/C/C++ convention
    - Assuring private data is private (allocated on stack) as in the Fortran convention
    - Managing the assignment of work to each of the threads with either a simple chunking method of a more sophistication dynamic work assignment given the measured workload
    - Synchronizing the threads around data dependencies, MPI calls, etc
      - Consider having each thread do its own message passing

What if you need a shared variable down the call chain

```
Subroutine within_a_parallel( )

Real, pointer ::  shared_p( : )

!$omp single
allocate(shared_p(0:100))
!$omp end single copyprivate(share_p)

…

End subroutine
```

A Simple Example HIMENO

- Single computational routine that does a nine point stencil
- Simulated Jacobi solver
- Stencil loop with reduction operator
- SPMD OpenMP does not outperform the traditional approach for this example; because, the traditional approach:
  - Does first touch
  - Maintains a consistent memory usage across threads
  - Unless something is done with MPI, the barriers cannot be eliminated.
- So this is just a demonstration of how to implement SPMD OpenMP

High level routine

```
!$OMP PARALLEL private(cpu0,cpu1,cpu)
C     Initializing matrixes
      call initmt
!      write(*,*) ' mimax=',mimax,' mjmax=',mjmax,' mkmax=',mkmax
!      write(*,*) ' imax=',imax,' jmax=',jmax,' kmax=',kmax
C     Start  measuring
c
      cpu0=sec_timer()
C
C     Jacobi iteration
      call jacobi(nn,gosa)
C
      cpu1= sec_timer()-cpu0
      cpu = cpu1
!$OMP MASTER
      write(*,*) 'cpu : ',cpu,'sec.'
        nflop=(kmax-2)*(jmax-2)*(imax-2)*34
        if(cpu1.ne.0.0) xmflops2=nflop/cpu*1.0e-6*float(nn)
        write(*,*) ' Loop executed for ',nn,' times'
        write(*,*) ' Gosa :',gosa
        write(*,*) ' MFLOPS measured :',xmflops2
        score=xmflops2/32.27
        write(*,*) ' Score based on MMX Pentium 200MHz :',score
!$OMP END MASTER
!$OMP END PARALLEL
```

Initialization loop

```
call get_bounds(1,kmax,nlo,nhi)
do k=nlo,nhi
  do j=1,jmax
    do i=1,imax
      a(i,j,k,1)=1.0
      a(i,j,k,2)=1.0
      a(i,j,k,3)=1.0
      a(i,j,k,4)=1.0/6.0
      b(i,j,k,1)=0.0
      b(i,j,k,2)=0.0
      b(i,j,k,3)=0.0
      c(i,j,k,1)=1.0
      c(i,j,k,2)=1.0
      c(i,j,k,3)=1.0
      p(i,j,k)  =float((k-1)*(k-1))/float((kmax-1)*(kmax-1))
      wrk1(i,j,k)=0.0
      bnd(i,j,k)=1.0
    enddo
  enddo
enddo
```

get_bounds

```
      subroutine get_bounds(innlo,innhi,nlo,nhi)
      integer innlo,innhi,nlo,nhi
      integer num_threads,my_thread_id
#ifdef _OPENMP
      my_thread_id = omp_get_thread_num()
      num_threads = omp_get_num_threads()
      nlo = innlo +((innhi-innlo+1)/num_threads+1)*(my_thread_id)
      nhi = min(innhi,nlo+(innhi-innlo+1)/num_threads)
#else
      nlo = innlo
      nhi = innhi
#endif
      end
```

Major Computational Loop with SPMD OpenMP

```
      call get_bounds(1,kmax-1,nlo,nhi)
      do 900 loop=1,nn
      my_gosa=0.0
      gosa=0.0
!$OMP BARRIER
        DO 100 K=nlo,nhi
        DO 100 J=2,jmax-1
        DO 100 I=2,imax-1
          S0=a(I,J,K,1)*p(I+1,J,K)+a(I,J,K,2)*p(I,J+1,K)
     1      +a(I,J,K,3)*p(I,J,K+1)
     3      +b(I,J,K,1)*(p(I+1,J+1,K)-p(I+1,J-1,K)
     *                  -p(I-1,J+1,K)+p(I-1,J-1,K))
     4      +b(I,J,K,2)*(p(I,J+1,K+1)-p(I,J-1,K+1)
     *                  -p(I,J+1,K-1)+p(I,J-1,K-1))
     5      +b(I,J,K,3)*(p(I+1,J,K+1)-p(I-1,J,K+1)
     *                  -p(I+1,J,K-1)+p(I-1,J,K-1))
     6      +c(I,J,K,1)*p(I-1,J,K)+c(I,J,K,2)*p(I,J-1,K)
     *      +c(I,J,K,3)*p(I,J,K-1)+wrk1(I,J,K)
          SS=(S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
          my_GOSA=my_GOSA+SS*SS
          wrk2(I,J,K)=p(I,J,K)+OMEGA *SS
  100   CONTINUE
```

```
!$OMP CRITICAL
        GOSA =GOSA + my_GOSA
!$OMP END CRITICAL
!$OMP BARRIER
        DO 200 K=nlo,nhi
        DO 200 J=2,jmax-1
        DO 200 I=2,imax-1
          p(I,J,K)=wrk2(I,J,K)
  200   CONTINUE
  900 continue
```

