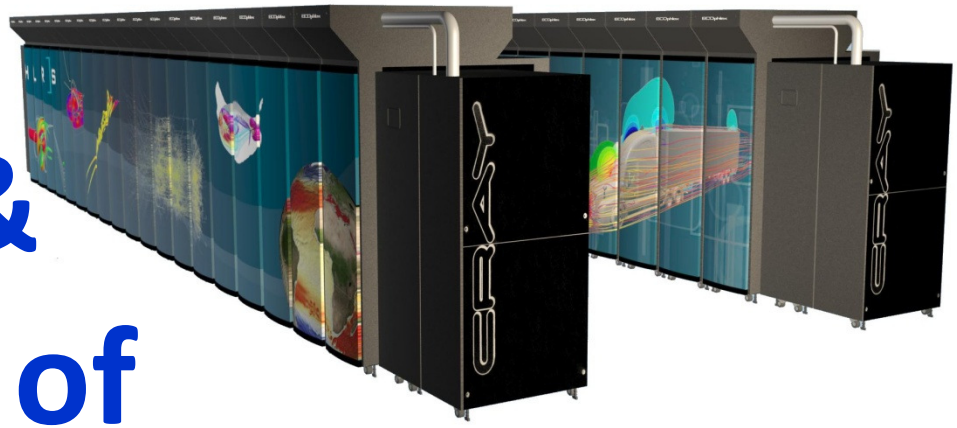
Performance & Measurement of OpenMP Programs



