Multiple data parallelism with HPF and KeLP*

John H. Merlin[1], Scott B. Baden[2], Stephen J. Fink[2], and Barbara M. Chapman[1]

[1] VCPC, Institute for Software Technology and Parallel Systems,
University of Vienna, Liechtensteinstr. 22, A-1090 Vienna, Austria.
{jhm, barbara}@vcpc.univie.ac.at
[2] Department of Computer Science and Engineering,
University of California, San Diego, La Jolla, CA 92093-0114, USA.
{baden, sfink}@cs.ucsd.edu

Abstract. High Performance Fortran (HPF) is an effective language for implementing regular data parallel applications on distributed memory architectures, but it is not well suited to irregular, block-structured applications such as multiblock and adaptive mesh methods.
A solution to this problem is to use a non-HPF SPMD program to coordinate multiple concurrent HPF tasks, each operating on a regular subgrid of an irregular data domain. To this end we have developed an interface between the C++ class library KeLP, which supports irregular, dynamic block-structured applications on distributed systems, and an HPF compiler, SHPF. This allows KeLP to handle the data layout and inter-block communications, and to invoke HPF concurrently on each block.
There are a number of advantages to this approach: it combines the strengths of both KeLP and HPF; it is relatively easy to implement; and it involves no extensions to HPF or HPF compilers.
This paper describes the KeLP-HPF implementation and programming model, and shows an example KeLP-HPF multiblock solver.

1 Introduction

High Performance Fortran (HPF [1, 2]) provides a simple, high level programming model for implementing regular data parallel applications on distributed memory architectures. For such applications it can greatly ease program development and maintenance and provide high performance.

However, HPF by itself is not well suited to irregular, block-structured data parallel applications. Such applications operate on domains consisting of a set of regular grids of various sizes, which are arranged irregularly and may be created dynamically. Examples include multiblock and adaptive mesh methods [3, 4].

Such applications can exploit data parallelism at two levels: within grids and between them. To exploit both types of parallelism on distributed memory systems one needs to distribute individual grids over subsets of processors, allocating different grids to different (possibly overlapping) processor subsets. Each

* Correspondence should be addressed to John Merlin.

grid should be distributed over a number of processors proportional to its size in order the balance the workload across the processors. Irregular patterns of communication must also be managed between the grids.

Such a model is not currently supported by commercial HPF compilers, which offer only static uniform data decompositions. Indeed, this limitation has motivated the development of a set of 'HPF 2.0 Approved Extensions' [1]. These extensions distribute the grids at runtime onto subsets of processors and specify how computations are to be mapped onto the processors [5].

This paper describes an alternative approach that does not rely on language extensions. Instead we use an external SPMD program to coordinate multiple concurrent HPF tasks and to perform the required data movements between them.

We employ the C++ class library *KeLP* [6, 7] as the SPMD coordination layer. KeLP can dynamically create and distribute multiblock domains, and it allows inter-block communication patterns to be constructed at runtime and preprocessed for efficiency. It was originally designed to invoke serial Fortran 77 kernels to perform the 'internal' computations on the subgrid stored on each processor. Here we have built a simple interface from KeLP to HPF. This allows KeLP to invoke multiple data parallel HPF tasks concurrently, each operating on an individual distributed grid. Our implementation of the interface works in conjunction with the public domain HPF compiler *SHPF* [8].

There are good reasons for using the KeLP-HPF approach for implementing 'multiple data parallelism', quite apart from the current difficulties of using pure HPF for this purpose. First, it combines the strengths of both languages: KeLP for concisely expressing block-structured computations and efficiently implementing the associated inter-block communications, and HPF for performing the regular data parallel computations within each block. Second, the process of interfacing KeLP to HPF is straightforward: it does not require the development of new language constructs or compiler technology, and it builds on an already existing SPMD class library. Third, such an approach is attractive on systems with hierarchical memory, such as SMP-based multicomputers. Finally, this approach may be generalised; we expect that similar techniques could be used to interface HPF to other SPMD coordination libraries, perhaps for the purpose of supporting other application areas.