ISSUES:

- my_gosa must be private
- gosa must be shared
- s0,ss must be private
- a,b,c,p,bnd,wrk1,wrk2 must be shared
- BARRIERs and CRITICAL region are required

What if you need a shared variable down the call chain

```
Subroutine Jacobi

Real, pointer ::  gosa

!$omp single
allocate(gosa)
!$omp end single copyprivate(gosa)

…

End subroutine
```

When we have MPI we need to BARRIER

```
!$OMP CRITICAL
      WGOSA = WGOSA +l_wgosa
!$OMP END CRITICAL
      DO K=max(nlo,2),min(nhi,kmax-1)
        DO J=2,jmax-1
          DO I=2,imax-1
            p(I,J,K)=wrk2(I,J,K)
          enddo
        enddo
      enddo
!$OMP BARRIER
!$OMP MASTER
      call sendp(ndx,ndy,ndz)
      call mpi_allreduce(wgosa,
     >                   gosa,
     >                   1,
     >                   mpi_real4,
     >                   mpi_sum,
     >                   mpi_comm_world,
     >                   ierr)
      wgosa=0.0
!$OMP END MASTER
!$OMP BARRIER
```

Nekbone

```
!$OMP PARALLEL PRIVATE(ur,us,ut,wk,i,ii,inn,nlo,nhi,tmp,iter,rtz2,beta,
!$OMP&                 alpha,alphm)
      do iter=1,miter
         rtz2 = rtz1
!$OMP BARRIER
!$OMP MASTER
         call solveM(z,r,n)     ! preconditioner here
         pap = 0.0
         rtr = 0.0
         rtz1 = 0.0
!$OMP END MASTER
!$OMP BARRIER
         call get_bounds(1, nelt, nlo, nhi)
         tmp = 0.0
         do ii = nlo,nhi
         do i = 1, nn
         inn = (ii-1)*nn + i
            tmp = tmp + r(inn)*c(inn)*z(inn)
         enddo
         enddo
!$OMP CRITICAL
      rtz1 = rtz1 + tmp
!$OMP END CRITICAL
!$OMP BARRIER
!$OMP MASTER
      call gop(rtz1,work,'+  ',1) ! sum-reduce the scalar tmp across all PE
!$OMP END MASTER
!$OMP BARRIER
```

Nekbone CG Calls AX

```
            if (iter.eq.1) rlim2 = rtr*eps**2
            beta = rtz1/rtz2
            if (iter.eq.1) beta=0.0
            do ii = nlo,nhi
            do i = 1, nn
            inn = (ii-1)*nn + i
            p(inn)=beta*p(inn)+z(inn)   !call add2s1(p,z,beta,n)                              ! 2n
            enddo
            enddo
            call ax(w,p,g,ur,us,ut,wk,n)                                        ! flopa
            tmp = 0.0
            do ii = nlo,nhi
            do i = 1, nn
            inn = (ii-1)*nn + i
               tmp = tmp + w(inn)*c(inn)*p(inn)
            enddo
            enddo
!$OMP CRITICAL
      pap = pap + tmp
!$OMP END CRITICAL
!$OMP BARRIER
!$OMP MASTER
      call gop(pap,work,'+  ',1) ! sum-reduce the scalar tmp across all PE
!$OMP END MASTER
!$OMP BARRIER
```

