Argonne NATIONAL LABORATORY

... for a brighter future

U.S. Department of Energy

UChicago Argonne LLC

Office of Science U.S. DEPARTMENT OF ENERGY

A U.S. Department of Energy laboratory managed by UChicago Argonne, LLC

MPITypes:
Processing MPI Datatypes Outside MPI

Rob Ross[1], Rob Latham[1], William Gropp[2], Ewing Lusk[1], Rajeev Thakur[1]

[1] Mathematics and Computer Science Division
Argonne National Laboratory
{rross, robl, lusk, thakur}@mcs.anl.gov

[2] Computer Science Department
University of Illinois at Urbana-Champaign
wgropp@illinois.edu

Motivation – Libraries and MPI

- Libraries for parallel computing play a critical role in improving the performance of codes and productivity of application writers (e.g., MPI libraries, ScaLAPACK, PETSc, HDF5)
- MPI communicators, requests, attributes, and datatypes are extremely useful constructs for building parallel libraries
- Some improvements could be made in generalized requests
  - R. Latham, W. Gropp, R. Ross, and R. Thakur, "Extending the MPI-2 Generalized Request Interface," Proc. of EuroPVM/MPI 2007.

The biggest missing piece for parallel libraries that build on MPI is a system for efficient, custom manipulation of data described by MPI datatypes.

Custom MPI Datatype Processing

We need more than MPI_Pack and MPI_Unpack.

- ROMIO – data sieving and two-phase optimizations
  - Operates on portions of types (partial processing)
  - Combines multiple types together
  - Types describing file regions, not just memory regions
- Parallel netCDF
  - May operate on portions of types
  - Byteswaps data on some systems
  - Converts data from one representation to another

The MPITypes Library

MPITypes is a portable, open source library for processing MPI datatypes in libraries and applications.

- Based on MPICH2 datatype processing component
- Built-in functions for packing, unpacking, and flattening
- Toolkit for building custom type processing routines

- Uses only MPI-2 functionality for accessing datatype information and caching data:
  - Datatype envelope and contents functions
  - Attributes on communicators and datatypes

Outline of Talk

- Motivation
- **Datatype processing in MPICH2**
  - **Dataloop representation of MPI datatypes**
  - **Segments, leaf functions, and traversing dataloop trees**
- MPITypes
  - Summary
  - Basic functionality
  - Building functions with MPITypes
- Performance evaluation
- Related work
- Concluding remarks

Datatype Processing (from MPICH2)

- **Uses a simplified representation, called dataloops**
- Five basic dataloop node types with increasing complexity
  - **contig** - blocklength
  - **vector** - count, blocklength, stride
  - **blockindexed** - count, blocklength, offset array
  - **indexed** - count, blocklength array, offset array
  - **struct** - count, blocklength array, offset array
- Nodes are used to build trees for more complex types
- Dataloop tree is processed nonrecursively, with state held in a data structure called a **segment**
- **Leaf functions** define the operation performed on data

Simple Dataloop Example

- MPI Vector datatype:

```
MPI_Vector, cnt = 2, blklen = 2, str = 4
MPI_INT
```

- Dataloop representation (just one leaf node, 8-byte integers):

```
DLP_Vector, cnt = 2, blklen = 2, str = 32,
        el_sz = 8, el_ext = 8, el_type = MPI_INT
```

Complex Dataloop Example (FLASH)

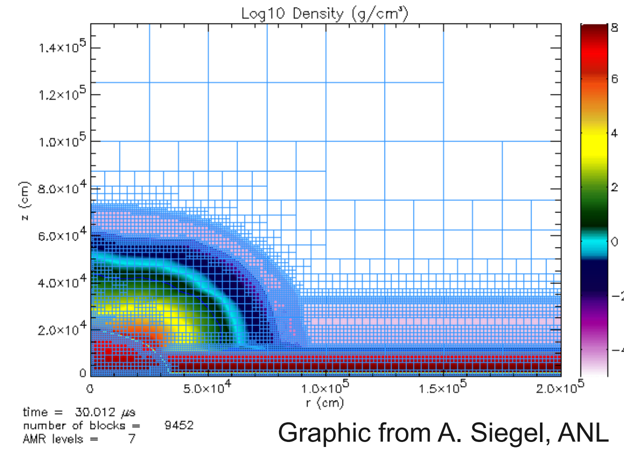
- Dataloop representation:

