# High-Performance Fortran as a Possible Successor to Global Arrays in the NWChem Parallel Computational Chemistry Code*


Yuhong Wen, D. Bryan Carpenter,
Erol Akarsu, Tomasz Haupt, and David E. Bernholdt†

Northeast Parallel Architectures Center
Syracuse University
111 College Place
Syracuse, NY 13422–4100


July 2, 1997


**Abstract**

This is the abstract


## 1 Introduction

When the development of the NWChem parallel computational chemistry package began at the Pacific Northwest National Laboratory (PNNL) in 1993, the question of what parallel programming environment to use received careful attention. It was clear that a straightforward message-passing model would place a significant burden on the programmer. High-Performance Fortran (HPF) provided a higher-level programming model, but suffered from


*Prepared for Battelle/Pacific Northwest National Laboratory under contract 315647-A9E.



†Author for correspondence. E-mail address *bernhold@npac.syr.edu*.

several other problems. In the first place, version 1.0 of the HPF standard addressed only data-parallel operations, while it was clear that both data- and task-parallel capabilities would be required for the algorithms envisioned in NWChem. Second, at that time, the standard had not yet been formally approved, and such HPF implementations as did exist were based on drafts of the standard, were incomplete in their implementation of the language, and were rather immature.

As a result, the NWChem developers decided to implement their own parallel programming model, *Global Arrays*, which would be modeled to a large extent on HPF. The hope was that a future version of HPF would support the capabilities required by NWChem, and that given time, compilers would become more complete and robust, making it possible to transition NWChem from the global array toolkit (GA) to HPF, thereby acruing the benefits of an accepted standard and vendor-supported compilers.

Since the HPF 1.0 standard has now been available for several years, and since version 2.0 of the language standard (currently in a public comment period prior to formal acceptance) includes some task parallelism and other features required by NWChem algorithms, it is appropriate at this time to evaluate the present state of HPF and the prospects for using it to replace the GA toolkit in NWChem and other similar computational chemistry applications.

There are two components to this analysis. The first is an examination of current HPF implementations, and their performance in comparison with GAs. The second is an examination of whether or not the HPF 2.0 standard supports the task parallel and other capabilities required by the algorithms inplemented in NWChem.

# 2 Performance of HPF and GAs

# 3 HPF 2.0 Capabilities Analysis

Version 2.0 of the High-Performance Fortran standard is currently in a public comment period prior to formal acceptance. This version of HPF incorporates some support for task-based parallelism as well as other features similar to those in the GA toolkit. In order to understand the degree to which HPF 2.0 supports the GA programming model, we have looked at the requirements of two algorithms implemented in NWChem which are exemplars of the kind of task parallelism used throughout the code.

We must note that since there are no HPF 2.0 compilers presently available, the proposed implementations we outline are "thought experiments" and subject to all of the uncertainties of both interpretation of the standard itself, and actual implementations.

## 3.1 HPF 2.0 vs HPF 1.0

The HPF 2.0 standard is divided into two parts: a core language definition and a set of "approved extensions"[1]. The core of HPF 2.0 is a conservative extension of HPF 1.1. The only major addition is "reduction variables", which will be important for the discussion below. The approved extensions are more innovative. They include new irregular data distribution formats, options for distributing data over processor subsets, explicit support for "ghost areas", support for a form of task parallelism, and new library and intrinsic procedures. We will need very few of these new features.

The GAs code discussed below is task parallel rather than data parallel. We initially hoped that the HPF 2.0 extensions for task parallelism would be useful, but it seems that the fairly complex extensions in section 9 of the standard are less relevant than expected. There is a strong emphasis there on restricting occurrence of "one-sided communication", whereas we find controlled use of one-sided communication very convenient. A desirable task-parallel feature—some mechanism for creating a load balanced task farm—is notably absent from the standard. On the other hand we have found that (load-balancing aside) the simple `INDEPENDENT` do loop, augmented with `REDUCTION` clauses, is enough to express most of the task parallelism we need.

A simple example of an `INDEPENDENT` do, taken from the standard:

```
!HPF$ INDEPENDENT, NEW(j), REDUCTION(x)
      DO i = 1, 10
!HPF$   INDEPENDENT
        DO j = 1, 20
          x = x + j
        END DO
      END DO
```

The `INDEPENDENT` directives assert that successive iterations of the loops do not interfere with one another, and may therefore be executed in parallel.

[1]There was a similar but differently-stated division of HPF 1.0 between "Subset HPF" and "Full HPF".

The NEW clause asserts that each instance of the i loop body can use an unrelated copy of the variable j, never relying on a value set in a previous iteration. The REDUCTION clause asserts that the variable x is only used in the very special context of an accumulating assignment with a commutative associative operation, as illustrated. The standard explicitly allows the reduction variable to be an array, so long as the subscripting pattern is identical on the right- and left-hand-side of the accumulating assignment. The language places no special restriction on procedure calls inside the body of an INDEPENDENT loop, so long as they respect the disjointness criterion.

We also considered the possibility of using a FORALL construct to achieve parallelism. A limited form of task parallelism can be achieved by calling a PURE function in the forall assignment. But the restrictions on PURE functions are too severe for our purposes.