Nekbone

```
            alpha=rtz1/pap
            alphm=-alpha
            call get_bounds(1, nelt, nlo, nhi)
            do ii = nlo,nhi
            do i = 1, nn
            inn = (ii-1)*nn + i
            x(inn)=x(inn) + alpha*p(inn)    !call add2s2(x,p,alpha,n)                    ! 2n
            r(inn) = r(inn) + alphm*w(inn)  !call add2s2(r,w,alphm,n)                    ! 2n
            enddo
            enddo
            tmp = 0.0
            do ii = nlo,nhi
            do i = 1, nn
            inn = (ii-1)*nn + i
               tmp = tmp + r(inn)*c(inn)*r(inn)
            enddo
            enddo
!$OMP CRITICAL
      rtr = rtr + tmp
!$OMP END CRITICAL
!$OMP BARRIER
!$OMP MASTER
      call gop(rtr,work,'+  ',1) ! sum-reduce the scalar tmp across all PE
         if (iter.eq.1) rlim2 = rtr*eps**2
         if (iter.eq.1) rtr0  = rtr
         rnorm = sqrt(rtr)
        if (nid.eq.0.and.mod(iter,100).eq.0)
     $      write(6,6) iter,rnorm,pap
   6    format('cg:',i4,1p4e12.4)
!$OMP END MASTER
!       if (rtr.le.rlim2) goto 1001

      enddo

 1001 continue
!$OMP END PARALLEL
```

Nekbone CG calls AX

```
      subroutine ax(w,u,gxyz,ur,us,ut,wk,n) ! Matrix-vector product: w=A*u
      include 'SIZE'
      include 'TOTAL'
      real w(nx1*ny1*nz1,nelt),u(nx1*ny1*nz1,nelt)
      real gxyz(2*ldim,nx1*ny1*nz1,nelt)
      parameter (lt=lx1*ly1*lz1*lelt)
      real ur(lt),us(lt),ut(lt),wk(lt)
      common /mymask/cmask(-1:lx1*ly1*lz1*lelt)

      integer e
      call get_bounds(1, nelt, nlo, nhi)
      do e=nlo, nhi                                    ! ~
         call ax_e( w(1,e),u(1,e),gxyz(1,1,e)     ! w   = A  u
     $                             ,ur,us,ut,wk) !  L     L  L
      enddo                                        !
!$OMP BARRIER
!$OMP MASTER
      call dssum(w)          ! Gather-scatter operation  ! w   = QQ  w
!$OMP END MASTER
!$OMP BARRIER
                                                              !            L
      do e=nlo, nhi                                    ! ~
      w(:,e) = w(:,e) + 0.1*u(:,e)    !call add2s2(w,u,.1,n)      !2n
      enddo
!$OMP BARRIER
!$OMP MASTER
      call maskit(w,cmask,nx1,ny1,nz1)  ! Zero out Dirichlet conditions
!$OMP END MASTER
!$OMP BARRIER

      nxyz=nx1*ny1*nz1
      flop_a = flop_a + (19*nxyz+12*nx1*nxyz)*nelt

      return
      end
```

Nekbone AX Calls AX1

```
c-----------------------------------------------------------------------
      subroutine ax1(w,u,n)
      include 'SIZE'
      real w(n),u(n)
      real h2i

      h2i = (n+1)*(n+1)
      do i = 2,n-1
         w(i)=h2i*(2*u(i)-u(i-1)-u(i+1))
      enddo
      w(1)  = h2i*(2*u(1)-u(2  ))
      w(n)  = h2i*(2*u(n)-u(n-1))

      return
      end
```

Running Nekbone on 8 Nodes of Haswell

Number of MPI Tasks	Number of Threads	GFLOPS
32 (4/Node)	1	.32
32(4/Node)	2	.927
32(4/Node)	4	1.35
32(4/Node)	8	2.35

Running CoMD on 1 Nodes of Haswell

Number of MPI Tasks	Number of Threads	Grind Time
1	1	3.17
1	2	1.70
1	4	.92
1	8	.56

This is a C mini-app

SPMD OpenMP Pros

- Much less overhead than low level !$OMP PARALLEL DO and !$OMP DO
- Scoping is handled with language standard
- In very complicated applications, the number of directives and code modifications is much less

SPMD OpenMP Cons

- Users must really understand threading and the implications of how the language handles global and local storage which would become shared by all threads and private to a thread.

HBM code modernization -

insights from a Xeon Phi experiment

Jacob Weismann Poulsen, DMI, Denmark
Per Berg, DMI, Denmark
Karthik Raman, Intel, USA

High-level OpenMP and Thread Scalable MPI-RMA:

Application Study with the Wombat Astrophysical MHD Code

Dr. Peter Mendygral
Cray Inc.

What is Wombat?

