Motivation

Distributed-memory architectures

- Physically distributed memory, disjoint addresses
- Advantages → high price/performance, scalability
- Disadvantages → local address spaces, communication
- Communicate via explicit send/recv messages
- Large messages amortize communication overhead

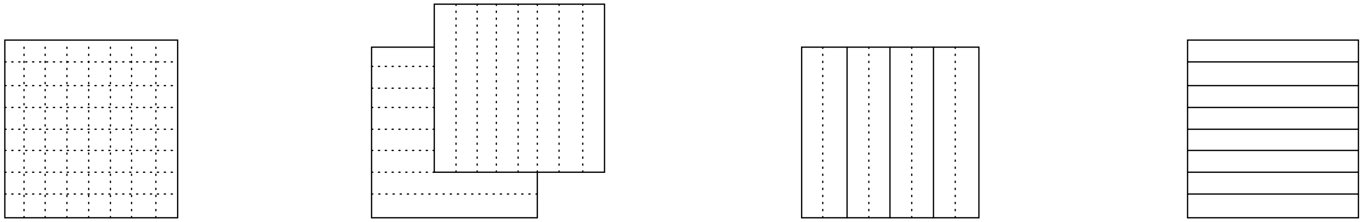
Data-Parallel Languages

- Uniform fine-grain operations on arrays
- Shared data in large, global arrays
- Implicit synchronization between operations
- Implicit communication derived from mapping hints
- Examples: APL, Fortran 90

At one point, data-parallel languages were viewed as the most feasible programming model for large distributed-memory multiprocessors.

High Performance Fortran (HPF)

TEMPLATE → abstract problem domain
ALIGN → map from array to decomposition
DISTRIBUTE → map from decomposition to machine



Example

```
REAL X(8,8)
TEMPLATE A(8,8)
ALIGN X(i, j) WITH A(j+3, i–2)
DISTRIBUTE A(*,BLOCK)
DISTRIBUTE A(CYCLIC,*)
```

FORALL → parallel loop with copy-in/copy-out semantics
INDEP → parallel loop
Intrinsics → parallel functions from Fortran 90

Using HPF

Help analysis with assertions

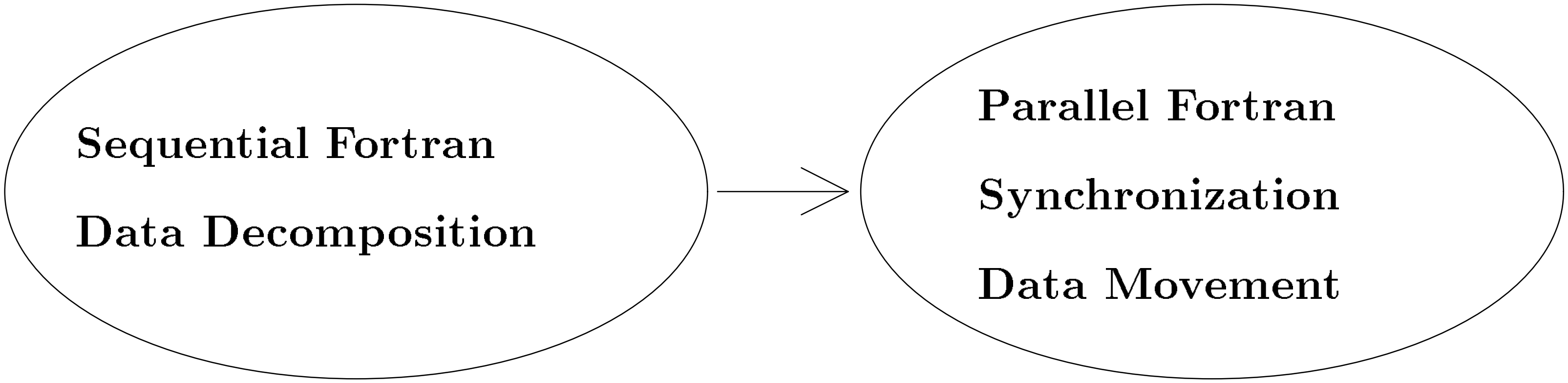
- Align, distribute
- Forall, independent
- Intrinsics

Distribute array dimensions for parallelism

- data updated in parallel should be on different processors
- data used together should be on the same processors

Don't try to hide from compiler what you're doing!