DLP_Vector, cnt = 80, blklen = 1, str = 768432,
el_sz = 4096, el_ext = 366920

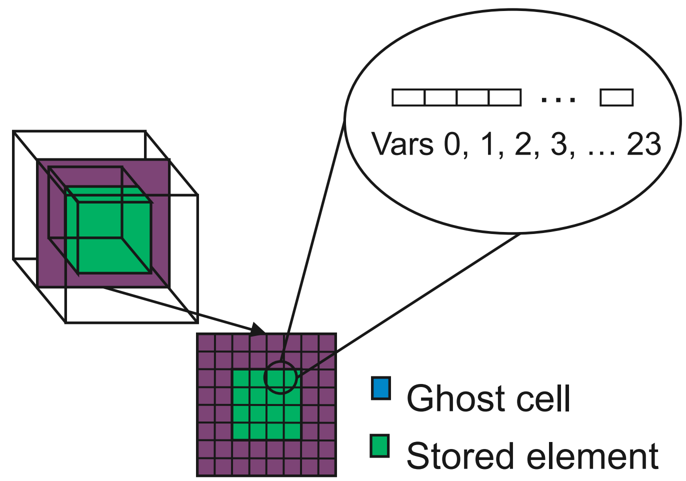
DLP_Vector, cnt = 8, blklen = 1, str = 49152,
el_sz = 512, el_ext = 22856

DLP_Vector, cnt = 8, blklen = 1, str = 3072,
el_sz = 64, el_ext = 1352

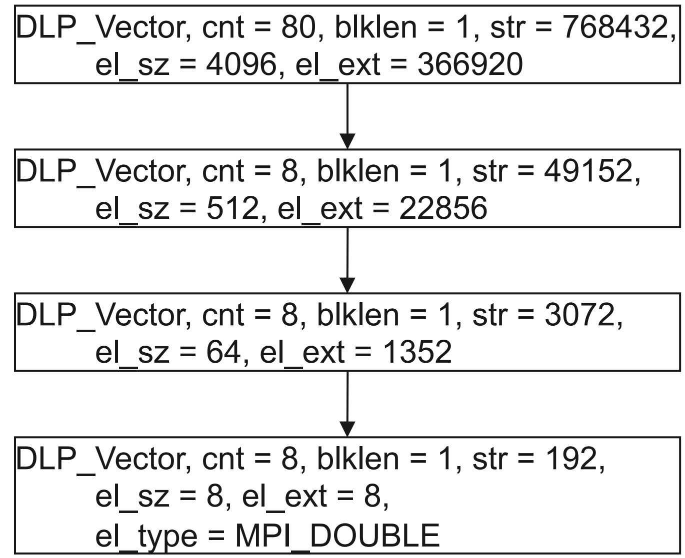
DLP_Vector, cnt = 8, blklen = 1, str = 192,
el_sz = 8, el_ext = 8,
el_type = MPI_DOUBLE



Graphic from A. Siegel, ANL



Traversing Dataloops




```
if (not a leaf node) {
    while (not done with this dataloop node) {
        update segment with new position
        push current dataloop state onto stack
        process next dataloop in tree
        decrement count/blklen in segment
    }
    pop dataloop off the stack and resume processing
} else /* leaf */ {
   if (leaf type is index && have index leaf fn) call index leaf fn
   if (leaf type is vector && have vector leaf fn) call vector leaf fn
   …
   else call contig leaf fn
   pop dataloop off stack in segment and resume processing
}
```

Outline of Talk

- Motivation
- Datatype processing in MPICH2
  - Dataloop representation of MPI datatypes
  - Segments, leaf functions, and traversing dataloop trees
- **MPITypes**
  - **Summary**
  - **Basic functionality**
  - **Building functions with MPITypes**
- Performance evaluation
- Related work
- Concluding remarks

The MPITypes Library

- Datatype processing extracted from MPICH2
  - Symbols renamed to avoid conflicts
  - Builds using mpicc of local MPI implementation
- MPITypes functions build a dataloop representation on demand
  - MPI_Type_get_envelope and MPI_Type_get_contents used to extract datatype parameters
  - Attribute on datatype is used to associate the dataloop representation with the type, so it is only built once
- Segment structure used to maintain state during processing (as in MPICH2)
- User-defined processing functions allow custom behavior

Basic MPITypes Operators

int **MPIT_Type_memcpy**(typebuf, count, type, streambuf, direction, start, end);

– *Like MPI_Pack/MPI_Unpack, but allows partial processing*

int **MPIT_Type_blockct**(count, type, start, end, blockct);