The GAs versions of the codes use distributed arrays with two-dimensional block distribution format. This format is supported in the HPF approved extensions where it is called GEN_BLOCK distribution. A simple example is

```
      REAL b(100)
!HPF$ DYNAMIC b
      INTEGER blks(4)
      ...
      blks = (/50, 10, 20, 20/)
!HPF$ REDISTRIBUTE b(GEN_BLOCK(blks))
```

The dynamic, REDISTRIBUTE, form of distribution directive was used here because the vector of block sizes is not a compile-time constant (this is presumably typical).

## 3.2 NWChem Distributed Data SCF Algorithm

As a concrete focus (following [?]) we will consider in detail computation of the Fock matrix in a self-consistent field (SCF) method. SCF is an important tool for electronic structure calculations, and construction of the Fock matrix is a computationally dominant part of the method. The basic problem is simply stated, while essential practical optimizations depend on features of the GA toolkit (like task parallelism and irregular data distribution) that are not well supported by HPF 1.0.

The Fock matrix describes the force field experienced by the electrons in a molecule. It is defined as

$$F_{ij} = h_{ij} + \sum_{k=1}^{N}\sum_{l=1}^{N} D_{kl}\left((ij|kl) - \frac{1}{2}(ik|jl)\right) \tag{1}$$

Here $i$ and $j$ run from 1 to $N$, where $N$ is the number of basis functions. Typically a few (say, 10) basis functions are introduced for each atom in a molecule. The choice of basis functions will vary from atom to atom (a hydrogen atom will need fewer basis functions than a more complex atom like oxygen). The fixed matrix $h$ takes account of the background field produced by the static atomic nucleii. The main work is in computing the sum over $k$ and $l$, which describes the field produced by the electron cloud itself. The density matrix $D$ is some estimate of the of the distribution of electrons. Inserting this into equation 1 gives a corresponding estimate of the Fock matrix. Diagonalizing the Fock matrix yields a new estimate of the electron wave function, and in turn a new value for the density $D$. The process is repeated until convergence (self-consistency) is achieved.

The symbols $(ij|kl)$, $(ik|jl)$ represent integrals accounting for Coulomb and exchange coupling between basis states. Typically the dominant part of the computation is evaluation of these integrals[2]. In the dense case where most integrals and elements are non-zero, diagonalization is $O(N^3)$ while computation of the matrix according to equation 1 is $O(N^4)$. In this report, therefore, we concentrate evaluation of equation 1 itself.

The $(ij|kl)$ symbols have various symmetries:

$$\begin{aligned}(ij|kl) &= (ji|kl) = (ij|lk) = (ji|lk) = (kl|ij) \\ &= (kl|ji) = (lk|ij) = (lk|ji)\end{aligned} \tag{2}$$

These symmetries can (and, for efficiency, must) be used to reduce the range of the loops used to compute the sum above. Moreover many basis states have very small overlap, and the corresponding integrals can be neglected ("screening"), further reducing the number of terms in the sum. For a more complete discussion of these issues see [?]. Here we only note that for these reasons that bounds on nested loops are often described by non-constant expressions, and loop bodies often contain non-trivial tests to check for screening and symmetry-induced redundancy.

[2] These are sequential "tasks" and most of the details of their computation are irrelevant to the discussion here.

## 3.3 The NWChem SCF code

For reference, figures 2 through 6 reproduce code fragments from the current NWChem release. Obviously we have only selected the subroutines most relevant to the discussion here, and within these procedures many lines of code that are peripheral to the discussion are ellided.

The code given in the figures incorporates one change to the original NWChem code, anticipating the needs of an HPF version. The procedure `fock_2e_cache_dens_fock` is important because it controls access to distributed arrays. Specifically, it copies values from the global arrays representing the density matrix to local temporaries. The copy operation is conditional on the subscripts from the previous iteration. If the values for the current iteration are already "in cache" (ie, in the local temporaries) the copy is not repeated. In the original version of the code `fock_2e_cache_dens_fock` also handled accumulation of results into the global array representing the Fock matrix, adopting a strategy which deferred the write-back until the new subscripts differ from the previous iteration. This saves some global operations and is presumably a useful optimization, but we will see that it is slightly awkward in the HPF translation, where more straightforward patterns of access to reduction variables are preferred (to simplify the discussion, at least). The conditional update operations have been moved into `fock_2e_task`. Changes to the original code are highlighted with asterisks.

Each of the subroutines displayed here raise issues for an HPF port, and we will discuss them in turn.

### 3.3.1 Subroutine `uhf_energy`

This procedure, outlined in figure 2, is responsible for creating the distributed arrays holding the density matrix and the Fock matrix. We see that in practice a handful of identically shaped arrays are used to hold separate components of each. The calls to `ga_create_atom_blocked` return handles to global arrays.

The NWChem package is implemented in Fortran 77, and follows traditional Fortran practises for dealing with dynamically allocated objects and arrays. In particular a global array is proxied by an integer handle. In transitioning to HPF this will certainly change. A global array will become a first-class Fortran array. This introduces a superficial problem in how to represent vectors of related arrays, like `d` and `f`. Fortran 90 provides several possible solutions. The most obvious is to replace `d` and `f` with three-dimensional arrays (instead of vectors of two dimensional arrays). Another solution is to replace them with vectors of objects[3] containing pointers to two-dimensional arrays. A third possibility would be to change the type of the *element* of the two-dimensional array to a derived type with several components. For simplicity, we assume here that `d` and `f` are replaced with three-dimensional arrays.