HPF Compiler



Sequential Fortran

Data Decomposition

Parallel Fortran

Synchronization

Data Movement

Requirements

- Partition data & computation
- Generate communication

Single-program, multiple-data (SPMD) node programs

“Owner Computes” Rule

- Owner of datum computes its value
- Dynamic data decomposition

Compiling for Distributed-Memory Machines

Data decomposition

- User-specified (HPF) or automatic
- Derive computation distribution
- Simple decompositions appear sufficient

Compilation process

1) Analyze program → apply dependence analysis
2) Partition data → template, align, distribute
3) Partition computation → owner computes rule
4) Analyze communication → find nonlocal references
5) Optimize communication → select communication
6) Manage storage → select overlaps and buffers
7) Generate code → instantiate partition & messages

Compilation approaches

- Calculates nonlocal data, generates send/recv
- Selects communication type, calls run-time library

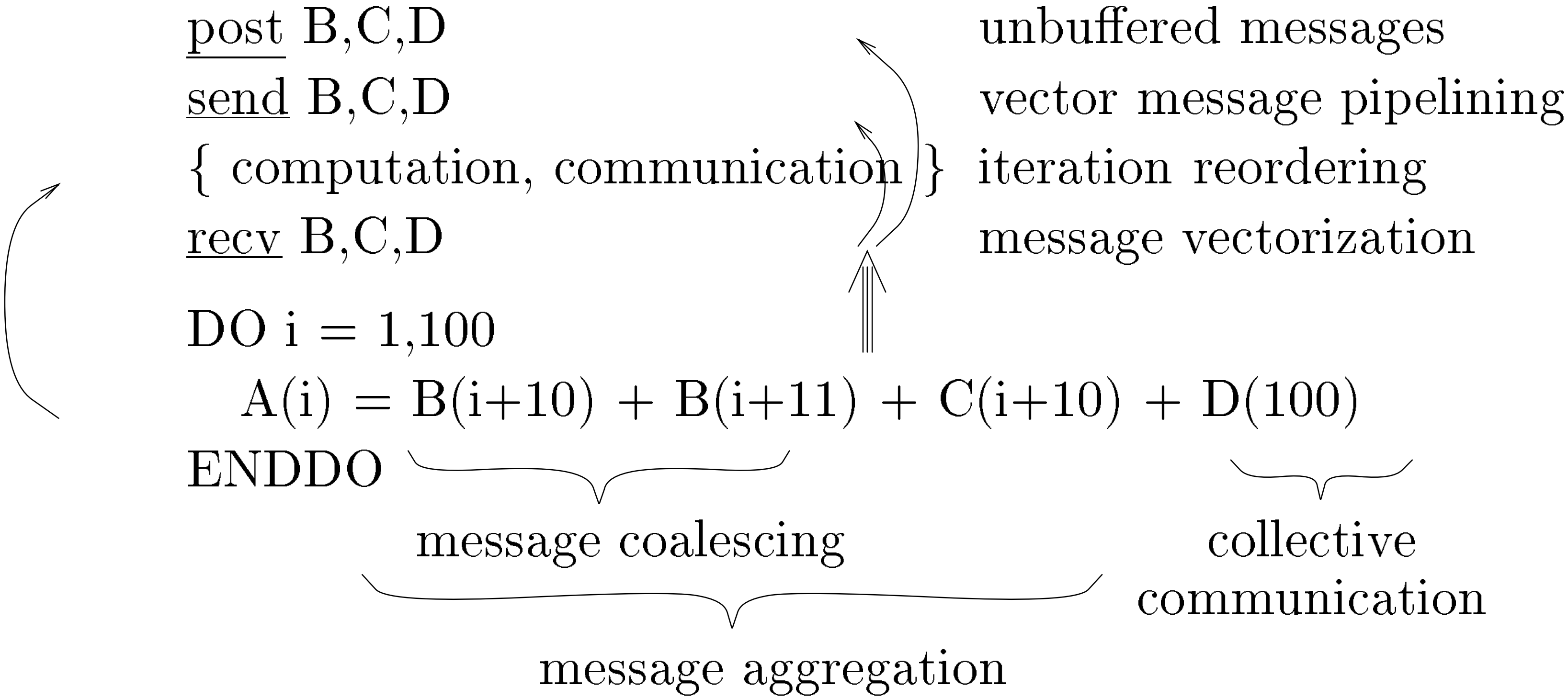
HPF Compilation Example

```
{ HPF Program }
REAL A(100), B(100)
N$PROC = 4
TEMPLATE D(100)
ALIGN A, B WITH D
DISTRIBUTE D(BLOCK)
DO i = 2,100
   A(i) = B(i–1)
ENDDO
```

```
{ Compiler Output }
REAL A(1:25), B(0:25)
P = myproc() { 0 ... 3 }
lb$1 = max(P*25+1,2)-(P*25)
IF (P < 3) send B(25) to P_right
IF (P > 0) recv B(0) from P_left
DO i = lb$1,25
   A(i) = B(i–1)
ENDDO
```