CRAY
THE SUPERCOMPUTER COMPANY

HLRS

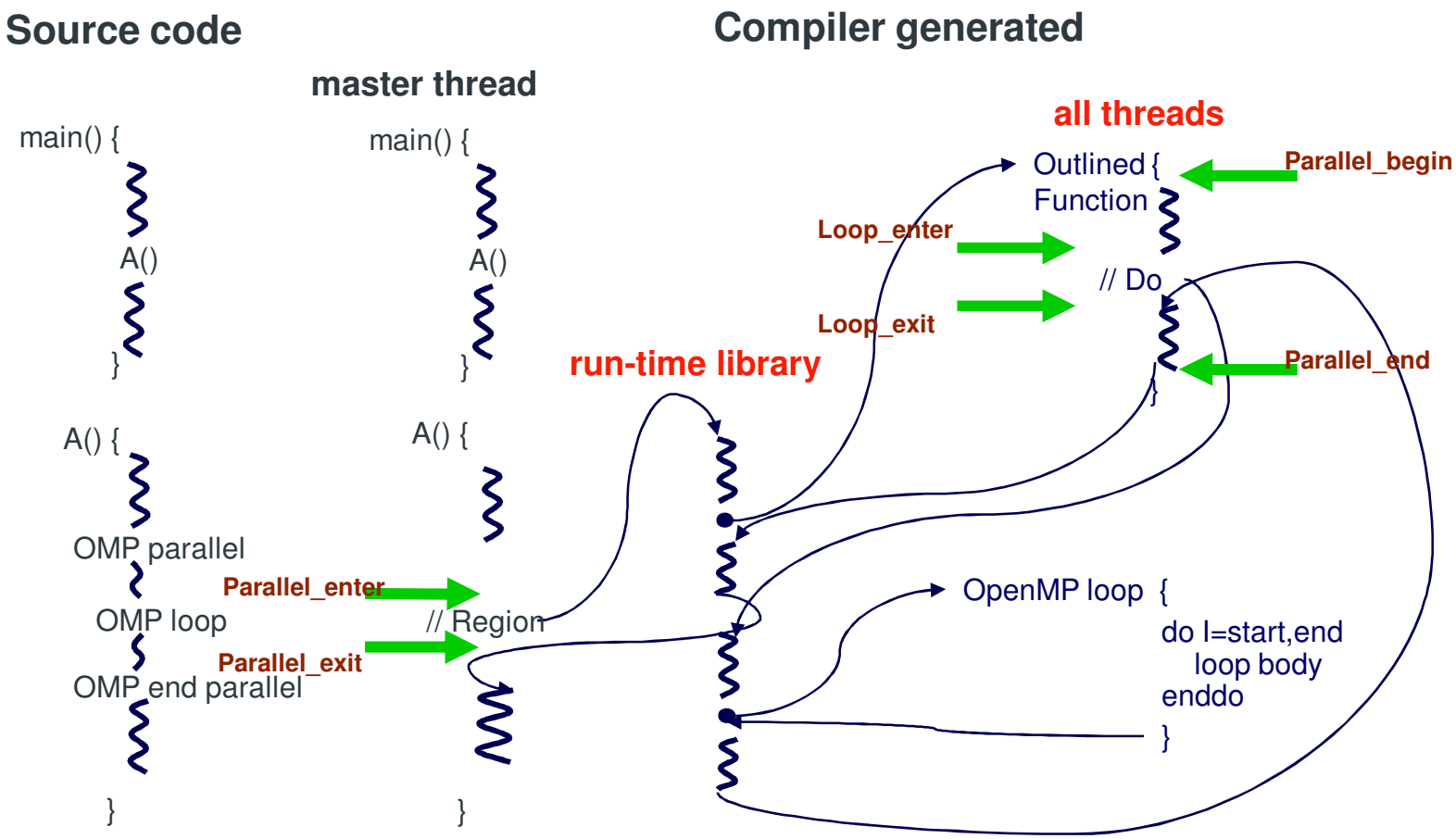
Topics

- What do we want to measure?
- Data collection
- Data reporting

MPI + OpenMP? (some ideas)

- When does it pay to add OpenMP to my MPI code?
  - Add OpenMP when code is network bound
  - Adding OpenMP to memory bound codes may aggravate memory bandwidth issues, but you have more control when optimizing for cache
  - Look at collective time, excluding sync time: this goes up as network becomes a problem
  - Look at point-to-point wait times: if these go up, network may be a problem

OpenMP (Ideal) Instrumentation



OpenMP Data Collection

- Measure overhead incurred entering and leaving
  - Parallel regions
  - Work-sharing constructs within parallel regions
- Trace entry points automatically inserted by Cray and PGI (7.2.0 or later) compilers (by loading “perftools”)
  - Provides per-thread information
    - For CCE : -homp_trace is added when loading perftools
- Can use sampling to get performance data without API (per process view… no per-thread counters)
  - ‚Tracing‘ needed for per-thread counters
    General Linux problem, working on a solution

OpenMP Data Collection (2)

- pat_build -g omp …
  - Specifies tracing of user OpenMP API functions (like omp_test_lock)
- Need to add tracing support for barriers (both implicit and explicit)
  - Need support from compilers
- User API also available for OpenMP trace points when using other compilers

OpenMP Trace Point API

- C API (Same names for Fortran)

```
void PAT_omp_parallel_enter (void);
void PAT_omp_parallel_exit (void);
void PAT_omp_parallel_begin (void);
void PAT_omp_parallel_end (void);
void PAT_omp_loop_enter (void);
void PAT_omp_loop_exit (void);
void PAT_omp_sections_enter (void);
void PAT_omp_sections_exit (void);
void PAT_omp_section_begin (void);
void PAT_omp_section_end (void);
```

- Don’t support combined parallel work sharing constructs
  - Must split apart into parallel construct that contains work sharing constructs

- See pat_help for API function requirements

OpenMP Performance Data Reporting

- Default view (no options needed to pat_report)
  - Focus on where program is spending its time
  - Calculate load imbalance across all threads
    - Options also available to report per MPI rank, per thread
  - OpenMP overhead
    - Remember : This is normally a very small time -> pat_report -T
  - Hardware counter statistics
    - Parallel regions
    - Work-sharing constructs within parallel regions
  - Assumes all requested resources should be used

Imbalance Options for Data Display (pat_report –O …)