- **Designed to simulate astrophysical plasmas with ideal magnetohydrodynamics (MHD)**
- **Uses shock capturing 2nd order accurate method**
  - Total variation diminishing (TVD) Roe-type solver of Ryu & Jones (1995)
  - Directionally un-split solver based on the CTU-CT method of Gardiner & Stone (2005)
  - Divergence of magnetic field maintained with constrained transport (CT) staggered grid solver
  - Team members have developed a 5th order directionally un-split WENO+CT scheme that will be integrated later this year
- **"Basic" version includes passive advection solver based on CTU implementation of MUSCL**
  - Arbitrary number of passive scalars can be advected with the flow

Application Design

Primary Development Concerns

- **Two main issues drove the design of Wombat and explain why other codes have not been sufficient for the science**
  - Scaling to extreme core count required for resolution requirements
  - Load balancing for SMR/AMR and dark matter particles

- **The approach to these problems in Wombat was**
  - Make communication matter as little as possible
  - Wide OpenMP on a node to soften impacts of load imbalances (and hardware imbalances) as much as possible before communicating work between ranks
  - Data structures that reduce AMR/SMR complexity and avoid significant global communication for refined patch tracking
  - Do it all in Fortran as that's what works best for me
    - Wombat uses object oriented features from Fortran 2003 and 2008

High-level OpenMP

- Two choices (could be used at the same time) for threading the Domain/Patch design

Option A

```
! Move OpenMP near the top of the call stack

!#OMP PARALLEL
DO WHILE (t .LT. tend)

   !#OMP DO SCHEDULE(GUIDED)
   DO patch = 1, npatches

      CALL update_patch()

      ! All threads drive MPI

   END DO

END DO
```

Option B

```
! Keep OpenMP within a “compute” loop

DO WHILE (t .LT. tend)

   DO patch = 1, npatches

      CALL update_patch()

      ! MPI driven by single thread

   END DO

END DO

SUBROUTINE update_patch()

   !$OMP PARALLEL DO SCHEDULE(STATIC)
   DO i = 1, nx
   …do work…
   END DO
```

High-level OpenMP

- **Benefits of OpenMP near the top of the stack**
  - Application more closely mimics completely independent processes
    - Less likely to be in the same portion of code at the same time
    - Bandwidth competition may decrease
    - Amdahl's law
  - Threads are less coupled => infrequent thread synchronization
  - Much less likely to have issues with memory conflicts between threads
  - Simpler to implement when done right
    - Large reduction in the amount of OpenMP directives
    - Very little variable scoping needed as most everything is shared => reduced memory footprint

Wombat Driver and Parallel Region

Setup and object constructors

Thread parallel region

Array allocation and initialization

Time step loop

DomainSolver%solve(mhd)
DomainSolver%solve(ct)
DomainSolver%solve(passive)
DomainSolver%solve(particle kick+drift)
FMGSolver%solve(all levels)
DomainSolver%solve(particle kick)

I/O data dump(s)