The next section gives an overview of the components of our system, KeLP and SHPF. Section 3 describes the implementation of the interface between them. Section 4 summarises the main KeLP-HPF language elements and explains the KeLP-HPF programming and execution models. Section 5 briefly compares our approach with related work, and Section 6 presents our conclusions. Finally, we show the KeLP source code for a KeLP-HPF version of a multiblock solver in an Appendix.

2 The components of the KeLP-HPF implementation

Here we briefly introduce the two components of our system: KeLP and SHPF. Both are available in the public domain and have been installed on a wide range of parallel and distributed architectures. The reader is referred to the references provided for more details about these systems.

2.1 KeLP

KeLP [6, 7] is a C++ class library built on the standard Message Passing Interface, MPI [9]. It supports general block decompositions via the following classes which are used to manage data decomposition at run time (where i is an integer in the range 1–4):

`Region`i: which represents an i-dimensional index space, that is, a rectangular subset of Z^i. It provides geometric operations over `Regions` such as `shift`, `grow` and `intersect` that help to specify data motion between them.
`Grid`i: which represents an i-dimensional data array instantiated over a `Region`i. A `Grid` object is analogous to a Fortran allocatable array.
`XArray`i: which represents an array of `Grid`i objects with processor assignments. Each `Grid` may have a different origin and size, but they all have the same datatype and dimensionality. Each `Grid` is stored on just one processor. This class allows the representation of arbitrary irregular block-structured data decompositions.

The basic unit of communication in KeLP is the *collective block copy* operation, which copies regular sections between two (not necessarily distinct) `XArray`s. KeLP provides *communication orchestration* abstractions to express and optimize such communications, which are described in [7].

Finally, KeLP defines a parallel looping construct[1], `for_all`–`end_for_all`, which iterates concurrently over the `Grid`s of an `XArray`. For example, if `X` is an `XArray` then:

```
for_all (i, X)
    ... code
end_for_all
```

executes the code in iteration i on the processor that 'owns' the `Grid X(i)`, in parallel with the other iterations. Typically this code performs serial computation on that `Grid` and possibly others stored on the same processor, and often this is done by passing them to routines written in a serial language such as Fortran 77 or C.

KeLP programs are single-threaded, that is, the same instructions are executed on every processor, except for code enclosed in a `for_all`–`end_for_all` construct. Data layout and communication operations must be performed in the single-threaded code.

[1] Actually `for_all` and `end_for_all` are macros.

2.2 SHPF

SHPF [8] is a public domain HPF 2.0 compilation system. It comprises a translator that translates HPF into Fortran 90 SPMD code that runs on each node of the target computer, and a runtime library, ADLIB, that is called by the Fortran 90 code to perform communications. ADLIB [10], like KeLP, is a C++ class library that calls MPI to perform message-passing.

3 Implementation of the KeLP-SHPF interface

Our programming model assumes that a KeLP program executes in SPMD mode on all processors and invokes concurrent HPF procedures[2] on subsets of processors. This section describes the implementation of this model. The principal issues that must be addressed are how the KeLP program allocates HPF-style distributed arrays, passes them to HPF procedures, and manages concurrent HPF invocations over subsets of processors.

Incidentally, by '*processor*' we mean an 'MPI process' unless otherwise stated.

3.1 HPF array allocation and argument passing

If KeLP is to call separately-compiled HPF procedures it must pass arguments to them in the same way that compiled HPF code would. The details of how arguments are passed in HPF depend on which HPF compiler is used. With the SHPF compiler, each array argument is accompanied by an extra argument—not visible in the original HPF code—called a *mapping descriptor*, which contains complete information about the array's mapping, i.e. its shape, alignment and distribution. This convention is used by most HPF compilers, although the details of how the information is stored in the descriptor is compiler-dependent.

In our implementation of KeLP-HPF, KeLP is responsible for creating the variables that it passes to HPF procedures. This is a natural strategy, since KeLP always calls HPF and not vice versa.[3] Again, KeLP must ensure that the variables that it passes to HPF conform to the HPF procedure's expectations in every respect, that is, in terms of their datatype and, for an array, their shape and layout. Note that C++ and Fortran store array elements in a different order: row major and column major respectively. KeLP resolves this conflict by using Fortran's column-major ordering. This is transparent to the user because all accesses to array elements in KeLP are via class functions, so the array's layout is hidden from the user.

