TOWARD EXASCALE COMPUTING
WITH FORTRAN 2015 ON INTEL MANY-CORE PROCESSORS

Damian Rouson
Alessandro Fanfarillo
Overview

- Parallelism in Fortran 2008
- Exascale challenges
- Fortran 2015 features for the exascale era
- Results & Conclusions
Overview

Parallelism in Fortran 2008
- Single Program Multiple Data (SPMD)
- Partitioned Global Address Space (PGAS)

Exascale challenges

Fortran 2015 features for the exascale era

Results & Conclusions
Images → \{\text{processes} \mid \text{threads}\}

Images execute asynchronously up to a programmer-specified synchronization:

\begin{align*}
sync & \text{all} \\
sync & \text{images} \\
\text{allocate/deallocate}
\end{align*}
Partitioned Global Address Space (PGAS)

Coarrays integrate with other languages features:

- Fortran 90 array syntax works on local data.
- Communicate objects

The ability to drop the square brackets harbors two important implications:

- Easily determine where communication occurs.
- Parallelize legacy code with minimal revisions.

```
1 program main
2 implicit none
3 type foo; end type
4 type(foo), allocatable :: a(:,:,)
5 integer, parameter :: local_size=5
6 allocate(a(local_size)[*],source=foo())
7 if (ubound(a,1)<local_size .or. ubound(a,1)<3) &
8   error stop "insufficient data"
9 associate(me=>this_image(),n=>num_images())
10  if (me<n) a(1:2) = a(2:3)[me+1]
11  if (me==1) a(4)[2] = a(5)[3]
12 end associate
13 end program
```
if (me<n) a(1:2) = a(2:3)[me+1]
if (me == 1) a[4][2] = a[5][3]
Overview

Parallelism in Fortran 2008

Exascale challenges
- Events
- Teams
- Collective subroutines
- Failed-image detection
- Richer set of atomic subroutines

Fortran 2015 features for the exascale era

Results & Conclusions
EXASCALE CHALLENGES & Fortran 2015 Response

- Million-way concurrency with high levels of on-chip parallelism
  - events
  - collective subroutines
  - richer set of atomic subroutines
  - teams

- Higher failure rates
  - failed-image detection

- Expensive data movement (locality control)
  - teams

- Heterogeneous hardware on processor
  - events

An intrinsic module provides the derived type `event_type`, which encapsulates an `atomic_int_kind integer` component default-initialized to zero.

An image increments the event count on a remote image by executing `event post`.

The remote image obtains the post count by executing `event query`.

<table>
<thead>
<tr>
<th>Image Control</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_post</td>
<td>□ atomic_add 1</td>
</tr>
<tr>
<td>event_query</td>
<td>□ defines count</td>
</tr>
<tr>
<td>event_wait</td>
<td>□ atomic_add -1</td>
</tr>
</tbody>
</table>
Performance-oriented constraints:
- Query and wait must be local.
- Post and wait are disallowed in do concurrent constructs.

Pro tips:
- Overlap communication and computation.
- Wherever safety permits, query without waiting.
- Write a spin-query-work loop & build a logical mask describing the remaining work.
A set of images that can readily execute independently of other images
Collective Subroutines

Each non-failed image of the current team must invoke the collective.

After parallel calculation/communication, the result is placed on one or all images.

Optional arguments:.stat, errmsg, result_image, source_image

All collectives have **intent(inout)** argument

A holding the input data and may hold the result on return, depending on result_image

No implicit synchronization at beginning/end, which allows for overlap with other actions.

No image’s data is accessed before that image invokes the collective subroutine.
COLLECTIVE SUM

Definition

**CO_SUM** (A [, RESULT IMAGE, STAT, ERRMSG])

**Description**  
Sum elements on the current team of images.

**Arguments**

- **A** shall be of numeric type. It shall have the same type and type parameters on all images of the current team. It is an INTENT(INOUT) argument. If it is a scalar, the computed value is equal to a processor-dependent and image-dependent approximation to the sum of the values of A on all images of the current team. If it is an array it shall have the same shape on all images of the current team and each element of the computed value is equal to a processor-dependent and image-dependent approximation to the sum of all the corresponding elements of A on the images of the current team.

- **RESULT IMAGE** (optional) shall be a scalar of type integer. It is an INTENT(IN) argument. If it is present, it shall be present on all images of the current team, have the same value on all images of the current team, and that value shall be the image index of an image of the current team.

- **STAT** (optional) shall be a scalar of type default integer. It is an INTENT(OUT) argument.

- **ERRMSG** (optional) shall be a scalar of type default character. It is an INTENT(INOUT) argument.
COLLECTIVE SUM
Example

call co_sum(a)
Extensible Set of Collective Subroutines

call co_broadcast (a, source_image [, stat, errmsg])
call co_max (a [, result_image, stat, errmsg])
call co_min (a [, result_image, stat, errmsg])
call co_sum (a [, result_image, stat, errmsg])
call co_reduce (a, operator [, result_image, stat, errmsg])
~
Performance

User-Defined vs Intrinsic Collectives

NERSC Hopper: Xeon nodes on Cray XE6

KNL nodes on a Cray XC
Fault Tolerance
Why do we need it?

Exascale machines composed of thousands of nodes
Each node composed of thousand of cores

<table>
<thead>
<tr>
<th>MTBF - one node</th>
<th>1 year</th>
<th>10 years</th>
<th>120 years</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTBF - one million nodes</td>
<td>30 seconds</td>
<td>5 minutes</td>
<td>1 hour</td>
</tr>
</tbody>
</table>

Checkpointing/Restart will not be feasible with such high failure rates.
FORTRAN 2015
Failed-Image Detection

- FAIL IMAGE (simulates a failures)
- IMAGE_STATUS (checks the status of a specific image)
- FAILED_IMAGES (provides the list of failed images)
- Coarray operations may fail. STAT= attribute used to check correct behavior.
Failure Detection

```fortran
use iso_fortran_env, only : STAT_FAILED_IMAGE
integer :: status
sync all(stat==status)
if (status==STAT_FAILED_IMAGE) call fault_tolerant_algorithm()
```

Coarray ICAR
Xeon vs KNL

Scaling comparison

Run time comparison

Fraction of Ideal Speedup

Number of Processes

Simulation time [s]

Number of Processes
Conclusions

- Fortran 2008 is a PGAS language that supports backwards compatible SPMD programming of new applications as well as parallelization of legacy apps.

- Fortran 2015 addresses several exascale challenges.

- Early lessons:
  - Let the language do its job: collectives
  - Balance efficiency gains of KNL against cost/core and raw performance.