From HPF to Locality-Aware High-Productivity Languages

Hans P. Zima

Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA

and

Institute of Scientific Computing, University of Vienna, Austria

Workshop on Programming Languages for High Performance Computing (HPC WPL)
Albuquerque, NM, December 12-13, 2006
Introduction

The Path Towards a High-Level Language

Why HPF Did Not Succeed

From HPF to Locality-Aware High Productivity Languages

Conclusion

Acknowledgment

* Part of this talk is based on an upcoming paper: Ken Kennedy, Charles Koelbel, and Hans Zima: The Rise and Fall of High Performance Fortran: An Historical Object Lesson Proc. History of Programming Languages III (HOPL III), San Diego, June 2007

* Major contributors to the Chapel features discussed were David Callahan, Brad Chamberlain, and Roxana Diaconescu
Abstraction in Programming

- Programming models and languages bridge the gap between “reality” and hardware – at different levels of abstraction.

- Abstraction implies loss of information, but the right abstractions make it harder to write incorrect programs:
  1. Gain in simplicity, clarity, verifiability, and portability.
  2. Loss of information may lead to performance degradation.

- A successful programming model for high performance computing must find an acceptable compromise between the level of abstraction and the resulting target code performance — with performance having priority.
Current HPC programming is dominated by the use of a standard language (Fortran, C/C++), combined with message-passing (MPI).

MPI has made a tremendous contribution to the field, providing a ubiquitous portable standard.

BUT:

- there is a wide gap between the domain of the scientist and this programming model.
- conceptually simple problems (e.g., stencil computations) can result in very complex programs.
- conceptually simple changes (like replacing a block data distribution with a cyclic distribution) can result in major program modifications.
- exploiting performance may require “heroic” programmer effort.
Contents

1  Introduction
2  The Path Towards a High-Level Language
3  Why HPF Did Not Succeed
4  From HPF to Locality-Aware High Productivity Languages
5  Conclusion
High Performance Fortran (HPF) Language Family
- HPF Predecessors: Kali, CM-Fortran, Fortran D, Vienna Fortran
- Post-HPF Developments: HPF+, JAHPF

OpenMP

ZPL

Partitioned Global Address Space (PGAS) Languages
- Co-Array Fortran, UPC, Titanium

High-Productivity Languages
- Chapel, X10, Fortress
do while (.not. converged)
do 
  do J=1,M
  do I=1,N
    B(I,J) = 0.25 * (A(I-1,J)+A(I+1,J)+A(I,J-1)+A(I,J+1))
  end do
end do
end do
A(1:N,1:N) = B(1:N,1:N)

if (MOD(myrank,2) .eq. 1) then
  call MPI_SEND(B(1,1),N,…,myrank-1,..)
call MPI_RCV(A(1,0),N,…,myrank-1,..)
  if (myrank .lt. s-1) then
    call MPI_SEND(B(1,M),N,…,myrank+1,..)
call MPI_RCV(A(1,M+1),N,…,myrank+1,..)
  endif
else ...
...
endif
The HPF code for the example is far simpler than MPI code (due to explicit communication and explicit data structure management in MPI)

Compiler generated object code can be as good as MPI code:
- parallelization of the loop
- static analysis of access patterns and communication
- aggregation and fusion of communication

BUT:

For more realistic application problems, HPF-1 presented serious difficulties regarding expressivity and performance (in particular, communication overhead, and load balancing).
Programmer needs to model irregular meshes

Access to such a mesh requires at least 2 levels of indirection.

```plaintext
REAL :: X(3,N_NODES), F(6,N_NODES), ...
INTEGER :: IX(4,N_ELEMS) !mesh connectivity
...
do i = 1, N_ELEMS
   do i = 1, 4
      F(:,IX(K,I)) = ...+F(:,IX(K,I))+ ...
   end do
end do
```
Problem: index space locality does not reflect locality in 3D space

Regular block or cyclic distributions cannot effectively deal with such a situation
HPF Goals (1993)

- Provide high-level programming support for scalable parallel computer systems, with particular emphasis on data parallelism.

- Provide a machine-independent programming model characterized by:
  - Global name space
  - Implicit generation of communication based on a high-level specification of data distribution
  - Single-threaded control

- Generate target code performance comparable to the best hand-coded MPI program.