– *Provides a count of the number of contiguous regions in a portion of a [count, datatype] tuple*

int **MPIT_Type_flatten**(typebuf, count, type, start, end, disps, blocklens, count);

– *Generates a list of offsets and lengths for a portion of a [buffer, count, datatype] tuple*

MPITypes Toolkit Functions

MPIT_Segment ***MPIT_Segment_alloc**();

int **MPIT_Segment_init**(buf, count, type, segment, flag);

int **MPIT_Segment_free**(segment);

- *Allocate, initialize, and free the data structure used to track progress while processing a [count, datatype] tuple*

int **MPIT_Segment_manipulate**(segment, start, end, (*contigfn) (...), (*vectorfn) (...), (*blkidxfn) (...), (*indexfn) (...), (*sizefn) (el_type), params);

- *Traverse the datatype representation, executing specified leaf functions as leaves are encountered*

Basic MPITypes operators are built with these functions.

MPITypes memcpy Implementation (1/2)

```
typedef struct MPIT_memcpy_params_s {
  int direction; char *packbuf, *userbuf;
} MPIT_memcpy_params;
```

Data structure used to hold parameters relevant to custom processing

```
int MPIT_Leaf_contig_memcpy(MPI_Aint *blocks_p, MPI_Type el_type,
    MPI_Aint dtype_pos, void *unused, void *v_paramp)
{
  MPI_Aint size, el_size; MPIT_memcpy_params *paramp = v_paramp;
  MPI_Type_size(el_type, &el_size); size = *blocks_p * el_size;

  if (paramp->direction == MPIT_MEMCPY_TO_USERBUF)
    memcpy(paramp->userbuf + dtype_pos, paramp->packbuf, size);
  else
    memcpy(paramp->packbuf, paramp->userbuf + dtype_pos, size);

  paramp->packbuf += size; return 0;
}
```

MPITypes memcpy Implementation (2/2)

```
int MPIT_Type_memcpy(void *typebuf, int count, MPI_Datatype type,
    void *streambuf, int direction, MPI_Aint start, MPI_Aint *end)
{
  MPIT_Segment *segp;
  MPIT_memcpy_params params;

  segp = MPIT_Segment_alloc();
  MPIT_Segment_init(NULL, count, type, segp, 0);

  params.userbuf   = typebuf;
  params.packbuf   = packbuf;
  params.direction = direction;

  MPIT_Segment_manipulate(segp, start, end,
  MPIT_Leaf_contig_memcpy, NULL, NULL, NULL, NULL, &params);

  MPIT_Segment_free(segp); return MPI_SUCCESS;
}
```

Optional vector, blockindexed, and indexed leaf functions would go here.

Outline of Talk

- Motivation
- Datatype processing in MPICH2
  - Dataloop representation of MPI datatypes
  - Segments, leaf functions, and traversing dataloop trees
- MPITypes
  - Summary
  - Basic functionality
  - Building functions with MPITypes
- **Performance evaluation**
- Related work
- Concluding remarks

Performance Evaluation

- Goals
  - Show that MPITypes performs on par with current MPI implementations for a variety of datatypes
  - Establish feasibility of using MPITypes to implement relevant datatype operations
  - Quantify benefit of using MPITypes in the context of a parallel library
- Evaluation
  - MPITypes memcpy implementation
  - MPITypes implementation of *transpacking*
  - Custom function for encoding data prior to I/O in Parallel netCDF

Test Environment

- 2.66 GHz Intel Xeon node
  - 8 cores
  - 16 Gbytes of main memory
  - Little endian
- Linux 2.6.27
- GNU C compiler version 4.2.4
- MPICH2 version 1.0.8p1
  - Compiled with "--enable-fast=O3"
- Open MPI version 1.3.1
  - Compiled with "CFLAGS=-O3 --disable-heterogeneous --enable-shared=no --enable-static --with-mpi-param-check=no"

Comparing MPI_Pack/MPI_Unpack, MPIT_Type_memcpy, and manual copy rates

MPITypes performance is essentially identical to MPI implementations.