In general HPF compilers may also change the array layout in various ways, e.g. by adding a layer of 'ghost cells' around the boundary of each local segment.

[2] A *procedure* in Fortran or HPF means a function or subroutine.

[3] The alternative strategy, in which HPF allocates its own variables and passes them back to KeLP via the HPF procedure interface is both unnatural and very difficult to implement, given that Fortran does not allow arguments to be `ALLOCATABLE` arrays and Fortran pointers have a totally different representation to C++ ones.

In that case KeLP must arrange for its 'HPF arrays' to have the same layout. Once again this should be transparent to the user since the array layout is encapsulated in a class. SHPF does not require the addition of ghost cells however, so this issue did not arise here.

To fulfill these requirements the following new classes were derived from the KeLP classes (where i is an integer in the range 1–4):

`HPFprocs`i`:` which represents an i-dimensional HPF virtual processor array together with its mapping onto the processors or a subset of them. The constructor for this class performs the SHPF runtime library initialisation for the processor subset as described below.

`HPFmap`i`:` which represents an HPF mapping of an i-dimensional array, that is, an i-dimensional index space together with its distribution over an HPF processor array (i.e. an `HPFprocs`i object). An object of this class is analogous to an HPF template together with its mapping. This class stores the SHPF mapping descriptor.

`HPFarray`i`:` which represents an i-dimensional HPF distributed array together with its HPF mapping, as specified by an `HPFmap`i object.

The `HPFarray` class is derived from KeLP's `XArray` class. It is basically an `XArray` whose `Grids` are arranged in a regular rectangular layout corresponding to the local segments of an HPF distributed array. An `HPFarray` object is instantiated with an HPF-level description of the whole array, i.e. its bounds, distribution[4] and target processor array, and its constructor computes the bounds and processor assignments of the component `Grids`.

Two macros were defined to assist in calling HPF procedures from KeLP:

`HPF_ARRAY_ARG (A, i)`, which takes a KeLP `HPFarray` object `A` and expands it to *local-segment-of-array, descriptor* for use as an HPF procedure argument;

`HPF_ARRAY_PROT (type)`, which expands to *type* `*, int *` for use in C++ prototypes of HPF procedures to declare the datatype of an array argument and its descriptor.

3.2 Invoking HPF on processor subgroups

The integration also involved enhancing SHPF's run-time library, ADLIB, to support the execution of HPF code on a dynamically defined subset of the processors assigned to the run. This is necessary because KeLP-HPF is intended to allow multiple HPF procedures to be invoked concurrently on different `HPFarrays` stored on different subsets of processors.

This turned out to be a simple extension to ADLIB, thanks to its clean design and MPI's facilities for creating processor subgroups. It entailed adding 3 new functions.

`ad_subgroup_create` creates a new processor subgroup containing a specified subset of the processors in the current 'active processor group', and returns a handle `p` to that subgroup. This function is invoked by the `HPFprocs` constructor.

[4] Currently only HPF's `BLOCK` distribution format is supported.

`ad_on(p)` makes the subgroup with handle `p` the new active processor group, and `ad_no()` changes the active processor group back to what it was before the matching `ad_on`. Code between `ad_on`–`ad_no` calls is only executed on the new subgroup `p`. These calls can be nested to allow successive processor subsetting (although this facility is not used here).

To hide the `ad_on` and `add_no` calls from the KeLP programmer, new macros `HPF_for_all` and `HPF_end_for_all` were defined for parallel looping over processor subgroups when invoking an HPF procedure. Thus:

```
HPF_for_all (i, A)    // 'A' is an 'HPF_array' object stored
                      // on the processor subset of interest
    ... HPF procedure call(s)
HPF_end_for_all
```

expands to:

```
for_all (i, A)
    ad_on (A.subgroup_handle ());
        ... HPF procedure call(s)
    ad_no ();
end_for_all
```

Incidentally, no synchronisation barrier is introduced when calling or returning from an HPF procedure.