- Local data → A(1:25), B(1:25)
- Local computation → [DO i = 1:25]
- Nonlocal accesses → B(0:24) – B(1:25) = B(0)
- Communication → send B(25) to P_{right}
- Overlap storage → Extend B to hold B(0)

Communication Optimization Example



post B,C,D

send B,C,D

{ computation, communication }

recv B,C,D

unbuffered messages

vector message pipelining

iteration reordering

message vectorization

```
DO i = 1,100
   A(i) = B(i+10) + B(i+11) + C(i+10) + D(100)
ENDDO
```

message coalescing

collective communication

message aggregation

Message Vectorization

Key optimization & code generation technique

Place communication at level of deepest loop that carries a true dependence OR contains endpoints of a loop-independent true dependence

Classify references as independent, carried-all, or carried-part

```
                 DO k = 1,M
                    DO i = 1,N
$\delta_\infty$          A(i) = B(i+2)
$\delta_k$               C(i) = C(i+2)
$\delta_i$               D(i) = D(i-2)
                    ENDDO
                 ENDDO
```

```
send & recv B
DO k = 1,M
   send & recv C
   recv D
   DO i = 1,N/P
      A(i) = B(i+2)
      C(i) = C(i+2)
      D(i) = D(i-2)
   ENDDO
   send D
ENDDO
```

Communication Selection

Utilize Collective Communication Primitives

- Simplifies communication, utilizes efficient primitives
- Syntactic pattern matching

Example

```
TEMPLATE D(N,N)
ALIGN A, B with D
DISTRIBUTE D(BLOCK,BLOCK)
do j = 2,N
   do i = 2,N
      A(i,j) = B(i,j-1)+B(i-1,j)            [shift]
      A(i,j) = B(c,j)                       [broadcast]
      A(c,j) = B(i,j)                       [gather]
      A(i,j) = B(j,i)                       [all-to-all,transpose]
      A(f(i),j) = A(f(i),j)+B(g(i),j)       [inspector/executor]
   enddo
enddo
```

Handling Irregular Accesses

Irregular codes

- Memory access pattern determined by index array
- Value of index array unknown at compile time

Inspector-executor approach

- Compiler inserts call to *inspector* (possible reuse)