- profile_pe.th (default view)
  - Imbalance based on the set of all threads in the program
- profile_pe_th
  - Highlights imbalance across MPI ranks
  - Uses max for thread aggregation to avoid showing under-performers
  - Aggregated thread data merged into MPI rank data
- profile_th_pe
  - For each thread, show imbalance over MPI ranks
  - Example: Load imbalance shown where thread 4 in each MPI rank didn't get much work

Profile by Function Group and Function (with –T)

```
Table 1:  Profile by Function Group and Function

 Time % |      Time |Imb. Time |   Imb. |   Calls |Group
        |           |          | Time % |         | Function
        |           |          |        |         |  PE.Thread='HIDE'

 100.0% | 12.548996 |       -- |     -- |  7944.7 |Total
|-----------------------------------------------------------------
|  97.8% | 12.277316 |       -- |     -- |  3371.8 |USER
||----------------------------------------------------------------
||  35.6% |  4.473536 | 0.072259 |   1.6% |   498.0 |calc3_.LOOP@li.96
||  29.1% |  3.653288 | 0.070551 |   1.9% |   500.0 |calc2_.LOOP@li.74
||  28.3% |  3.545677 | 0.056303 |   1.6% |   500.0 |calc1_.LOOP@li.69
. . .
||================================================================
|   1.2% |  0.155028 |       -- |     -- |  1000.5 |MPI_SYNC
||----------------------------------------------------------------
||   1.2% |  0.154899 | 0.674518 |  82.0% |   999.0 |mpi_barrier_(sync)
||   0.0% |  0.000129 | 0.000489 |  79.8% |     1.5 |mpi_reduce_(sync)
||================================================================
|   0.7% |  0.082943 |       -- |     -- |  3197.2 |MPI
||----------------------------------------------------------------
||   0.4% |  0.047471 | 0.158820 |  77.6% |   999.0 |mpi_barrier_
||   0.1% |  0.015157 | 0.295055 |  95.9% |   297.1 |mpi_waitall_
. . .
||================================================================
|   0.3% |  0.033683 |       -- |     -- |   374.5 |OMP
||----------------------------------------------------------------
||   0.1% |  0.013098 | 0.078620 |  86.4% |   125.0 |calc2_.REGION@li.74(ovhd)
||   0.1% |  0.010298 | 0.052760 |  84.3% |   124.5 |calc3_.REGION@li.96(ovhd)
||   0.1% |  0.010287 | 0.068428 |  87.6% |   125.0 |calc1_.REGION@li.69(ovhd)
||================================================================
|   0.0% |  0.000027 | 0.000128 |  83.0% |     0.8 |PTHREAD
|        |           |          |        |         | pthread_create
|================================================================
```

OpenMP Parallel DOs <function>.<region>@<line> automatically instrumented

OpenMP overhead is normally small and is filtered out on the default report (< 0.5%). When using "–T" the filter is deactivated

Hardware Counters Information at Loop Level

```
========================================================================
USER / calc3_.LOOP@li.96
------------------------------------------------------------------------
  Time%                                              37.3%
  Time                                           6.826587 secs
  Imb.Time                                       0.039858 secs
  Imb.Time%                                          0.6%
  Calls                              72.9 /sec         498.0 calls
  DATA_CACHE_REFILLS:
    L2_MODIFIED:L2_OWNED:
    L2_EXCLUSIVE:L2_SHARED         64.364M/sec     439531950 fills
  DATA_CACHE_REFILLS_FROM_SYSTEM:
    ALL                            10.760M/sec      73477950 fills
  PAPI_L1_DCM                      64.973M/sec     443686857 misses
  PAPI_L1_DCA                     135.699M/sec     926662773 refs
  User time (approx)                6.829 secs   15706256693 cycles  100.0%Time
  Average Time per Call                            0.013708 sec
  CrayPat Overhead : Time            0.0%
  D1 cache hit,miss ratios          52.1% hits         47.9% misses
  D1 cache utilization (misses)      2.09 refs/miss    0.261 avg hits
  D1 cache utilization (refills)     1.81 refs/refill  0.226 avg uses
  D2 cache hit,miss ratio           85.7% hits         14.3% misses
  D1+D2 cache hit,miss ratio        93.1% hits          6.9% misses
  D1+D2 cache utilization           14.58 refs/miss    1.823 avg hits
  System to D1 refill              10.760M/sec      73477950 lines
  System to D1 bandwidth          656.738MB/sec    4702588826 bytes
  D2 to D1 bandwidth             3928.490MB/sec   28130044826 bytes
========================================================================
```

3. Touch your memory, or someone else will.

This isn't as dirty as it sounds.