3.3 Other points

The following points must also be considered when calling SHPF-compiled HPF procedures from KeLP:

- KeLP's `main` function must call ADLIB's startup and termination routines: `InitHPF` and `FinishHPF`.
- Fortran passes all arguments *by reference*, unlike C++.
- SHPF, in common with many Fortran and HPF compilers, changes procedure names according to a particular 'name mangling' convention.

The last point is handled by defining a macro, `HPF_SUBR (name)`, which converts an HPF subroutine name into the form expected by SHPF.

We have only considered a simple, restricted interface between KeLP and HPF. That is, we only call HPF subroutines, not functions, to avoid the complication of handling return values (which may be array-valued), and we have only considered passing arguments that are scalars or arrays of primitive datatypes. This simplifies the implementation and makes it portable to different C++ compilers and Fortran 90 'backend' compilers for SHPF.

4 Using KeLP-HPF

With these elements in place, we shall now give some brief examples of using the KeLP-HPF interface.

The KeLP code to create an 'HPF array' `A(nx,ny)` of type `double` that is distributed `(BLOCK,BLOCK)` onto a processor array `P(px,py)`, starting at processor `p0`, is as follows:

```
HPFprocs2  P (px, py, p0);  // create HPF processor subgroup P
HPFmap2    T (nx, ny);      // create HPF template T
T.distribute (BLOCK, BLOCK, P);  // distribute T onto P
HPFarray2 <int>  A (T);     // create array A aligned with T
```

The following KeLP code calls an HPF subroutine on the processor subgroup that stores array `A`, passing it arguments `A`, `px` and `py`:

```
HPF_for_all (i, A)
    HPF_SUBR(sub) ( HPF_ARRAY_ARG(A,i), & px, & py );
HPF_end_for_all
```

The macro `HPF_SUBR` changes the subroutine name `sub` in accordance with SHPF's 'name mangling' convention, and `HPF_ARRAY_ARG` expands the `HPFarray` object `A` into two comma-separated arguments—a data array and its mapping descriptor—as expected by SHPF-compiled HPF procedures. The address operator `&` is applied to the scalar arguments `px` and `py` because in Fortran arguments are passed by address rather than by value.

Subroutine `sub` is just a normal HPF subroutine, e.g.:

```
SUBROUTINE sub (A, px, py)
  INTEGER         A (:,:), px, py
!HPF$ PROCESSORS  P (px, py)
!HPF$ DISTRIBUTE  A (BLOCK, BLOCK) ONTO P
  ...
```

It is perhaps worth explaining the KeLP-HPF programming model in more detail. The above `HPF_for_all` construct invokes the HPF subroutine `sub` in parallel on all processors that 'own' a portion of the `HPFarray A`, i.e. all processors over which it is distributed, and the `HPF_ARRAY_ARG` macro passes-in the locally owned segment of `A` (plus a mapping descriptor) to the subroutine on each processor. This is the normal way in which HPF subroutines are called, and the subroutine is unaware that that it has been called from an extrinsic language—it could have been called from HPF.

Furthermore, the subroutine is unaware of the existence of any processors outside of the subgroup of processors on which it is executing (its 'active processor group'), and it cannot access any processors outside of this group. The values returned by the HPF intrinsics `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` are defined by the procedure's active processor group, and collective operations such as broadcasts that occur within the HPF code are restricted to it.