In the function `ga_create_atom_blocked` (figure 2) a pair of *map* vectors are created and an irregularly distributed global array of double precision variables is created. HPF 1.0 did not provide facilities for irregular distribution of arrays, but the *approved extensions* to HPF 2.0 include a new distribution format called `GEN_BLOCK`. A minor difference to Global Arrays is that the HPF general block distribution is parametrized by a vector of block sizes rather than the vector of starting subscripts for blocks. The computation of *map* vectors can easily be adapted to compute *block_size* vectors instead.

Schematically, the body of `uhf_energy` will be replaced with:

```
      ...
      double precision, allocatable :: d(:, :, :), f(:, :, :)
!hpf$ dynamic :: g_array
      ...
      integer block_size1(max_nproc), block_size2(max_nproc)
c
c     Arrays for AO density, coulomb and exchange matrices
c
      ... compute 'nbf', 'nblock1', 'block_size1', 'nblock2',
          'block_size2' for set of atoms and basis functions ...

      allocate d(nbf, nbf, 4), f(nbf, nbf, 4)
!hpf$ redistribute d(gen_block(block_size1(1 : nblock1)),
     $               gen_block(block_size2(1 : nblock2)), *)
!hpf$ redistribute f(gen_block(block_size1(1 : nblock1)),
     $               gen_block(block_size2(1 : nblock2)), *)

c
c     Make the densities and build the fock matrices
c
      ... initialization
```

[3] By which we mean entities of derived type: Fortran does not directly permit arrays of pointers.

```
      ...
      call fock_2e(geom, basis, 4, jfac, kfac, tol2e,
     $             oskel, d, f)
      ...
      deallocate d, f
      ...
```

Because the *block_size* vectors are not compile-time constants it is necessary to use a `REDISTRIBUTE` directive after the arrays are allocated. The argument of the `GEN_BLOCK` format is a vector of size equal to the corresponding extent of the processor arrangement, the *nblock* values in this case.

An explicit two dimensional processor arrangement could be introduced to help the compiler:

```
!hpf$ processors p(nblock1, nblock2)
```

changing the `REDISTRIBUTE` directives to

```
!hpf$ redistribute d(...) onto p
```

This may require that the *nblock* values be compile-time constants or restricted expressions: some reorganization of the mapping computation may be needed to take acount of this.

Note that `d` and `f` are usually called `vg_fock` and `vg_dens` in the rest of this section.

### 3.3.2 Subroutine `ao_fock_2e`

After creating the arrays, `uhf_energy` calls `fock_2e`. This is a simple wrapper for `ao_fock_2e`, represented in figure 3.

The main role of `ao_fock_2e` is to determine an appropriate task chunking (through an array that will later be called `blocks`) prior to calling the routine that dispatches the tasks, and to allocate a large number of local temporary arrays, mostly used to cache values from the main global arrays in the course of the computation. The allocation is done using stack management routines provided by the MA package from Global Arrays.

In Fortran 90 it is much more natural to create such temporaries as *automatic* arrays. In an HPF version this will be almost mandatory. These local arrays should really be defined in the routine that dispatches the parallel loops (because, we will see, they should appear in the `NEW` list of those loops). So the body of `ao_fock_2e` may be stripped down to something like

```
      ...
      integer, allocatable :: blocks(:)
c
c     Determine appropriate task chunking and max no. of bf in a
c     block of the density/fock matrix
c
      ...
      allocate blocks(2 * natoms)
      call fock_2e_block_atoms(basis, oskel, tol2e,
     $     blocks, nblock, maxblock)
      ...
         call fock_2e_a( geom, basis, nfock, ablklen,
     $        jfac, kfac, tol2e, oskel,
     $        vg_dens, vg_fock,
     $        blocks, nblock)
      ...
c
      deallocate blocks
      ...
```

Allocation of the `dij`, `fij`, etc temporaries is now left to `fock_2e_a`.

### 3.3.3 Subroutine `fock_2e_a`

This subroutine is represented in figure 4. It contains the main parallel loop of the algorithm.

In the global arrays version of the program all blocks of atoms are enumerated by every processor, and each block is allocated to a particular processor using a shared task counter accessed through the function `nxtask`. HPF provides no explicit mechanism for this kind of dynamic load balancing. The best we can achieve is to turn the loops into `INDEPENDENT` do loops, and hope that the resulting tasks are reasonably balanced[4].

The tasks are still not disjoint because they share write-access to the Fock array. The shared access is purely through commutative add-accumulate operations, so the program is deterministic. HPF 2.0 allows this kind of access through *reduction variables*. The Fock matrix must appear in a `REDUCTION` clause of the `INDEPENDENT` directive. References to reduction variables are very restricted in HPF 2.0. In particular reduction variables cannot be passed to procedures[5]. So `vg_fock` must be removed from the argument list of

[4]Or hope that the compiler somehow balances tasks by intelligent scheduling.