- Define the language as a directive-based extension of Fortran 90.
Experiences with HPF-1 (1993)

- **Initial Response: Welcome by many in the user community**
  1. hope for a high-level programming paradigm providing portability
  2. compiler efforts by Digital, IBM, Thinking Machines, Portland Group (PGI)
  3. up to 17 vendors offering HPF products, 35 major applications in HPF

- **Soon serious problems with performance and portability arose:**
  1. immature compiler technology
  2. lack of key functionality
  3. inconsistency of implementations
  4. complex relationship between source and target programs – difficulty of identifying and correcting performance problems
Compiler technology for Fortran 90 was immature at the time:

- Fortran 90 was a recently completed major extension of F77, providing array operations, modules, dynamic storage allocation, and pointer-based data structures

HPF itself required new compiler technology, which had been implemented only in a few research compilers and in the CM Fortran compiler:

- global distribution analysis
- partitioning of the computation
- generation of communication
- communication optimization

Different compiler vendors often focused on different features for optimization, resulting in diminished (performance) portability

There was heavy pressure on compiler vendors to release early implementations
Problems with HPF-1: Missing Functionality

- **Data distribution**
  - only regular block and cyclic distributions provided
  - all distributions targeted whole processor array
  - this was sufficient for dense and regular array computations, linear algebra
  - no effective support for unstructured, semi-structured (e.g., multi-block, structured AMR, or parallel multigrid), or sparse computations

- **Focus on data parallelism: no support for task parallelism**

- **Based on owner computes paradigm: no adequate support for expressing work distribution, in particular affinity between locus of control and locus of data**

- **These deficiencies resulted in serious performance problems, in particular with respect to communication and load balancing**
HPF compilers performed a source-to-source translation:

\[ \text{HPF} \rightarrow \text{Fortran + MPI} \]

This process generally involved major program transformations, including many optimizations.

As a result, the relationship between source and target programs was difficult for the user to understand.

Some tools were developed that mapped performance properties of the MPI target program back to the HPF source (e.g., *Pablo*, *Medea*) – but still the average user did not know how to modify the source program in order to address a performance problem.
HPF Beyond HPF-1

- **HPF-2 (1997)** addressed some of the shortcomings of HPF-1:
  0 more flexible data distributions (including general block, indirect)
  0 support for distribution to subsets of the processor array
  0 support for expression of affinity
  0 rudimentary tasking facility

- This turned out to be too little, too late

- **HPF+ (1998)**, developed at Vienna University, went a step further and provided high-level control of communication:
  0 asserting invariance of a communication schedule for an irregular loop
  0 allowing explicit specification and management of “halos” in the language

- **JAHPF** assimilated many features of HPF+ via a cooperation with NEC: this led to a successful implementation on the Earth Simulator – e.g., a JAHPF plasma code reached 40% efficiency on the Earth Simulator
Contents

1 Introduction
2 The Path Towards a High-Level Approach
3 Why HPF Did Not Succeed
4 From HPF to Locality-Aware High Productivity Languages
5 Conclusion
HPCS Language Overview

- **HPCS Languages**
  - Chapel (Cascade Project, led by Cray Inc.)
  - X10 (PERCS Project, led by IBM)
  - Fortress (HERO Project, led by Sun Microsystems Inc.)

- **Global name space and global data access**
  - In general, no static distinction between local and global references

- **Explicit high-level specification of parallelism**

- **Explicit high-level locality management**

- **High-level support for distributed collections**

- **Support for data and task parallelism**

- **Object orientation**
Locality Management in Chapel

Locale Set

distribute data

align data with work (affinity)

align data

distribute work

work
Locality Control in Chapel: Basic Concepts

**Locale**
- "locale": abstract unit of locality, bound to an execution
- user-defined views of locale sets
- explicit allocation of data and computations on locales

**Domain**
- first-class entity
- components: index set, distribution, associated arrays, iterators
- different kinds of domains including arithmetic and indefinite

**Array --- Mapping from a Domain to a Set of Variables**

**Object-Oriented Framework for User-Defined Distributions**
- original ideas in Kali and Vienna Fortran
- user can work with distributions at three levels
  - naïve use of a predefined library distribution
  - explicit specification of a distribution by its global mapping
  - explicit specification of a distribution by global mapping and data layout
Example: Matrix-Vector Multiplication (dense)

Version 1

```plaintext
var Mat:   domain(2) = [1..m, 1..n];
var MatCol: domain(1) = Mat(2);
var MatRow: domain(1) = Mat(1);

var A:[Mat]   float;
var v:[MatCol] float;
var s:[MatRow] float;

s = sum(dim=2) [i,j in Mat] A(i,j)*v(j);
```

