"HPF Optimisation"

D.B. Carpenter

April 1, 1993

1 Baseline

Thesis:

- High Performance Fortran is a very high level language, which will be difficult to execute efficiently in general.
- In important special cases it can be executed efficiently, given careful compiler optimisation.
- In other cases programmers can make code efficiently executable, if they know roughly what the compiler is doing.

Southampton intend to have a research implementation of subset HPF available at the start of the project. Will exploit the parallelism in explicit F90 array syntax (no automatic vectorisation, initially). First implementation will exploit a very high level collective communication library (ADLIB) rearranging data at the level of distributed Fortran arrays.

This will be used as a platform for investigation of HPF translation and optimisation issues.

2 Some issues in optimisation

2.1 Placement of computation

HPF as currently defined provides no explicit control over where calculations are performed, and no explicit guarantees about where the compiler will put them. Current HPF-like systems often use the "owner computes" heuristic

```
TEMPLATE T (1000)
REAL A (1000), B (1000), C
ALIGN A (I) WITH T (I)
ALIGN B (I) WITH T (I)
ALIGN C WITH T (*)

C = A (15) * B (15)
```

Literal-minded "owner computes" means *both* `A (15)` and `B (15)` are broadcast, and the multiplication is repeated everywhere.

Need to develop more efficient heuristics. (In "owner computes" placement of subexpression calculation is an "inherited attribute"—need some "synthesised" component?)

Good schemes have the potential to simultaneously minimise communication, redundant calculation and temporary storage, and maximise effective concurrency.

2.2 Managing local and non-local data in loops

In many cases (especially with problems involving locality of reference) `DO` loops can profitably be split into parts operating on local data and parts involving communication. This may facilitate overlapping of communication and computation, and make calculations on internal data (main part of loop) simpler (vectorisable?).

When the compiler can tell algorithm is very local (eg, Laplace equation), can allocate "overlap areas" at the edge of the local array segment, automatically. Alternatively separate temporary buffers can be allocated for copies of non-local data, and end-ranges of DO-loops treated in separate DO-loop.

Where separation of loop can't be determined statically, use "Inspector executor" method?

Alternative strategy (ADAPT): take redistributed copies of whole distributed arrays, by collective communication, before starting computation loop.

A fully optimising compiler may be able to select any one of these schemes, as appropriate.

2.3 Unpeeling layered run-time libraries

Desirable to have flexible view of borderline between compiler generated code, and libraries used for communication and data redistribution.

ADAPT strategy was to work in terms of high-level collective communication libraries wherever possible. Very general, but performance penalties for not exploiting all static information.

ADLIB implementation modelled on Soton ACL

1. Platform. Low-level message passing and process control (Was Soton VCR. Will be PVM or MPI)

2. ACL channel layer.

3. ACL-I. Support for collective communication of "amorphous" data on per-process level.

4. ACL-II. Collective communication at the distributed array level.

Layering approach not *optimal*, viewed as a run-time library. But high-level of abstraction means code for each layer simple. Suitable for "in-lining", when appropriate static information about parameters available.

Compiler's management of communication could be organized around this conceptual model.