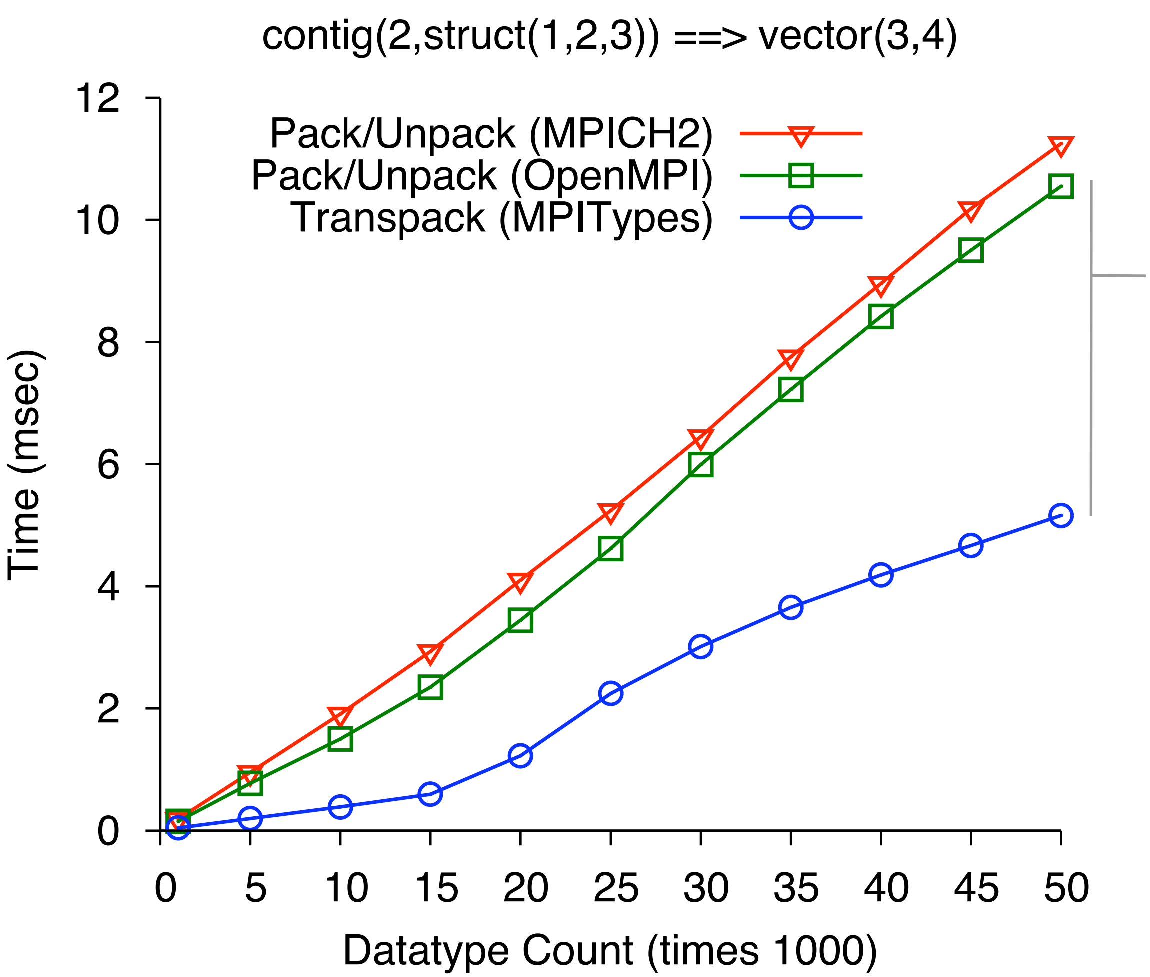
- Copy into and then back out of a contiguous buffer, many times (provides opportunity to verify correctness)

Test	MPICH2 (MB/sec)	Open MPI (MB/sec)	MPITypes (MB/sec)	Manual (MB/sec)	Size (MB)	Extent (MB)
Contig	4152.07	4157.69	**4149.13**	2650.81	8.00	8.00
Vector	1776.81	1680.23	**1777.04**	1777.60	8.00	16.00
Indexed	1120.59	967.69	1123.97	**1575.41**	4.00	8.00
XY Face	17564.43	18143.63	**17520.11**	16423.59	0.50	0.50
XZ Face	4004.26	4346.81	**3975.23**	3942.41	0.50	127.50
YZ Face	153.89	154.19	**153.88**	153.96	0.50	127.99