Memory Allocation: Make it local

- Linux has a “first touch policy” for memory allocation
  - *alloc functions don’t actually allocate your memory
  - Memory gets allocated when “touched”
- Problem: A code can allocate more memory than available
  - Linux assumed “swap space,” we don’t have any
  - Applications won’t fail from over-allocation until the memory is finally touched
- Problem: Memory will be put on the core of the “touching” thread
  - Only a problem if thread 0 allocates all memory for a node
- Solution: Always initialize your memory immediately after allocating it
  - If you over-allocate, it will fail immediately, rather than a strange place in your code
  - If every thread touches its own memory, it will be allocated on the proper socket/die

Using data in spreadsheets

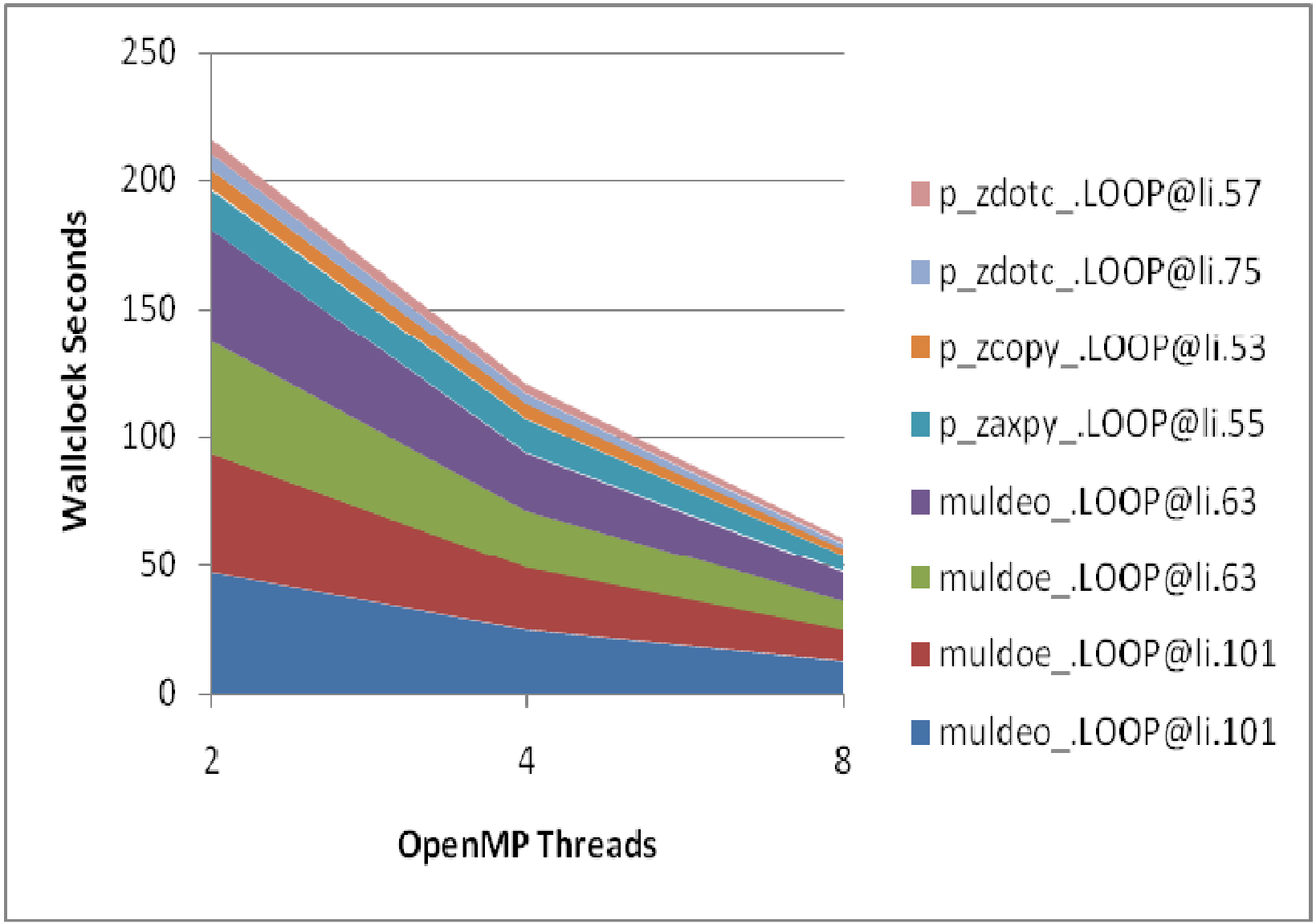
- The full set of pat_report options can be used to customize reports in many ways, but you may prefer spreadsheet analysis, or wish to see the data charted. For that purpose, pat_report can format the data in its tables as comma-separated value lists with the option:
  - -s show_data=csv