If two `HPFarrays` `A1` and `A2` are stored on different processor subgroups $\{P_1\}$ and $\{P_2\}$, then different HPF procedures can be invoked in parallel over these two arrays using, e.g.:

```
HPF_for_all (i, A1)
    HPF_SUBR(sub1) ( HPF_ARRAY_ARG(A1,i) );
HPF_end_for_all
HPF_for_all (i, A2)
    HPF_SUBR(sub2) ( HPF_ARRAY_ARG(A2,i) );
HPF_end_for_all
```

Processors in group $\{P_1\}$ only execute the call to `sub1` and processors in group $\{P_2\}$ only execute the call to `sub2`, so these calls will proceed in parallel. Similarly, if `A[n]` is an array of `HPFarrays`, each of which is stored on a disjoint sets of processors, then:

```
for (i = 0; i < n; i++) {
    HPF_for_all (j, A[i])
        HPF_SUBR(sub) ( HPF_ARRAY_ARG(A[i],j) );
    HPF_end_for_all
}
```

will cause n separate parallel invocations of `sub`, with invocation i active on the subgroup of processors that owns `A[i]`. These separate invocations are independent and non-interacting.

More generally, different HPF invocations can be active on overlapping, or even the same, processor subgroups. They will still be independent and conceptually concurrent, although in that case they may not execute in parallel.

This programming model places some responsibilities on the programmer to ensure correctness. When calling an HPF subroutine, the programmer must ensure that:

- All arguments have the distribution expected by the subroutine.
- All `HPFarray` arguments are mapped to exactly the set of processors that will execute the subroutine.
- If an HPF dummy argument is *replicated*, then the corresponding actual argument passed from KeLP must have the same value on every processor that executes the subroutine. This is necessary to conform with HPF's semantics, which require that replicated variables are consistent, that is, have the same value on all processors. (Actually this rule is not strictly necessary if the argument has `INTENT (OUT)`).

With respect to the last point, note that the *return* value of a replicated HPF argument is guaranteed to be consistent according to HPF's semantics, provided that the rules for invoking HPF have been obeyed.

Many of the issues governing the coordination of multiple concurrent HPF tasks by a SPMD program are described in a proposal to the HPF Forum [11].

4.1 An example KeLP-HPF multiblock solver

The Appendix shows the KeLP source code (excluding `include` files, preprocessor directives and function prototypes) for a 2-dimensional multiblock solver written in KeLP-HPF. The number of grids, their sizes and coordinates, and their distribution over processors are input at runtime.

KeLP allocates and decomposes the grids and performs the inter-grid communications. To initialise and update the grids it invokes the HPF subroutines `init_grid` and `update` (whose source code is not given since it is straightforward) concurrently on each grid as a separate parallel computation.

The KeLP code should be fairly self-explanatory, except perhaps for the function `update_edges`. This function exchanges the boundary data between all pairs of overlapping grids. Its code is 'standard KeLP', that is, it is unaffected by using HPF invocations in lieu of serial Fortran 77, and a detailed explanation of it can be found in [7].

5 Related work

The work most closely related to ours—in the sense that it uses a communication library to coordinate concurrent data parallel tasks—is HPF/MPI [12], in which HPF tasks communicate by explicit message-passing in MPI. If one ignores the HPF directives in an HPF/MPI program it can be read as a normal Fortran-MPI program, so this approach has both the advantages—flexibility and control—and the disadvantage—complexity—of message-passing programming. In contrast with HPF/MPI, both of the language components of KeLP-HPF are high-level and have no explicit message-passing, and KeLP-HPF requires no changes to be made to HPF code. On a related front, KeLP has been extended to support 'multi-tier' programming on SMP clusters [13]. In effect this model exposes the second level of parallelism to the programmer, with the advantage that it is more flexible than HPF, but the disadvantage of increased code complexity.

Other approaches to coordinating multiple data parallel tasks are based on new languages, e.g. [14], or compiler analysis guided by directives to extract implicit task parallelism, e.g. [15]. In contrast to our approach, these techniques require sophisticated compiler technology and/or the adoption of new languages.

6 Conclusions

We have presented an implementation of KeLP-HPF, in which the C++ class library KeLP is used to coordinate multiple HPF invocations for the purpose of supporting 'multiple data parallel' applications such as multiblock methods and adaptive mesh refinement.

The use of a C++ a class library to coordinate multiple HPF tasks has several advantages. It is relatively straightforward to implement: it involves no new compiler technology and builds on an already existing SPMD class library. It does not require extensions to HPF itself, nor to the target HPF compiler (except to support multi-HPF execution if it is not already supported). It combines the strengths of both languages: KeLP for concisely expressing irregular block-structured computations and efficiently implementing the associated irregular communications, and HPF for performing the regular data parallel computations within each block. By separating the primitive operations supported by each model we simplify the design of performance optimisations. KeLP's object oriented design facilitated the integration with HPF considerably, as we were able to leverage the existing class structure.