Transpacking

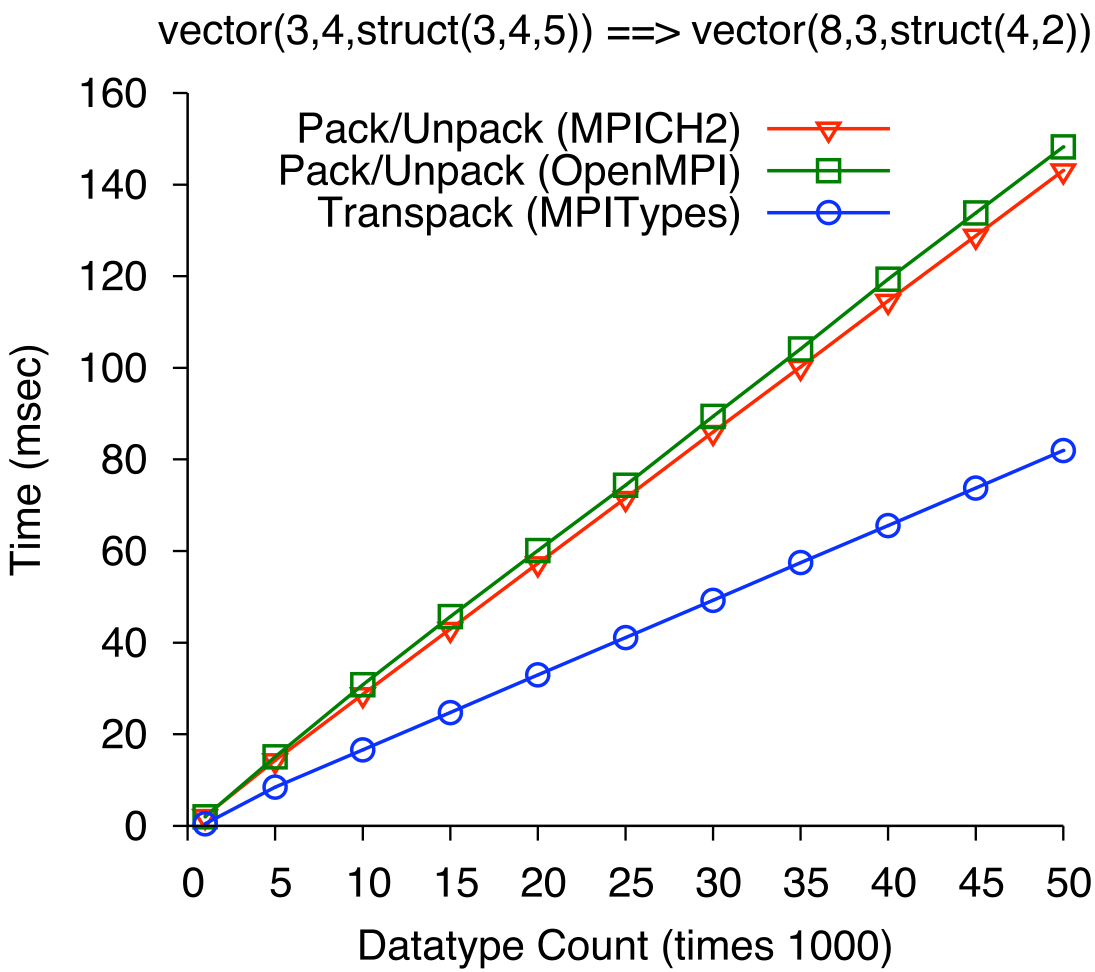
- **Transpacking is a solution to the *typed copy* problem – moving data from one datatype representation to another.**
  - Simple solution is to MPI_Pack and then MPI_Unpack, but this requires two copies and a large intermediate buffer
  - Partial processing reduces memory requirement
  - Better solution is to directly copy from one representation to another
- Quite elegant solutions to this have been proposed previously (see Mir and Träff)
- We implemented a less elegant solution using MPITypes (~200 lines)
  - Best for like-sized types with a relatively small number of contiguous regions in a single instance
  - Generates a "template" for how to copy between a single instance of each, iterates on this.

Comparing MPITypes Implementation of Transpack to MPI_Pack/MPI_Unpack (1/2)