[5]There is not an explicit statement to this effect in the language definition, but proce-

fock_2e_task, and updates of this array must appear inline in the body of the fock_2e_a main loop.

At the start of section 3.3 we explained that the code in figures 2 through 6 is slightly modified from the original NWChem code. With a view to simplifying access to reduction variables, all updates of vg_fock have been collected together at the end of fock_2e_task, in six calls to fock_upd_blk. We now want to move the updates out of fock_2e_task altogether, pushing them into the calling program. (Given that requirement, it is also natural to move the computation of i_lo, i_hi, etc from the start of fock_2e_task into the calling program, because these values are needed in the update.) Of course that isn't quite the end of the story, because we cannot pass vg_fock to fock_upd_blk either. The latter call has to be replaced by an inline assignment to the reduction variable vg_fock.

Essentially the call

```
call fock_upd_blk(nfock, vfock, ilo, ihi, jlo, jhi, fac, buf, tmp)
```

is an array section assignment of the form

```
jlen = jhi - jlo + 1
ilen = ihi - ilo + 1
do ii=1,nfock
  vfock(ilo : ihi, jlo : jhi, ii) =
      vfock(ilo : ihi, jlo : jhi, ii) +
          fac(ii) * buf(1 : ilen, 1 : jlen, ii)
enddo
```

To make this work, buf has been changed to a three dimensional array, matching the rank of vfock.

Some corresponding changes are needed for the accesses to the density matrix vg_dens. As mentioned earlier, this array is accessed in fock_2e_cache_dens_fock which is called from fock_2e_task. The references to the density matrix global arrays are intrinsically non-interfering, and HPF does not forbid *non-interfering* accesses from within a procedure called inside an "independent" loop. Unfortunately the technique of saving the value of the local temporaries between iterations if the subscripts have not changed *is* interfering, and has to be abandoned. The array ijk_prev will be eliminated, and the dij, etc

dures calls are clearly not "special locations in assignment statements of a special form", which, according to the standard, is the only place inside the loop where a reduction variable may be referenced.

local temporaries will be declared `NEW`. The actual access to the global array can stay inside the task procedure.

Incorporating the changes discussed above, the body of `fock_2e_a` is sketched in figure 1. The local temporaries `dij`, `dik`, `flj`, `flk` are now declared as three-dimensional *automatic* arrays. They do not appear in the argument list of `fock_2e_a`. They *do* appear in the `NEW` clause of the inner parallel loop.

### 3.3.4 Subroutine `fock_2e_task`

Finally we discuss anticipated changes to `fock_2e_task`. The main changes are

- Delete computation of `i_lo`, `i_hi`, etc calls to `fock_upd_blk`, because the updates have moved into the calling program.
- Change all local temporaries to three-dimensional arrays.
- Simplify `fock_2e_cache_dens_fock`. `fock_get_blk` calls are replaced with array section assignments, this time copying from the global array to the local temporary. All these are operations are now unconditional: there is no test on the value of `ijk_prev`.

The second change will carry through to the inner routine `fock_2e_b`. Although these changes to the rank of the array are "trivial", they may well constitute the largest single set of changes to the code as a whole.

Access to the global arrays through `fock_2e_cache_dens_fock` has been troublesome in this translation to HPF 2.0 proposed above. We were already forced to take accesses to the `Fock` matrix out of this procedure and put them in the main parallel loop. It would now probably be more natural to handle accesses to the global *density matrix* array in the same way. In fact, once all global array accesses are in the main loop we anticipate that many of them can be pulled into the outer loops, restoring the optimizations that were lost by deleting the `ijk_prev` mechanism.

## 3.4 NWChem MP2 Back-Transformation Algorithm

We investigated the Global Array codes of NWChem MP2 Back-Transformation Algorithm and found that this algorithm can also be rewritten in HPF 2.0. It

```
      double precision dij(maxblock, maxblock, nfock)
      double precision dik(maxblock, maxblock, nfock)
      ... similarly declare `dli', `djk', ..., `fij', `fik', ..., `flk' ...

      ...
!hpf$ independent, new(jb, iatlo, iathi), reduction(vg_fock)
      do ib = nblock, 1, -1
         iatlo = blocks(1,ib)
         iathi = blocks(2,ib)
         ...
!hpf$    independent, new(kb, jatlo, jathi, ...)
         do jb = 1, ib
            jatlo = blocks(1,jb)
            jathi = blocks(2,jb)
            if (... some ij pair survives screening ...) then
!hpf$          independent, new(lb, katlo, kathi, lbhi)
               do kb = ib, 1, -1
                  katlo = blocks(1,kb)
                  kathi = blocks(2,kb)
                  lbhi = kb
                  if (ib .eq. kb) lbhi = jb
!hpf$             independent, new(latlo, lathi, otest,
!hpf$$                             dij, dik, dli, djk, dlj, dlk,
!hpf$$                             fij, fik, fli, fjk, flj, flk)
                  do lb = 1, lbhi
                     latlo = blocks(1,lb)
                     lathi = blocks(2,lb)
                     ... set `otest' if some kl pair survives screening and
                         interaction is ``symmetry-unique'' ...
                     if (otest) then
                        ... compute ilo, ihi, jlo, ..., llo, lhi for block ...

                        call fock_2e_task(
     $                       geom, basis, oskel,
     $                       iatlo, jatlo, katlo, latlo,
     $                       iathi, jathi, kathi, lathi,
     $                       ilo, ihi, jlo, jhi, klo, khi, llo, lhi
     $                       nfock, vg_dens,
     $                       jfac, kfac,
     $                       dij, dik, dli, djk, dlj, dlk,
     $                       fij, fik, fli, fjk, flj, flk,
     $                       tmp, tol2e, dentol)

                      jlen = jhi - jlo + 1
                      ilen = ihi - ilo + 1
                      ...
                      do ii=1,nfock
                        vg_fock(ilo : ihi, jlo : jhi, ii) =
                            vg_fock(ilo : ihi, jlo : jhi, ii) +
                                jfac(ii) * fij(1 : ilen, 1 : jlen, ii)
                        vg_fock(ilo : ihi, klo : khi, ii) =
                            vg_fock(ilo : ihi, klo : khi, ii) +
                                kfac(ii) * fik(1 : ilen, 1 : klen, ii)
                        ... simililary add `fjk', `fli', `flj', `flk'
                      enddo
                     end if
                  end do
               enddo
            endif
         end do
      end do

      ...
```