Update time step

Array cleanup

Object destructors

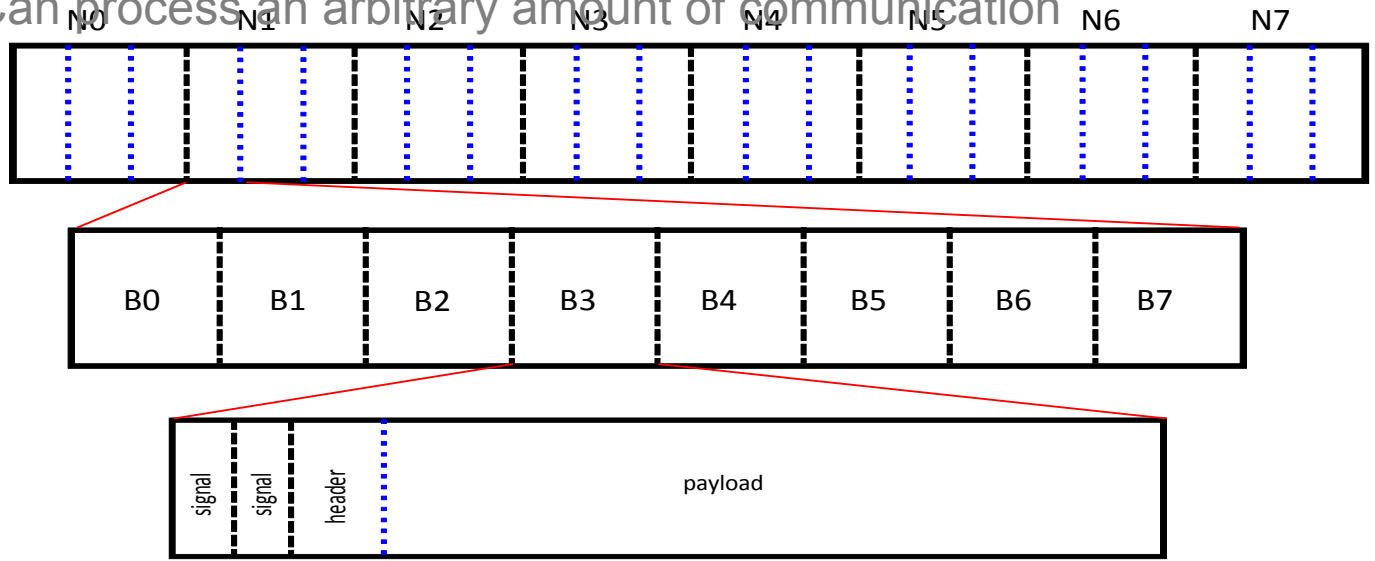
Simulation complete

Communication Concerns

- **If a rank is made much wider with threads, serialization around MPI will limit thread scaling and overall performance**
  - Nearly all MPI libraries implement thread safety with a global lock
  - Cray is addressing this issue
    - Released per-object lock library
    - Threading enhancements under design now for two-sided (released per-object lock library a first step)
- **Wide OpenMP also means more communication to process**
  - Every Patch now has its own smaller boundaries to communicate
  - Starts tipping the behavior towards the message rate limit
  - Two-sided tag matching cannot be done in parallel and will limit thread scaling
    - May start hitting tag limit
- **Slower serial performance of KNL => maybe look for the lightest weight MPI layer available**
  - MPI-RMA over DMAPP on Cray systems is a thin software layer that achieves similar performance to SHMEM

Single RMA Window Buffer

- Single buffer used for (almost) all communication
- Messages can be processed concurrently if MPI allows it
- Design is similar to mailboxes within MPI
  - Can process an arbitrary amount of communication

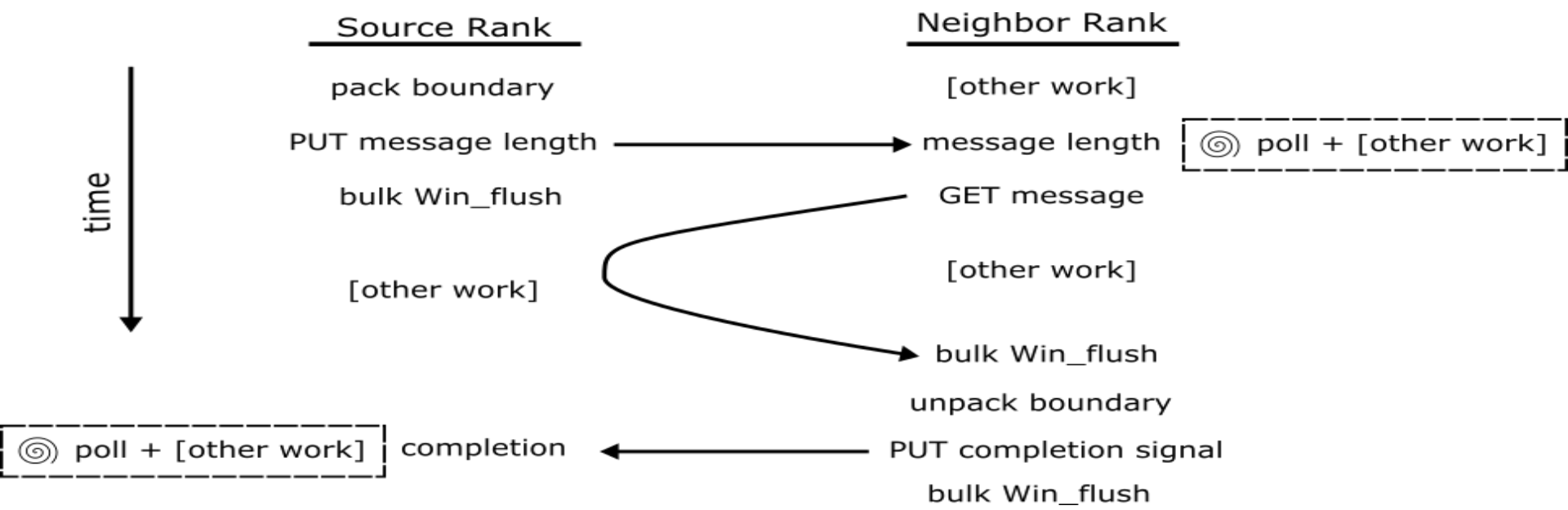


Generalized Update+Comm Engine

- DomainSolver class
- Single class method capable of driving any of the solvers' data exchange and update process
- Used by every solver in Wombat