55% reduction in time over MPI_Pack/MPI_Unpack

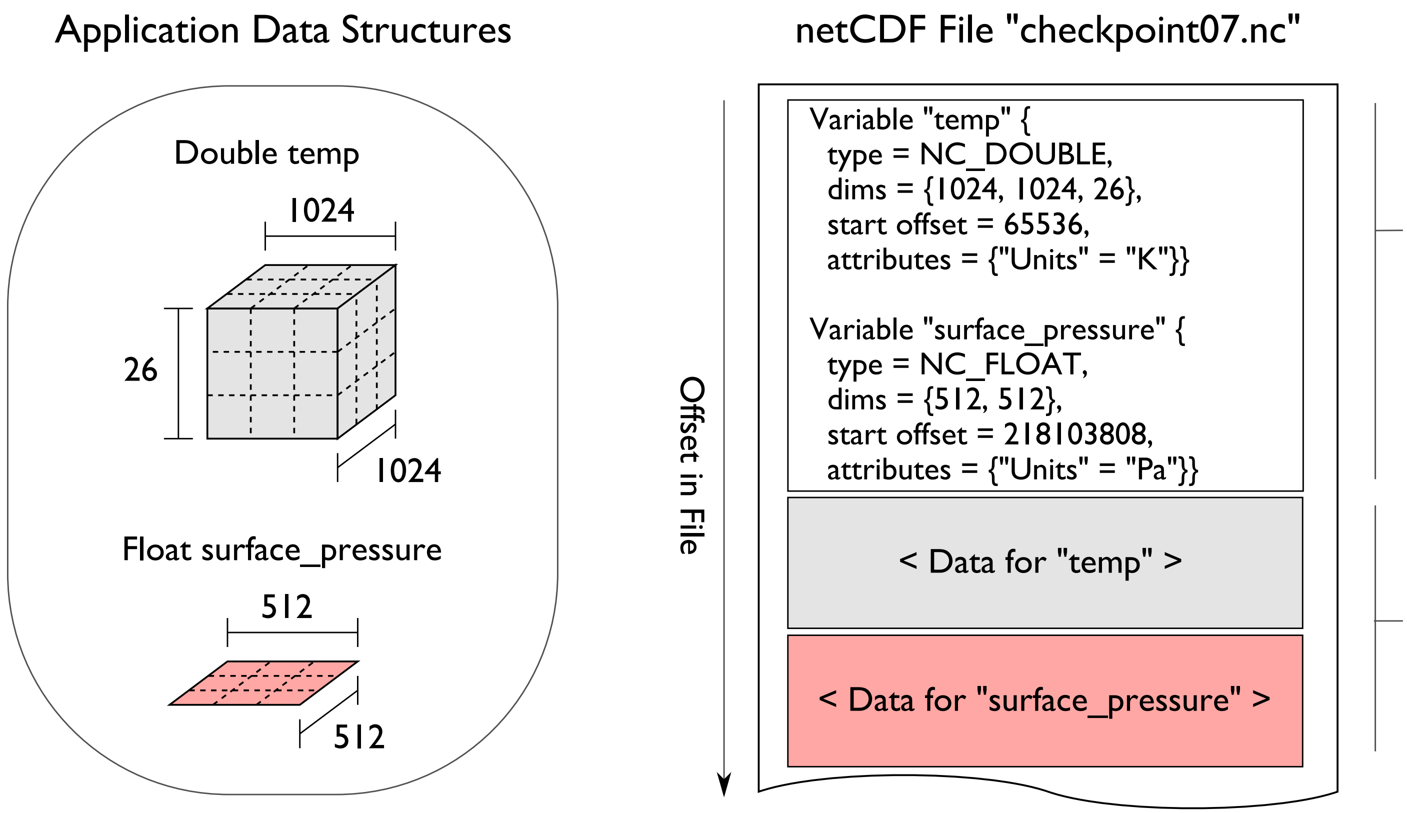
Comparing MPITypes Implementation of Transpack to MPI_Pack/MPI_Unpack (2/2)