Figure 1: HPF version of body of `fock_2e_a`.

will be easier than the DDSCF algorithm, due to the simple calling structure in the kernel loop. We can just use FORALL statemants and constructs to implement the parallel tasks in this algorithm, instead of using the INDEPENDENT directive.

The main things we need to focus on when rewriting the codes will again be in the two areas of dynamic array allocation and distribution, and the implementation of parallel tasks. First, dynamic array allcation. We need to replace the function call to ga_create_irreg in subroutine mp2_nonsep_uhf in file mp2_back_transform.F. We can use the same strategy used in the DDSCF codes to deal with this irregular distribution, and the dynamic array allocation problem.

After setting up the parallel tasks in the mp2_backt_info function call and the dynamic array allocation, the remaining thing is how to implement them in parallel way in HPF. We need to rewrite the function mp2_back_transform_uhf in file mp2_back_transform.F in the HPF codes. The main point we need to focus on is to rewrite the following Global Array functions as HPF codes with FORALL statements.

- The call

  ```
        call ga_fill_patch(g_buf, 1, nva, 1, nbfpair, 99.0d0)
  ```

  will be changed to:

  ```
        FORALL (i=1:nva, j=1:nbfpair) ga_buf(i,j) = 99.0d0
  ```

- The call

  ```
  ga_dgemm(transa, transb, m, n, k, alpha, g_a, g_b, beta, g_c )
  ```

  can be replaced by:

  ```
  FORALL (i=1:k, j=1:k) d(i,j) = alpha * OPA(1,j) * OPB(i,j)
  FORALL (i=1:m, j=1:n) c(i,j) = d(i,j) + beta * C(i,j)
  ```

  where OPA = A or TRANSPOSE(A), OPB = B or TRANSPOSE(B).

- The Global Array function

  ```
              call ga_put(g_buf, a-nva_lo+1, a-nva_lo+1,
       $           1, nbfpair, tmp, 1)
  ```

to update the global result array after the computing can be replaced with:

```
dimension g_buf(1:dim1,1:dim2)
dimension tmp(1:1, 1:*)

g_a(a-nva_lo+1:a-nva_lo+1, 1:nbfpair) = buf(1:1, 1:nbfpair)
```

# 4 Conclusions and Recommendations

Although we have by no means completed the conversion of even the SCF kernel to HPF, our analysis suggests that expressing task parallel algorithms like these in HPF may be less difficult than is often assumed. In addition its strictly data parallel constructs (array syntax and `FORALL`) HPF 2.0 provides a general parallel loop in the form of its `INDEPENDENT` do loop. Newly enhanced with reduction variables, this construct appears sufficiently powerful to implement the algorithms considered here.

There are a few caveats.

- We had to make a few reorganizations to the logic of the original code to fit it into constraints of the language. These reorganizations sometimes involved undoing optimizations in the original code. At the present time it is impossible to determine whether these changes have a significant impact on performance.

- Probably the most serious of these reorganizations was the replacement of a load-balancing, task-farming loop with a deterministic parallel loop. We regard the lack of a simple mechanism for dynamic load balancing as a worrying shortcoming of the language. Perhaps a compiler can automatically adopt a load balancing strategy for independent loops, but the practicality of this is unclear to us.

- Besides changes to the logic, the data layout has to be modified extensively. This isn't simply a matter of adding directives to specify distribution format. The HPF code probably will not work well unless the ad hoc Fortran 77 style of memory management (relying on sequence and storage associations) is replaced globally with consistent, type-secure, Fortran-90-style handling of arrays.

