Dalton cpp-lr parallelization

phase1:
duration: 10 days
Task: profiling and analyzing the code. Figure out the time consuming parts and suggest a parallelization strategy.

The most expensive routines in terms of self time[1]

1. dgemm – 46 %
2. ccsd_symsq – 17 %
3. daxpy – 7 %
4. dcopy – 7 %
5. ccrhs_ipm – 5 %
6. ddot – 3 %

Out of these routines ccsd_symsq & ccrhs_ipm are part of the code.

- ccsd_symsq & ccrhs_ipm can be tried for serial optimization
  - ccsd_symsq – is a very small loop

The other routines are from optimized blas library. The blas routines dgemm, dcopy, ddot etc have parallel versions in the standard library which is very easy to invoke.

- tried parallel mkl blas.
  - no performance improvement for the given benchmark
  - individual calls are small in time
  - may give some benefit for large jobs where each dgemm call takes substantial time
- to make parallelization effective it has to start at a higher level.

The most expensive routines in terms of self + child time

1. cc_trdrv – 84 %
2. cc_cpp_solver2 – 49%
3. cceq_sol – 42 %

Both cc_cpp_solver2 & cceq_sol above call cc_trdrv where they spend most time

- initially thought of parallelizing cc_cpp_solver2
  - within cc_cpp_solver2 it is difficult to make parallel calls to cc_trdrv
  - parallelization will not be effective without parallel calls to cc_trdrv
- we are now thinking on the parallelization of cc_trdrv
  - it will benefit both cc_cpp_solver2 & cceq_sol
  - cc_trdrv has a main loop running over the number of vectors
    - this loop can be parallelized
      - each rank computes one / some vector(s)
    - A more detailed analysis of this routine is given below
  - difficulties in parallelizing cc_trdrv
    - not sure if the vector loop is inherently parallel

1 Self time: without child subroutine call time

- done some test for dependency tracking
  - reversing the order of loop, setting work array to 0 every time etc
    - program works
  - give hints that the vector loop is likely parallel
- discussion with the developers of cc_trdrv will be helpful

- cc_trdrv has many code paths
  - needs clean up
    - created a cleaned cc_trdrv rouine
  - not sure whether calls from cc_cpp_solver2 & cceq_sol follow same path

Analysis of cc_trdrv

Below is the code listing of cc_trdrv when called from cc_cpp_solver2. This is partial source code listing with main blocks for easier readability. The actual loop is much bigger with lots of different code paths.

```
      DO 100 I = 1, NL
        K1 = I + IST - 1
         DO 150 IV = 1, NSIMTR
           KOFF1  = KC1AM + NT1AM(ISYMTR)*(IV - 1)
           CALL CC_RVEC(LUFC1,FC1AM,NT1AM(ISYMTR),NT1AM(ISYMTR),
     *                  IV+K1-1,WORK(KOFF1))
  150      CONTINUE

           CALL CCLR_DIASCL(WORK(KRHO2),TWO,ISYMTR)

           CALL CC_T2SQ(WORK(KRHO2),WORK(KC2AM),ISYMTR)

         IF (.NOT. (CCS.OR.CC2)) THEN
           CALL WOPEN2(LUFSD,FR2SD,64,0)
         ENDIF

         NRHO2 = MAX(NT2AM(ISYMTR),2*NT2ORT(ISYMTR))
         IF (CC2 ) NRHO2 = NT2AM(ISYMTR)

        IVEC = K1
        ITR  = K1

        LRHO1 = NT1AM(ISYMTR)
            CALL CC_RHTR(ECURR,
     *                   FRHO1,LUFR1,FR2SD,LUFSD,FRHO12,LUFR12,
     *                   FC1AM,LUFC1,FC2AM,LUFC2,FC12AM,LUFC12,
     *                   WORK(KRHO1),WORK(KRHO2),
     *                   WORK(KC1AM),WORK(KC2AM),
     *                   WORK(KEND1),LWRK1,NSIMTR,
     *                   IVEC,ITR,LRHO1,.FALSE.,DUMMY,APROXR12)


         DO 90 IV = 1, NSIMTR
           NR1 = IV + K1 - 1

           CALL CC_WVEC(LUFR2,FRHO2,NT2AM(ISYMTR),
     *                    NT2AM(ISYMTR),NR1,WORK(KRHO2))

  90       CONTINUE

  999     CONTINUE    ! From Cholesky section
        IF ( .NOT.(CCS.OR.CC2)) THEN
          CALL WCLOSE2(LUFSD,FR2SD,'DELETE')
        ENDIF


  100  CONTINUE
```

In the above “DO 100 I = 1, NL” is the loop over number of vectors. In this loop three routines are important

1. CC_RVEC – reads vector
2. CC_WVEC – writes vector
3. CC_RHTR – transformation

parallelization strategy

- split the loop NL
  - can it run completely in parallel?
    - If not where is the dependency
    - as there are many arrays involved it is not easy to figure out
  - which arrays are returned, which arrays are scratch
- MPI may be better for long term
- OpenMP seems easier to implement

Other Observations

- The CC_RVEC / CC_WVEC read/write data from/to ascii format
  - slow (although for the benchmark data % time w.r.t total runtime is small)
  - data in file has less floating point precision than data in memory
- Can be replaced by binary data read write
  - fast
  - same floating point precision as the data in memory

Conclusion

- It seems that parallelization of cc_trdrv will be most beneficial
  - cc_trdrv has a clear parallelizable block
- OpenMP seems easier to implement, will also satisfy typical job requirement
- A discussion with the developers of the cc_trdrv is crucial
- Actual work can be carried out in phase2