43% reduction in time over MPI_Pack/MPI_Unpack

The Parallel netCDF I/O Library

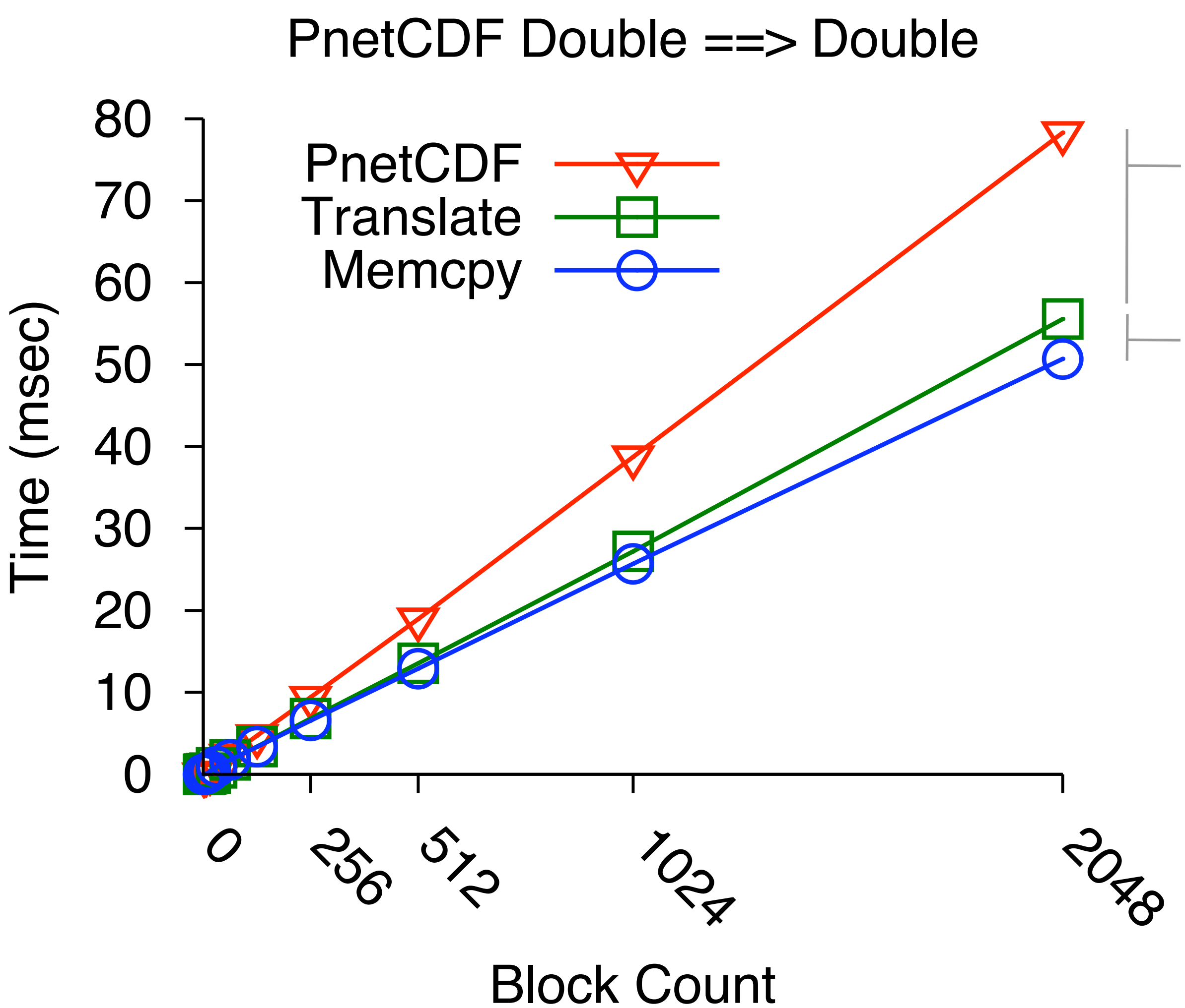
Parallel netCDF (PnetCDF) provides a convenient, efficient way of storing scientific data in a portable file format.



netCDF header describes the contents of the file: typed, multi-dimensional variables and attributes on variables or the dataset itself.

Data for variables is stored in contiguous blocks, encoded in a portable binary format according to the variable's type.