- For example, a flat profile in this format can be generated with:
  - pat_report -s show_data=csv -O profile -s show_groups=no ...

Performance Considerations for OpenMP

- Granularity of Computation
  - The parallel region should be as large, in number of operations, as possible
- Load Balancing
  - The work should be distributed evenly across the threads working the OpenMP region

Example : WUPWISE



Major OMP Loop in WUPWISE

```
C$OMP PARALLEL
C$OMP+        PRIVATE (AUX1, AUX2, AUX3),
C$OMP+        PRIVATE (I, IM, IP, J, JM, JP, K, KM, KP, L, LM, LP),
C$OMP+        SHARED (N1, N2, N3, N4, RESULT, U, X)
C$OMP DO
      DO 100 JKL = 0, N2 * N3 * N4 - 1
       L = MOD (JKL / (N2 * N3), N4) + 1
       LP=MOD(L,N4)+1
        K = MOD (JKL / N2, N3) + 1
        KP=MOD(K,N3)+1
         J = MOD (JKL, N2) + 1
         JP=MOD(J,N2)+1
         DO 100 I=(MOD(J+K+L+1,2)+1),N1,2
           IP=MOD(I,N1)+1
           CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
           CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)
           CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
           CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
           CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
           CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
           CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
           CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
           CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
           CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
           CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
           CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
 100  CONTINUE
C$OMP END DO
```

Dependence upon Memory BW Utilization

	2 to 4 Threads	2 to 8 Threads
muldoe_.LOOP@li.101	1.95	3.88
muldoe_.LOOP@li.101	1.94	3.81
muldoe_.LOOP@li.63	1.96	3.90
muldeo_.LOOP@li.63	1.96	3.90
p_zaxpy_.LOOP@li.55	1.2	2.39
p_zcopy_.LOOP@li.53	1.16	2.36
p_zdotc_.LOOP@li.75	1.63	3.10
p_zdotc_.LOOP@li.57	1.52	2.93