Our implementation interfaces KeLP to a particular HPF compiler: SHPF. However, it should be possible to reimplement the classes and macros for other HPF compilers without changing the KeLP-HPF Application Programmer's Interface, provided that the compiler supports the concurrent execution of multiple HPF procedures on processor subgroups[5]. Indeed, it should also be possible, using techniques similar to those described here, to interface HPF to other libraries that provide high-level coordination facilities, perhaps for the purpose of supporting other application areas.

The KeLP-SHPF system is currently installed on clusters of Digital Alpha and Sun workstations and on a Meiko CS2, and installation on a Cray T3E is imminent. We plan to evaluate its performance on various applications on these systems.

Finally, a by-product of this research was an enhanced SHPF that supports multiple HPF invocations. This will also enable task parallelism *within* HPF, and is necessary to support some HPF features in their full generality,[6] such as `PURE` procedures in `FORALL`, `INDEPENDENT` loops and the tasking construct (an HPF-2 'Approved Extension'). We hope to take advantage of this enhancement by developing general implementations of some of these features in SHPF.

The software and documentation for SHPF, KeLP, the KeLP-SHPF interface and the complete example program given in the Appendix are available from `http://www.vcpc.univie.ac.at/~jhm/shpf-kelp/`.

7 Acknowledgments

We would like to thank Bryan Carpenter (NPAC) for providing the extensions to the ADLIB runtime library necessary to support this work, and Chuck Koelbel and Joel Saltz for their collaboration with one of us (Baden) in developing the 'SPMD coordination of HPF' proposal under the auspices of the High Performance Fortran Forum which was organised by Ken Kennedy.

KeLP is part of the dissertation topic of Stephen J. Fink (Ph.D. expected June 1998), who is supported by the DOE Computational Science Graduate Fellowship Program. Scott Baden is supported by The U.S. National Science

[5] We believe, however, that this facility is currently not widely supported by HPF compilers as it is not required for basic HPF 2.0 implementation.

[6] For example, invoking multiple `PURE` procedures in parallel from within a `FORALL`, with individual invocations active on more than one processor.

Foundation Contract ASC-9520372, by the University of California, San Diego (UCSD), and by the NSF-funded National Partnership for Advanced Computational Infrastructure (NPACI), with headquarters at UCSD. KeLP is part of NPACI's Common Adaptive Runtime Environment (CARTE) effort.