...which examines index array, calculates communication

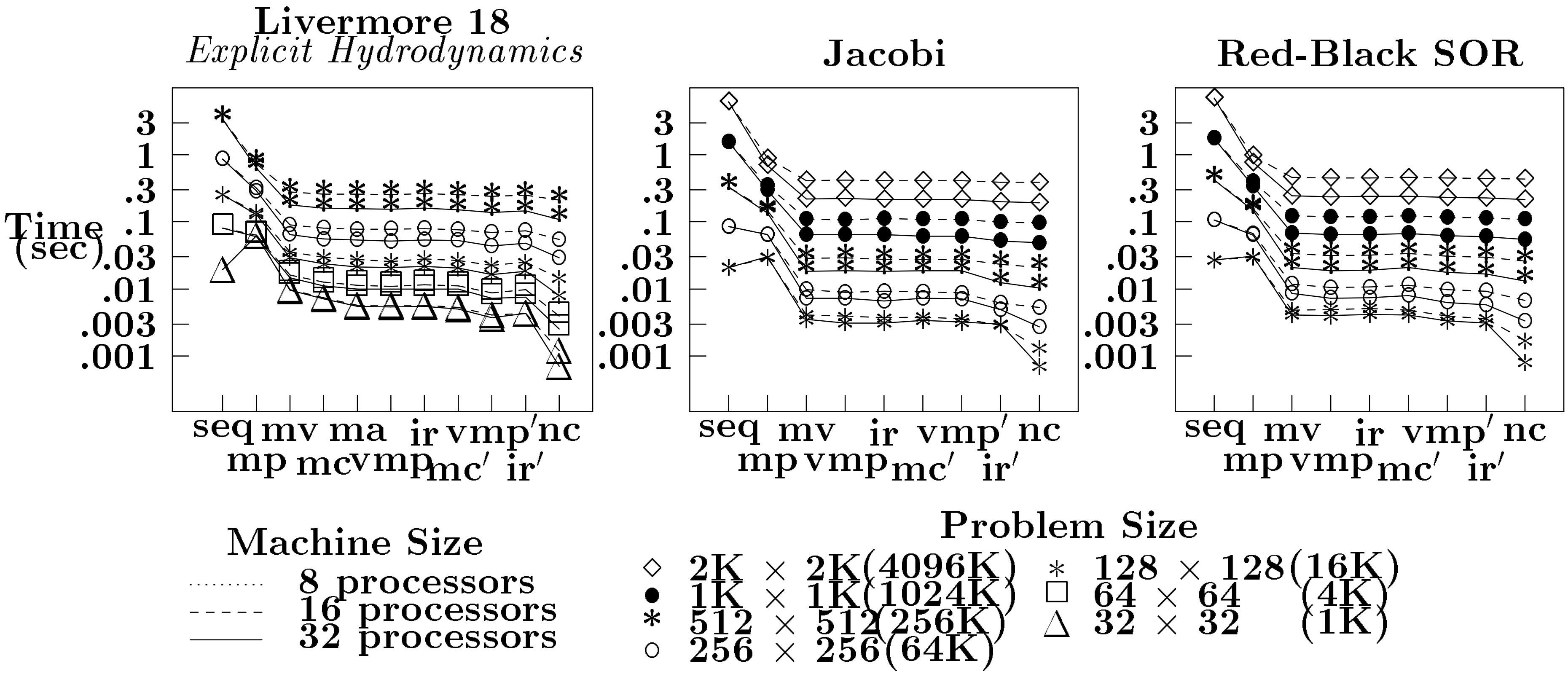
- Compiler transforms loop into *executor*

...which performs communication & computation based on inspector

```
// irregular code
do j = 1,100
  B(...) =
  do i = 1,100
    A(i) = B(IDX(i))
```

```
// compiler output
inspect IDX, calculate MYIDX
do j = 1,100
  B(...) =
  execute communication
  do i = 1,100
    A(i) = B(MYIDX(i))
```

Comparing Communication Optimizations



Experimental evaluation

- Applied communication optimizations by hand
- iPSC/860 timings for different data sizes, # of processors
- Message vectorization (mv) main optimization

HPF Experience

Successes

- Standardized data-parallel languages
- Language quickly adopted ($<$ 2 year)
- Multiple commercial compilers implemented
- Extensions proposed for HPF-2

Failures

- Initial compilers poor
- Performance unstable
- Support for complex applications limited
- Bleeding-edge users preferred message-passing standard (MPI)
- Casual users avoided distributed-memory multiprocessors