RMA Boundary Communication Cycle

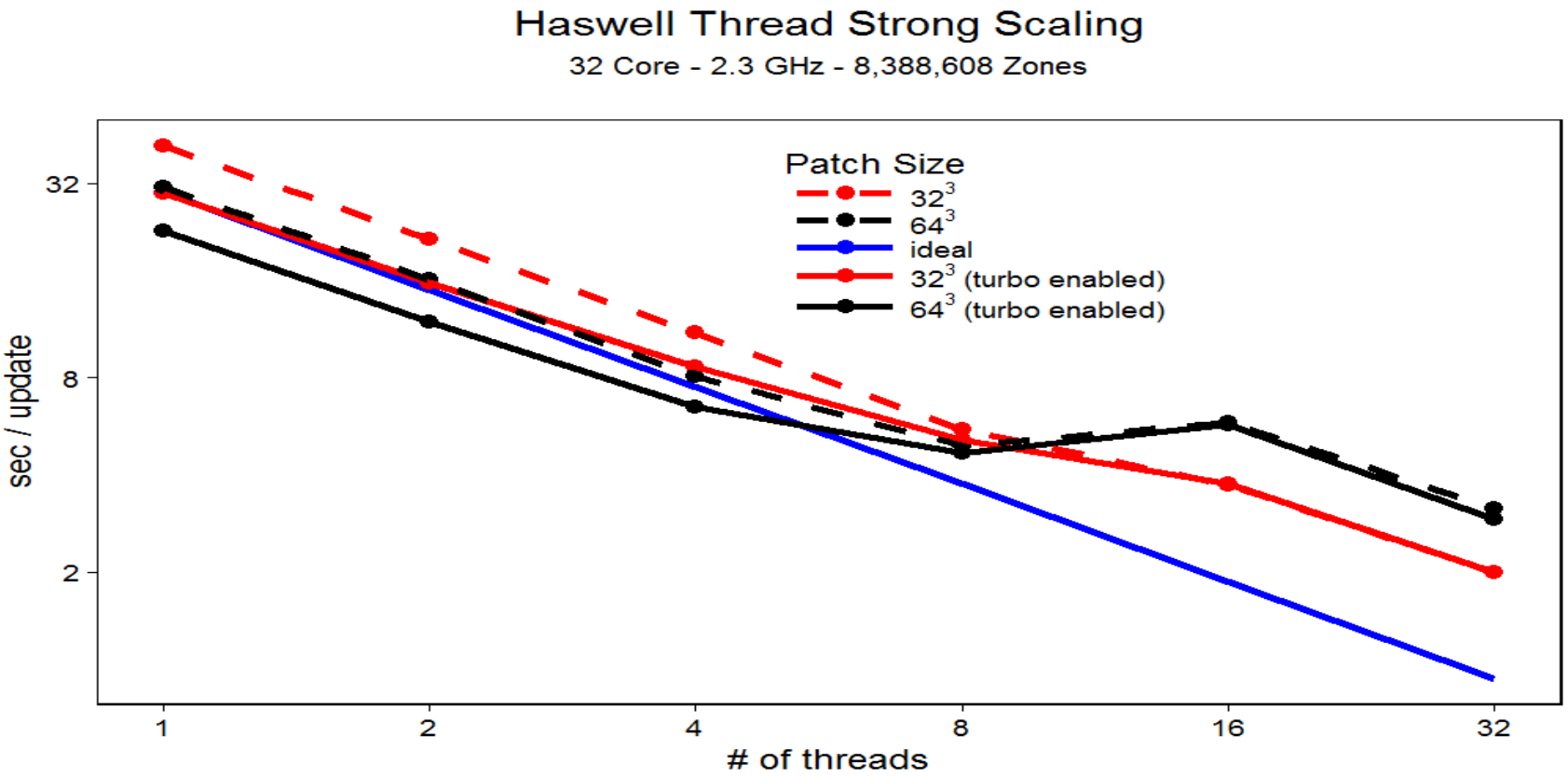
- **Single passive RMA exposure epoch used for the duration of the application**
  - No explicit synchronization between ranks
  - RMA semantics make computation/communication overlap simpler to achieve