- We are unaware of any existing HPF compiler that supports the HPF 2.0 extensions used here.
- *New features aside, we question whether existing HPF compilers will generate good code from* `INDEPENDENT` *loops as complex as the ones considered here.*

HPF compilers often concentrate on parallelizing simple loops, where communication can be reduced to cooperative or collective operations. On encountering a complex loop where communication cannot sensibly be reduced to such simple patterns, a compiler may well fall back on a default strategy of sequentializing the loop. A better approach would presumably be to incorporate Global-Arrays-style one-sided-communication in the compiler run-time, allowing relatively straightforward translation of general task-parallel loops.

```
      subroutine uhf_energy( g_vecs, eone, etwo, enrep, energy,
     $     g_grad )
      implicit none

      ...
      integer d(4), f(4)
      integer g_a_dens, g_a_coul, g_a_exch
      integer g_b_dens, g_b_coul, g_b_exch
      ...
c
c     Arrays for AO density, coulomb and exchange matrices
c
      g_a_coul = ga_create_atom_blocked(geom, basis, 'uhf:a coul')
      g_b_coul = ga_create_atom_blocked(geom, basis, 'uhf:b coul')
      g_a_exch = ga_create_atom_blocked(geom, basis, 'uhf:a exch')
      g_b_exch = ga_create_atom_blocked(geom, basis, 'uhf:b exch')
      g_a_dens = ga_create_atom_blocked(geom, basis, 'uhf:a dens')
      g_b_dens = ga_create_atom_blocked(geom, basis, 'uhf:b dens')
c
c     Make the densities and build the fock matrices
c
      ... initialization

      d(1) = g_a_dens
      d(2) = g_a_dens
      d(3) = g_b_dens
      d(4) = g_b_dens
      f(1) = g_a_coul
      f(2) = g_a_exch
      f(3) = g_b_coul
      f(4) = g_b_exch
      ...
      call fock_2e(geom, basis, 4, jfac, kfac, tol2e,
     $             oskel, d, f)
      ...
      if (.not. ga_destroy(g_a_dens)) call errquit('uhf_e: destroy',0)
      if (.not. ga_destroy(g_b_dens)) call errquit('uhf_e: destroy',0)
      if (.not. ga_destroy(g_a_exch)) call errquit('uhf_e: destroy',0)
      if (.not. ga_destroy(g_b_exch)) call errquit('uhf_e: destroy',0)
      if (.not. ga_destroy(g_a_coul)) call errquit('uhf_e: destroy',0)
      if (.not. ga_destroy(g_b_coul)) call errquit('uhf_e: destroy',0)
      ...
      end



      integer function ga_create_atom_blocked(geom, basis, name)
      ...
c
c     Allocate a global array that is distributed so that atom
c     blocks are not spilt between processors.
c
      ...
      integer map1(max_nproc), map2(max_nproc)

      ... compute `nbf', `nblock1', `map1', `nblock2', `map2' for set of
          atoms and basis functions ...

      status =  ga_create_irreg(MT_DBL, nbf, nbf, name,
     $     map1, nblock1, map2, nblock2, g_a)
      ...

      ga_create_atom_blocked = g_a
c
      end
```

Figure 2: Procedures `uhf_energy` and `ga_create_atom_blocked`

```
      subroutine ao_fock_2e( geom, basis, nfock, jfac, kfac,
     $                    tol2e, oskel, vg_dens, vg_fock )
c
c  Distributed-data AO 2e-Fock construction routine
c
      ...
      integer l_dij, l_dik, l_dli, l_djk, l_dlj, l_dlk
      integer l_fij, l_fik, l_fli, l_fjk, l_flj, l_flk
      integer k_dij, k_dik, k_dli, k_djk, k_dlj, k_dlk
      integer k_fij, k_fik, k_fli, k_fjk, k_flj, k_flk
      ...
      logical status
      ...
c
c     allocate necessary local temporary arrays on the stack
c
c     l_d** ... ** block of density matrix
c     l_f** ... ** block of fock matrix
c
c     k_* are the offsets corrsponding to the l_* handles
c
      ...
c
c     Determine appropriate task chunking and max no. of bf in a
c     block of the density/fock matrix
c
      ...
      if (.not. ma_push_get(HT_INT, 2*natoms, 'fock2e:block',
     $     l_block, k_block))call errquit('fock_2e: ma failed',2*natoms)
      call fock_2e_block_atoms(basis, oskel, tol2e,
     $     int_mb(k_block), nblock, maxblock)
      ...
      status = .true.
      status = status .and. ma_push_get(HT_DBL, maxd, 'dij',
     $     l_dij, k_dij)
      status = status .and. ma_push_get(HT_DBL, maxd, 'dik',
     $     l_dik, k_dik)
      ... similarly `dli', `djk', ..., `flk', `fjk' ...
      status = status .and. ma_push_get(HT_DBL, maxd, 'flj',
     $     l_flj, k_flj)
      status = status .and. ma_push_get(HT_DBL, maxd, 'flk',
     $     l_flk, k_flk)
      status = status .and. ma_push_get(HT_DBL, ablklen, 'atmp',
     $     l_atmp, k_atmp)

      ...
         call fock_2e_a( geom, basis, nfock, ablklen,
     $        jfac, kfac, tol2e, oskel,
     $        dbl_mb(k_dij), dbl_mb(k_dik), dbl_mb(k_dli),
     $        dbl_mb(k_djk), dbl_mb(k_dlj), dbl_mb(k_dlk),
     $        dbl_mb(k_fij), dbl_mb(k_fik), dbl_mb(k_fli),
     $        dbl_mb(k_fjk), dbl_mb(k_flj), dbl_mb(k_flk),
     $        dbl_mb(k_atmp), vg_dens, vg_fock,
     $        int_mb(k_block), nblock)
      ...
c
      status = .true.
      status = status .and. ma_pop_stack(l_atmp)
      status = status .and. ma_pop_stack(l_flk)
      status = status .and. ma_pop_stack(l_flj)
      ... similarly `fjk', `fli', ..., `djk', `dli' ...
      status = status .and. ma_pop_stack(l_dik)
      status = status .and. ma_pop_stack(l_dij)
      status = status .and. ma_pop_stack(l_block)
      ...
      end
```