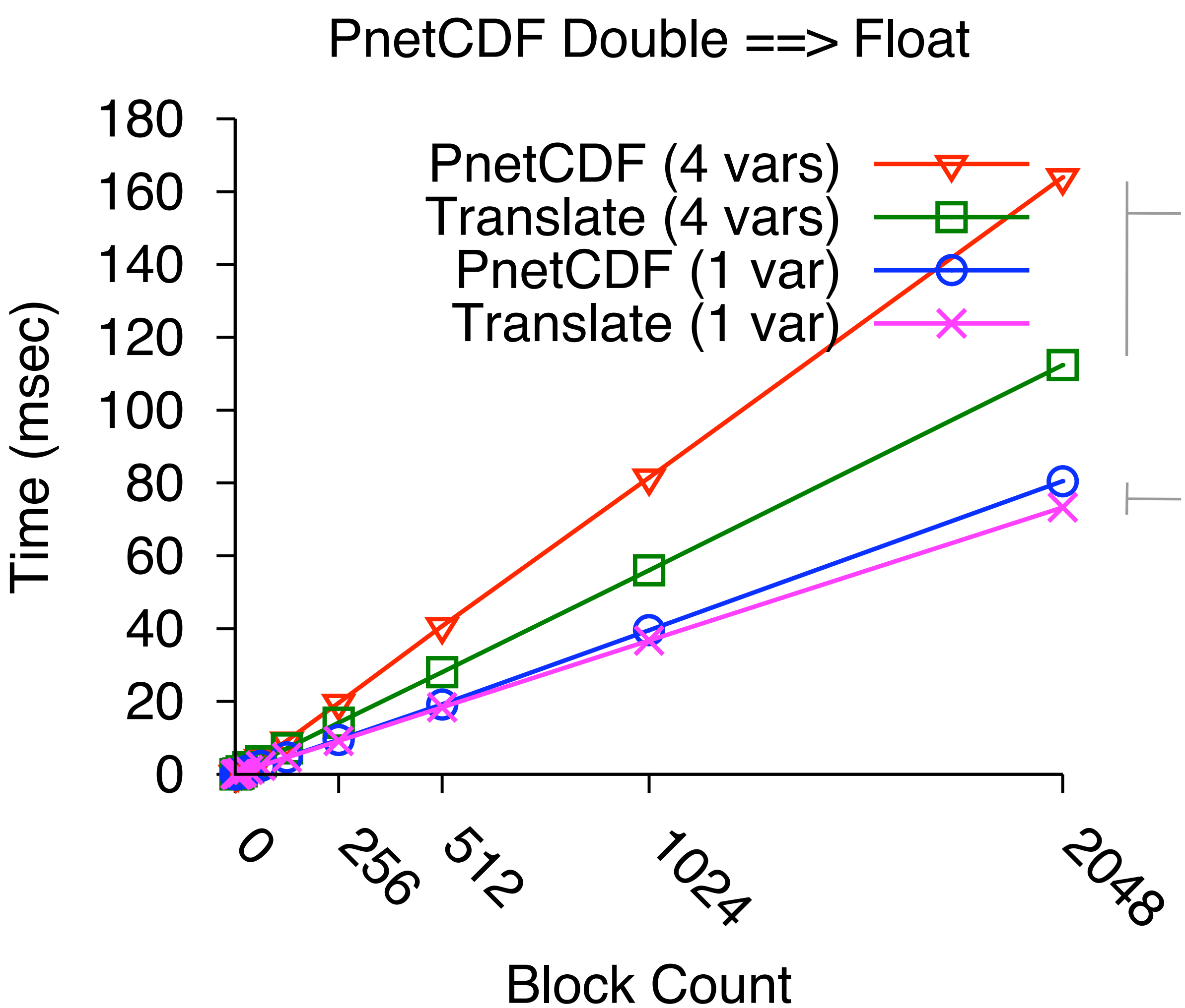
PnetCDF: Byteswapping in FLASH Case



29% reduction in time over the original PnetCDF approach

At most 9.5% gap between MPITypes copy and byteswap and only copying the data (no byteswap)

PnetCDF: Data Conversion in FLASH Case



31% reduction in time over the original PnetCDF approach when four types are adjacent to one another

9% reduction in time between MPITypes and PnetCDF approach, when no elements are adjacent to one another

Related Work in Datatype Processing

- J. Träff, R. Hempel, H. Ritzdoff, and F. Zimmermann, "Flattening on the fly: Efficient handling of MPI derived datatypes," In Proceedings of EuroPVM/MPI 1999.
- R. Ross, N. Miller, and W. Gropp, "Implementing fast and reusable datatype processing," In Proceedings of EuroPVM/MPI 2003.
- J. Worringen, J. Träff, and H. Ritzdorf, "Improving generic noncontiguous file access for MPI-IO," In Proceedings of EuroPVM/MPI 2003.
- F. Mir and J. Träff, "Constructing MPI input-output datatypes for efficient transpacking," In Proceedings of EuroPVM/MPI 2008.
- F. Mir and J. Träff, "Exploiting efficient transpacking for one-sided communication and MPI-IO," In Proceedings of EuroPVM/MPI 2009.

Concluding Remarks

- MPITypes provides a high performance, customizable implementation of datatype processing
  - Hides most of the complexity of efficiently manipulating MPI datatypes
  - Retains performance characteristics of MPI implementations
- Easy to use and incorporate into new and existing parallel libraries and applications
  - Uses MPICH2 source code license (BSD-like)
- Source code now available
  - See **http://www.mcs.anl.gov/mpitypes**
- Perhaps worth considering incorporating similar functionality into future MPI standards?