Version 2:
distributions added, algorithm unchanged

```plaintext
var L: [1..p1,1..p2] locale = reshape(Locales);

var Mat:   domain(2) distributed(myB,myB) on L = [1..m,1..n];
var MatCol: domain(1) aligned(*,Mat(2)) = Mat(2);
var MatRow: domain(1) aligned(Mat(1),*) = Mat(1);

var A:[Mat]   float;
var v:[MatCol] float;
var s:[MatRow] float;

s = sum(dim=2) [i,j in Mat] A(i,j)*v(j);
```
Key Functionality of the Distribution Framework

- **Two levels:** *global mapping* and *layout mapping*

- **User-Defined Global Mappings from Index Sets to Locales**
  - Standard interface for the definition of mapping, distribution segments, sequential and parallel iterators
  - Some functionality provided by the system can be overridden by the user
  - “Standard” distributions (block, block-cyclic, etc.) will be placed in a library
  - Application-specific distributions will be part of specialized libraries

- **User-Defined Layout Specifications**
  - Layout specifies data arrangement within a locale
  - Sparse data structures important target

- **Dynamic Reallocation, Redistribution**

- **High-Level Control of Communication**
  - User-defined specification of halos
  - User-defined assertions on communication
Example: Matrix-Vector Multiplication (Sparse CRS)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>53</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>17</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>93</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>13</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>23</td>
<td>69</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

const D: domain(2) = [1..m,1..n]; /* dense data domain */
var myBRD: BRD = BRD(...);
/* declaration and initialization of sparse subdomain */
const DD: sparse domain(D) distributed(myBRD,CRS()) = …;
var AA:[DD]eltype;var x:[1..n]eltype; var y:[1..m]eltype;
...
forall(i,j)in DD { y(i)=sum reduce (dim=2) A(i,j)*x(j);... }

D^0  | C^0  | R^0  |
---|---|---|
53 | 2 | 1 |
19 | 1 | 2 |
17 | 4 | 3 |
93 | 5 | 4 |

D^1  | C^1  | R^1  |
---|---|---|
21 | 7 | 1 |
16 | 8 | 2 |
72 | 6 | 3 |
13 | 4 | 4 |

D^2  | C^2  | R^2  |
---|---|---|
23 | 2 | 1 |
69 | 3 | 1 |
27 | 1 | 3 |
11 | 4 | 5 |

D^3  | C^3  | R^3  |
---|---|---|
44 | 5 | 1 |
19 | 8 | 2 |
37 | 5 | 3 |
64 | 7 | 4 |

user program
class BRD: Distribution {
    ...
    function map(i:index(source)):locale{...};  /* global mapping for dense domain */
    function GetDistributionSegment(loc:locale):domain(1){...};  /* "box" for loc */
    ...
}

class CRS: LocalSegment {
    const loc: locale = this.getLocale();
    /* declaration of dense and sparse distribution segment for locale loc: */
    const locD: domain(2);
    const locDD: sparse domain(locD) = GetDistributionSegment(loc);
    ...
    const LocalDomain: domain(1)=1..nnz;   /* local data domain */
    /* persistent data structures in the local segment: */
    var cx: [LocalDomain] index(locD(2));   /* column index vector */
    var ro: [l1..u1+1] index(xLocalDomain);  /* row vector */
    ...
    function define_column_vector(): {[z in LocalDomain] cx(z)=nz2x(z)(2)}
    function define_row_vector(): {...}
    ...
    function layout(i: index(D)): index(LocalDomain) return(x2nz(i))
    constructor LocalSegment(){define_column_vector(); define_row_vector(); }
}
HPF was a first major attempt at defining a high-productivity language for programming HPC systems

This approach did not succeed at the time – for a variety of reasons some of which were unrelated to the functionality provided by the language

The HPCS languages revived a key HPF idea--high-level locality awareness--in a more general, object-oriented context; in addition to enhanced functionality

Acceptance of a new language depends on many criteria, including:

- functionality and target code performance
- mature compiler and runtime system technology
- user familiarity with conventional features
- easy integration/migration of legacy codes
- integrated development environment
- flexibility to deal with new hardware developments
- support by funding agencies and major vendors

Rapport Kilocore chip
1024 processing elements announced for 2007