Figure 3: Subroutine ao_fock_2e

```
      subroutine fock_2e_a( geom, basis, nfock, ablklen,
     $     jfac, kfac, tol2e, oskel,
     $     dij, dik, dli, djk, dlj, dlk,
     $     fij, fik, fli, fjk, flj, flk,
     $     tmp, vg_dens, vg_fock,
     $     blocks, nblock)
      ...
      double precision dij(nfock*ablklen),dik(nfock*ablklen)
      double precision dli(nfock*ablklen),djk(nfock*ablklen)
      ... similarly 'dlj', 'dlk', 'fij', 'fik', 'fli', 'fjk', 'flj', 'flk' ...
      integer vg_dens(nfock)
      integer vg_fock(nfock)
      integer blocks(2,*)
      ...
      integer ijk_prev(3,2)     ! (i/j/k, lo/hi)

      ... set 'ijk_prev' elements to -1
      ...
      ijkl = 0
      next = nxtask(nproc, 1)
c
c     Loop thru blocked atomic quartets
c
      ...
      do ib = nblock, 1, -1
         iatlo = blocks(1,ib)
         iathi = blocks(2,ib)
         ...
         do jb = 1, ib
            jatlo = blocks(1,jb)
            jathi = blocks(2,jb)
            if (... some ij pair survive screening ...) then
               do kb = ib, 1, -1
                  katlo = blocks(1,kb)
                  kathi = blocks(2,kb)
                  lbhi = kb
                  if (ib .eq. kb) lbhi = jb
                  do lb = 1, lbhi
                     latlo = blocks(1,lb)
                     lathi = blocks(2,lb)
                     ... set 'otest' if some kl pair survive screening and
                         interaction is ''symmetry-unique'' ...
c
c     Load balance over non-zero interactions
c
                     if (otest .and. (ijkl .eq. next)) then
                        call fock_2e_task(
     $                       geom, basis, oskel,
     $                       iatlo, jatlo, katlo, latlo,
     $                       iathi, jathi, kathi, lathi,
     $                       ijk_prev,
     $                       nfock, vg_dens, vg_fock,
     $                       jfac, kfac,
     $                       dij, dik, dli, djk, dlj, dlk,
     $                       fij, fik, fli, fjk, flj, flk,
     $                       tmp, tol2e, dentol)
c
                        next = nxtask(nproc, 1)
                     end if
                     if (otest) ijkl = ijkl + 1
                  end do
               enddo
            endif
         end do
      end do

      *** Final conditional 'fock_upd_blk' calls removed from here ***

      next = nxtask(-nproc, 1)
      ...
      end
```

Figure 4: Subroutine `fock_2e_a`

```
      subroutine fock_2e_task(
     $     geom, basis, oskel,
     $     iatlo, jatlo, katlo, latlo,
     $     iathi, jathi, kathi, lathi,
     $     ijk_prev,
     $     nfock, vg_dens, vg_fock,
     $     jfac, kfac,
     $     dij, dik, dli, djk, dlj, dlk,
     $     fij, fik, fli, fjk, flj, flk,
     $     tmp, tol2e, dentol)
      ...
c
c     Given an block of atomic quartets, fetch the necessary blocks
c     of the density matrices, call fock_2e_b to add in
c     in the fock matrix contribution and then accumulate the contributions.
c
      ...
      integer ijk_prev(3,2)
      ...
      integer vg_dens(nfock), vg_fock(nfock)
      ...
      double precision dij(nfock,*), dik(nfock,*), dli(nfock,*),
     $     djk(nfock,*), dlj(nfock,*), dlk(nfock,*)
      double precision fij(nfock,*), fik(nfock,*), fli(nfock,*),
     $     fjk(nfock,*), flj(nfock,*), flk(nfock,*)
      ...

      ... compute ilo, ihi, jlo, jhi, klo, khi, llo, lhi for block ...

      call fock_2e_cache_dens_fock(
     $     ilo, jlo, klo, llo, ihi, jhi, khi, lhi, ijk_prev,
     $     nfock, vg_dens, vg_fock,
     $     jfac, kfac,
     $     dij, dik, dli, djk, dlj, dlk, fij, fik, fli, fjk, flj, flk, tmp)

      ...
         do iat = iatlo, iathi
            ... set 'jattop'
            do jat = jatlo, jattop
               if (... ij pair survive screening ...) then
                  ... set 'kattop'
                  do kat = katlo, kattop
                     ... set 'lattop'
                     do lat = latlo,lattop
                        ... set 'otest' if kl pair survive screening and no
                            symmetry-equivalent interaction appeared before ...

                        if (otest) then
                           call fock_2e_b(basis, nfock, sijkl, tol2e,
     $                          dentol, q4, iat, jat, kat, lat,
     $                          ilo, jlo, klo, llo, ihi, jhi, khi, lhi,
     $                          dij, dik, dli, djk, dlj, dlk,
     $                          fij, fik, fli, fjk, flj, flk)
                        end if
                     end do
                  end do
               end if
            end do
         end do
       ...
c
c     Update F blocks
c
      *** First three 'fock_upd_blk' calls moved from 'fock_2e_cache_dens_fock' ***

      call fock_upd_blk(nfock, vg_fock, ilo, ihi, jlo, jhi, jfac, fij, tmp )
      call fock_upd_blk(nfock, vg_fock, ilo, ihi, klo, khi, kfac, fik, tmp)
      call fock_upd_blk(nfock, vg_fock, jlo, jhi, klo, khi, kfac, fjk, tmp)
      call fock_upd_blk(nfock, vg_fock, llo, lhi, ilo, ihi, kfac, fli, tmp)
      call fock_upd_blk(nfock, vg_fock, llo, lhi, jlo, jhi, kfac, flj, tmp)
      call fock_upd_blk(nfock, vg_fock, llo, lhi, klo, khi, jfac, flk, tmp)
      ...
      end
```