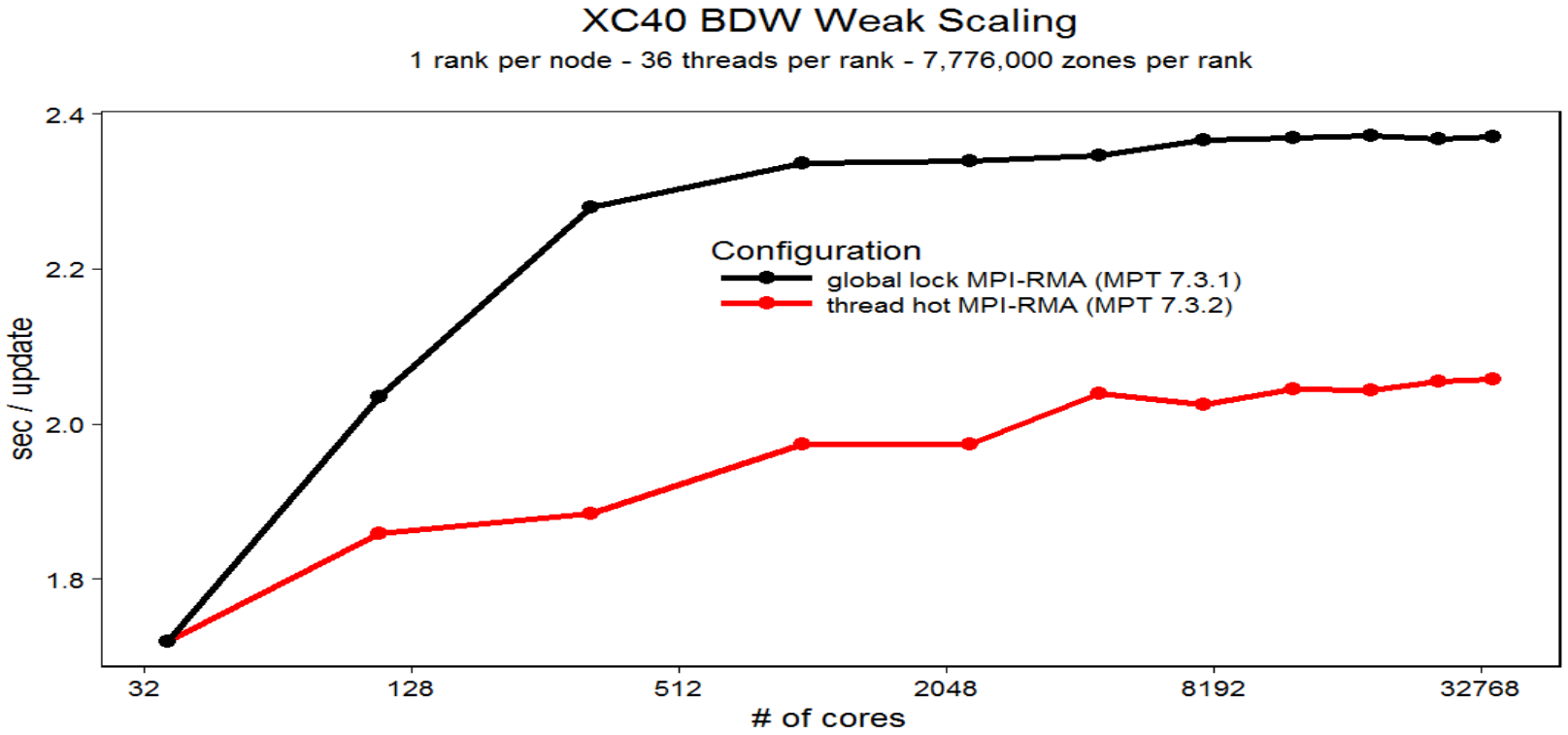
Thread Hot MPI-RMA

- **DMAPP library was enhanced to be “thread hot” for SHMEM**
  - “thread hot” is more than “thread safe”
  - “thread hot” implies concurrency and performance across threads was central to the design
- **MPI-RMA over DMAPP leverages this feature as of MPT 7.3.2**
  - No locks used in DMAPP layer
  - Very light weight locking in MPI layer
  - Design makes it very likely that locks are uncontended
  - Network resources efficiently managed among threads
  - Performance approaches that of N independent processes when using N threads
- **Example on HSW with 16 threads each on 2 nodes**
  - OSU passive MPI_Put bandwidth for 8 B message
  - MPT 7.3.1 = 5.27 MB/s
  - MPT 7.3.2 = 399.9 MB/s
  - 75X improvement