References

1. High Performance Fortran Forum: High Performance Fortran Language Specification version 2.0. URL: `http://www.crpc.rice.edu/HPFF/hpf2/` or `http://www.vcpc.univie.ac.at/information/mirror/HPFF/hpf2/`.
2. Merlin, J. H., Chapman, B. M.: High Performance Fortran 2.0, in *Proc. Sommerschule über Moderne Programmiersprachen und Programmiermodelle*, Technical University of Hamburg-Harburg, Sept 15–19, 1997. To appear. Also at URL: `ftp://ftp.vcpc.univie.ac.at/vcpc/jhm/proc/hpf2.ps.gz`
3. Berger, M. J., Oliger J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations, *Jnl. of Computational Physics*, **53**, 3 (March 1984), 484–512.
4. Berger, M. J., Colella P.: Local Adaptive Mesh Refinement for Shock Hydrodynamics, *Jnl. of Computational Physics*, **82**, 1 (May 1989), 64–84.
5. Mehrotra, P., Van Rosendale, J., Zima, H.: High Performance Fortran: History, Status and Future, *Parallel Computing*, Special Issue on Languages and Compilers for Parallel Computers, 1997.
6. Fink, S. J., Baden, S. B., Kohn, S. R.: Flexible Communication Mechanisms for Dynamic Structured Applications, in *Proc. 3rd Int'l Workshop IRREGULAR '96*, 203–215, Aug. 1996.
7. Fink, S. J., Kohn, S. R., Baden, S. B.: Efficient Run-time Support for Irregular Block-Structured Applications, *Jnl. of Parallel and Distrib. Computing*, 1998. To appear.
8. Merlin, J. H., Carpenter, D. B., Hey, A. J. G.: SHPF: a Subset High Performance Fortran Compilation System, *Fortran Journal* (March / April 1996), 2–6.
9. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard, Technical report, University of Tennessee, Knoxville, TN, June 1995.
10. Zhang, G., Carpenter, D. B., Fox, G., Li, X., Li, X., Wen, Y.: PCRC-based HPF Compilation, in *10th International Workshop on Languages and Compilers for Parallel Computing*, 1997, To appear in Lecture Notes in Computer Science.
11. Baden, S. B., Koelbel, C.: Proposed Extension for SPMD calling HPF, HPF Forum proposal, 1996. `ftp://ftp.cs.ucsd.edu/pub/baden/tr/MDP/spmd-hpf.ps.gz`.
12. Foster, I., Kohr, D., Krishnaiyer, R., Choudhary, A.: Double standards: Bringing task parallelism to HPF via the Message Passing Interface, *Proc. Supercomputing '96*, Pittsburg, PA, November 1996.
13. Fink., S. J., Baden, S. B.: Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters, in *Proc. Int'l Scientific Computing in Object-Oriented Parallel Environments Conf.*, Dec 1997, Marina del Rey, CA.
14. Chapman, B., Haines, M., Mehrotra, P., Van Rosendale, J., Zima, H.: Opus: A Coordination Language for Multidisciplinary Applications, *Sci. Prog.*, 1997.
15. Gross, T., O'Halloran, D., Subhlok, J.: Task Parallelism in a High Performance Fortran Framework, *IEEE Parallel and Distributed Technology*, **2** (1994), 16–26.

Appendix: KeLP code for a KeLP-HPF multiblock solver

```
typedef  HPFarray2 <Grid2<double> >  Arr;
int main (int argc, char **argv) {
    int     ngrids, lx, ly, ux, uy, px, py, p0;
    double  diff;
    InitKeLP (argc,argv);  InitHPF ();

// create and initialise the grids
    INPUT (ngrids);
    Arr  * a = new Arr [ngrids];
    for (int i = 0;  i < ngrids;  i++) {
        INPUT (lx);  INPUT (ux);  INPUT (ly);  INPUT (uy);
        INPUT (px);  INPUT (py);  INPUT (p0);
        HPFprocs2 *p = new HPFprocs2 (px, py, p0);
        HPFmap2   *t = new HPFmap2   (lx, ly, ux, uy);
        t->distribute (BLOCK, BLOCK, p);
        a[i].instantiate (t);      // allocate local storage

        HPF_for_all (j, a[i])
            HPF_SUBR(init_grid) (HPF_ARRAY_ARG(a[i], j), & px, & py);
        HPF_end_for_all
    }
// compute the solution
    do {
        for (int i = 0;  i < ngrids;  i++) {
            const int  px = a[i].proc_size(0),  py = a[i].proc_size(1);
            HPF_for_all (j, a[i])
                HPF_SUBR(update) (HPF_ARRAY_ARG(a[i],j), &px, &py, &diff);
            HPF_end_for_all
        }
        update_edges (a, ngrids);
    } while (MAX (diff) > TOL);   // ... a reduction over all processors
    FinishHPF ();  mpExit();
    return(0);
}
void  update_edges (Arr  a [],  const int ngrids) {
    for (int i = 0;  i < ngrids;  i++) {
        for (int j = 0;  j < ngrids;  j++) {
            if (i == j) continue;
            MotionPlan2  m;
            for_1 (p, a[i])
                Region2 inside = grow (a[i](p).region(), -1);
                for_1 (q, a[j])
                    m.CopyOnIntersection (a[i], p, a[j], q, inside);
                end_for
            end_for
            Mover2<Grid2<double>, double> dm (a[i], a[j], m);
            dm.execute();
}   }   }
```