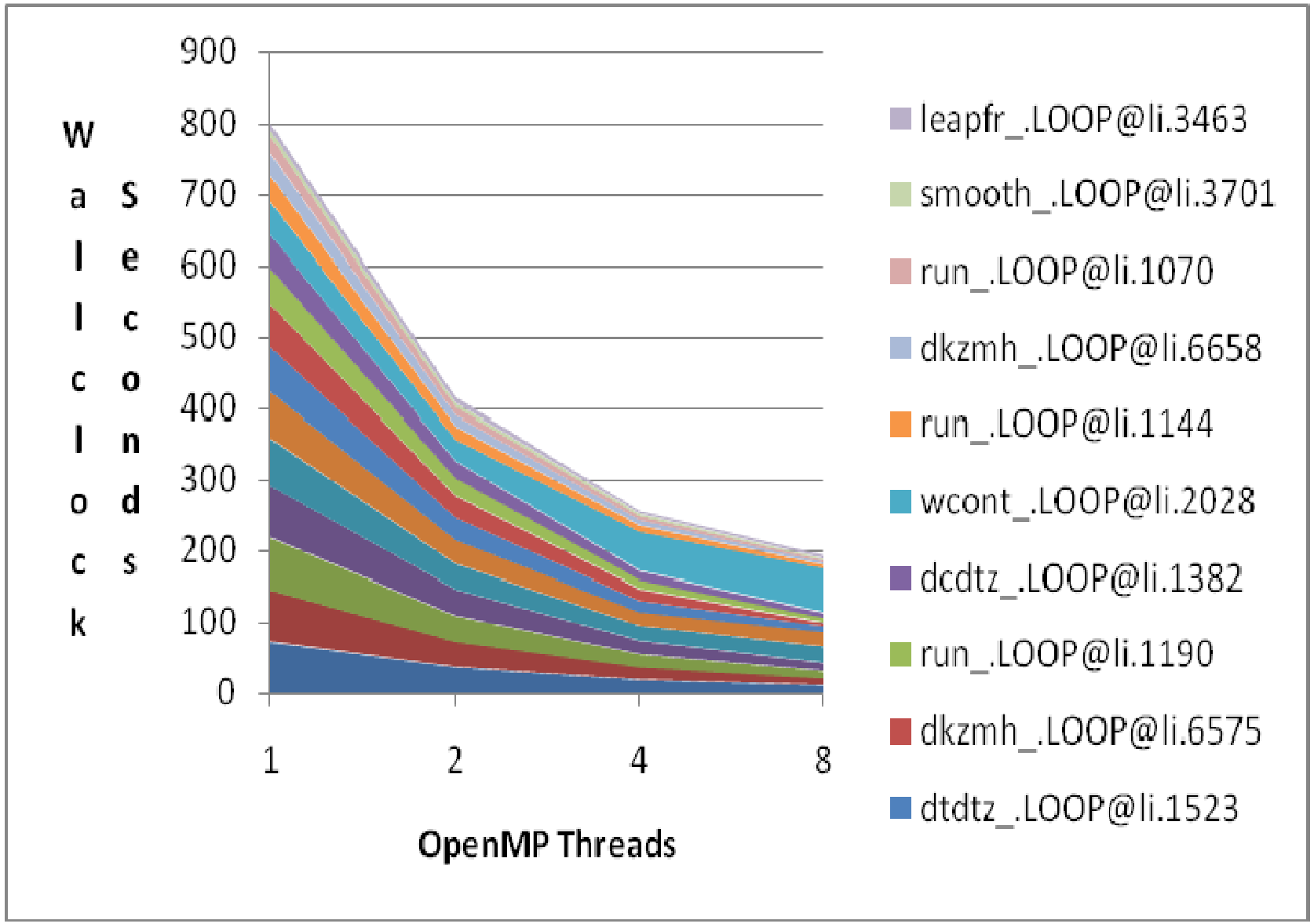
Level 3

Level 2

Performance is Excellent

- Large Percentage of the code that uses the computation time is parallelized
- Granularity of the computation is very large
- Load Balance of the computation is good
- Some of the computation that is parallelized is memory bandwidth limited

APSI


Wall clock
Seconds
900
800
700
600
500
400
300
200
100
0
1
2
4
8
OpenMP Threads
leapfr_.LOOP@li.3463
smooth_.LOOP@li.3701
run_.LOOP@li.1070
dkzmh_.LOOP@li.6658
run_.LOOP@li.1144
wcont_.LOOP@li.2028
dcdtz_.LOOP@li.1382
run_.LOOP@li.1190
dkzmh_.LOOP@li.6575
dtdtz_.LOOP@li.1523