Figure 5: Subroutine `fock_2e_task`

```
      subroutine fock_2e_cache_dens_fock(
     $     ilo, jlo, klo, llo,
     $     ihi, jhi, khi, lhi,
     $     ijk_prev,
     $     nfock, vg_dens, vg_fock,
     $     jfac, kfac,
     $     dij, dik, dli, djk, dlj, dlk,
     $     fij, fik, fli, fjk, flj, flk,
     $     tmp)
      ...
c
c     For the given ranges of i,j,k,l fetch the six blocks
c     of the density matrices (ij,ik,il,jk,jl,kl) and accumulate
c     the fock matrix blocks.  Scale the fock matrix blocks
c     by the necessary factors for Coulomb/Exchange.
c     Take advantage of caching by storing in ijk_prev() the
c     previous i,j,k ranges (l assumed to be inner loop and
c     thus not to benefit from caching).  If any of ij,ik,jk
c     have not changed then nothing need be done for those blocks.
c
      ...
      integer ijk_prev(3,2)     ! old values (i/j/k, lo/hi)
      integer nfock             ! No. of matrices
      integer vg_dens(nfock), vg_fock(nfock) ! GA handles
      double precision jfac(nfock), kfac(nfock)
      double precision
     $     dij(nfock,*), dik(nfock,*), dli(nfock,*),
     $     djk(nfock,*), dlj(nfock,*), dlk(nfock,*)
      double precision
     $     fij(nfock,*), fik(nfock,*), fli(nfock,*),
     $     fjk(nfock,*), flj(nfock,*), flk(nfock,*)
      double precision tmp(*)
c
      ... set 'idim', 'jdim', ..., to 'ihi - ilo + 1', etc ...
c
c     Blocks of D/F without l label ... caching is useful
c
      *** 'fock_upd_blk' and 'dfill' calls removed from following IF constructs ***

      if (ijk_prev(1,1).ne.ilo .or. ijk_prev(2,1).ne.jlo)
     $   call fock_get_blk(nfock, vg_dens,
     $        ilo, ihi, jlo, jhi, dij, tmp)
      if (ijk_prev(1,1).ne.ilo .or. ijk_prev(3,1).ne.klo)
     $   call fock_get_blk(nfock, vg_dens,
     $        ilo, ihi, klo, khi, dik, tmp)
      if (ijk_prev(2,1).ne.jlo .or. ijk_prev(3,1).ne.klo)
     $   call fock_get_blk(nfock, vg_dens,
     $        jlo, jhi, klo, khi, djk, tmp)
c
c     Blocks with l label always change
c
      call fock_get_blk(nfock, vg_dens,
     $     llo, lhi, klo, khi, dlk, tmp)
      call fock_get_blk(nfock, vg_dens,
     $     llo, lhi, ilo, ihi, dli, tmp)
      call fock_get_blk(nfock, vg_dens,
     $     llo, lhi, jlo, jhi, dlj, tmp)

      *** First three 'dfill' calls moved from IF constructs above ***

      call dfill((idim*jdim*nfock), 0.0d0, fij, 1)
      call dfill((idim*kdim*nfock), 0.0d0, fik, 1)
      call dfill((jdim*kdim*nfock), 0.0d0, fjk, 1)
      call dfill((kdim*ldim*nfock), 0.0d0, flk, 1)
      call dfill((idim*ldim*nfock), 0.0d0, fli, 1)
      call dfill((jdim*ldim*nfock), 0.0d0, flj, 1)
c
      ijk_prev(1,1) = ilo
      ijk_prev(2,1) = jlo
      ... similarly save current 'klo', 'ihi', 'jhi', 'khi' in 'ijk_prev' ...

      ...
      end
```

Figure 6: Subroutine `fock_2e_cache_dens_fock_2e_task`