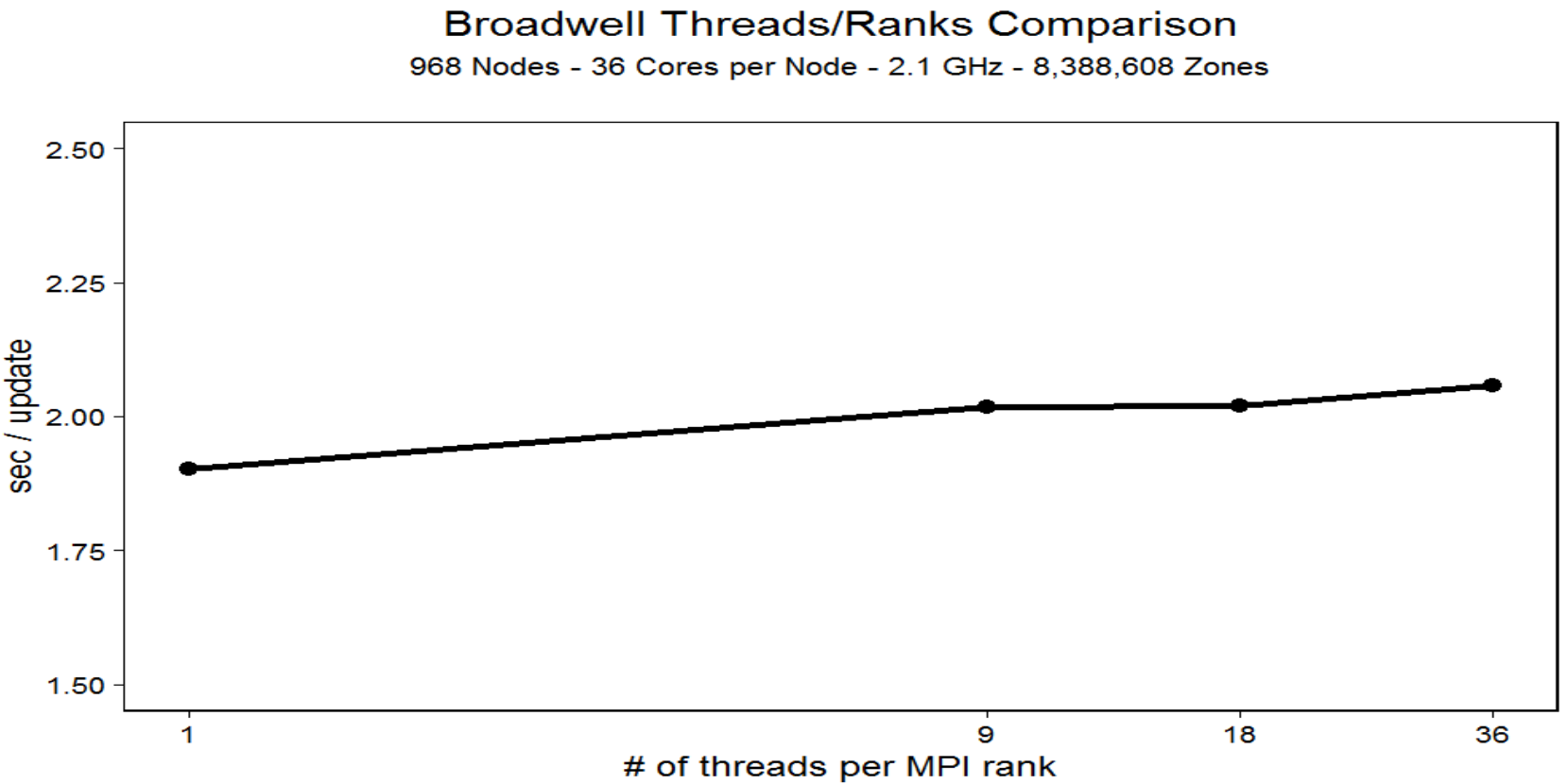
Wombat Performance



- Tunable Patch size very important to performance



- **Rank reordering**
  - Cartesian domain optimization for XC topology/placement improves largest run wall time by additional 2.3%



- Less than 8% difference between 1 and 36 threads per rank
- Ideal for application like Wombat is 0%