Very Poor Cache utilization – WHY??

```
C
      DO 25 I=1,NUMTHREADS
         WWIND1(I)=0.0
         WSQ1(I)=0.0
 25   CONTINUE

!$OMP PARALLEL
!$OMP+PRIVATE(I,K,DV,TOPOW,HELPA1,HELP1,AN1,BN1,CN1,MY_CPU_ID)
       MY_CPU_ID = OMP_GET_THREAD_NUM() + 1
!$OMP DO
      DO 30 J=1,NY
         DO 40 I=1,NX
            HELP1(1)=0.0D0
            HELP1(NZ)=0.0D0
            DO 10 K=2,NZTOP
               IF(NY.EQ.1) THEN
                  DV=0.0D0
                           ELSE
                  DV=DVDY(I,J,K)
               ENDIF
               HELP1(K)=FILZ(K)*(DUDX(I,J,K)+DV)
 10         CONTINUE
C
C      SOLVE IMPLICITLY FOR THE W FOR EACH VERTICAL LAYER
C
            CALL DWDZ(NZ,ZET,HVAR,HELP1,HELPA1,AN1,BN1,CN1,ITY)
            DO 20 K=2,NZTOP
               TOPOW=UX(I,J,K)*EX(I,J)+VY(I,J,K)*EY(I,J)
               WZ(I,J,K)=HELP1(K)+TOPOW
               WWIND1(MY_CPU_ID)=WWIND1(MY_CPU_ID)+WZ(I,J,K)
               WSQ1(MY_CPU_ID)=WSQ1(MY_CPU_ID)+WZ(I,J,K)**2
 20         CONTINUE
 40      CONTINUE
 30   CONTINUE
!$OMP END DO
!$OMP END PARALLEL
```

```
      DO 35 I=1,NUMTHREADS
         WWIND=WWIND+WWIND1(I)
         WSQ=WSQ+WSQ1(I)
35    CONTINUE
```

CRAY THE SUPERCOMPUTER COMPANY

HLRS

Very Good Cache utilization – WHY??

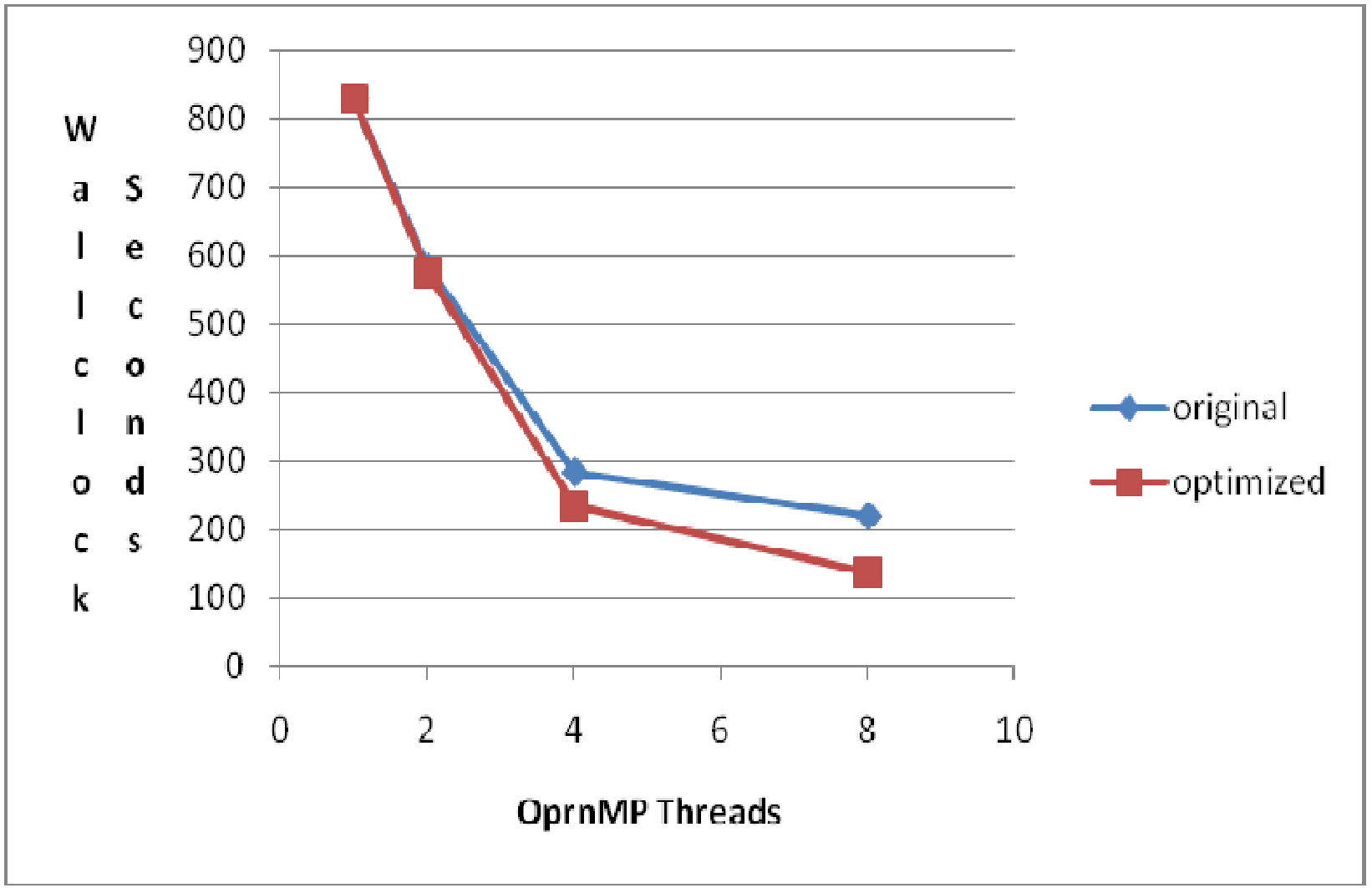
```
C                                                    DIMENSION WWIND1(32,NUMTHREADS), WSQ1(32,NUMTHREADS)
      DO 25 I=1,NUMTHREADS
         WWIND1(32,I)=0.0
         WSQ1(32,I)=0.0
 25   CONTINUE

!$OMP PARALLEL
!$OMP+PRIVATE(I,K,DV,TOPOW,HELPA1,HELP1,AN1,BN1,CN1,MY_CPU_ID)
       MY_CPU_ID = OMP_GET_THREAD_NUM() + 1
!$OMP DO
      DO 30 J=1,NY
         DO 40 I=1,NX
            HELP1(1)=0.0D0
            HELP1(NZ)=0.0D0
            DO 10 K=2,NZTOP
               IF(NY.EQ.1) THEN
                  DV=0.0D0
                           ELSE
                  DV=DVDY(I,J,K)
               ENDIF
               HELP1(K)=FILZ(K)*(DUDX(I,J,K)+DV)
 10         CONTINUE
C
C      SOLVE IMPLICITLY FOR THE W FOR EACH VERTICAL LAYER
C
            CALL DWDZ(NZ,ZET,HVAR,HELP1,HELPA1,AN1,BN1,CN1,ITY)
            DO 20 K=2,NZTOP
               TOPOW=UX(I,J,K)*EX(I,J)+VY(I,J,K)*EY(I,J)
               WZ(I,J,K)=HELP1(K)+TOPOW
               WWIND1(32,MY_CPU_ID)=WWIND1(32,MY_CPU_ID)+WZ(I,J,K)
               WSQ1(32,MY_CPU_ID)=WSQ1(MY_CPU_ID)+WZ(32,I,J,K)**2
 20         CONTINUE
 40       CONTINUE
 30   CONTINUE
!$OMP END DO
!$OMP END PARALLEL
```

```
      DO 35 I=1,NUMTHREADS
         WWIND=WWIND+WWIND1(32,I)
         WSQ=WSQ+WSQ1(32,I)
 35   CONTINUE
```

CRAY THE SUPERCOMPUTER COMPANY

HLRS

Performance Gain from Restructuring



And now for a demo