... for a brighter future

Automatic Memory Optimizations for Improving MPI Derived Datatype Performance

Surendra Byna[†] Xian-He Sun[†] Rajeev Thakur[‡] William Gropp[‡]

[†]Department of Computer Science, Illinois Institute of Technology, USA

[‡]Math. and Comp. Science Division, Argonne National Laboratory, USA

A U.S. Department of Energy laboratory
managed by The University of Chicago

Review of MPI Datatypes

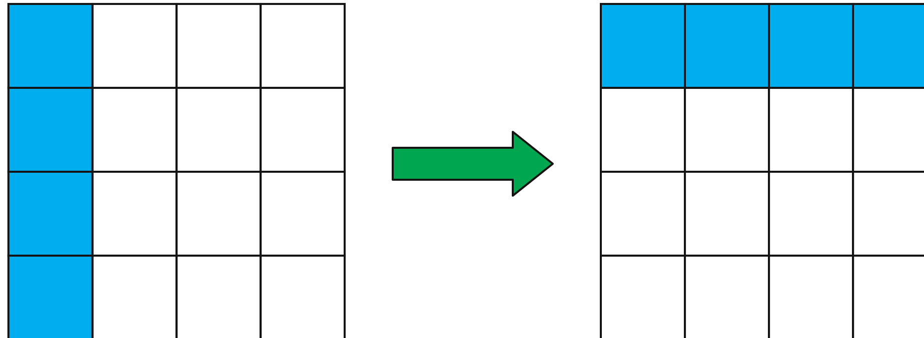
- In MPI, the data to sent or received is described by a triplet (address, count, datatype)
  - `MPI_Send(buf, count, datatype, dest, tag, comm)`
- The datatype can either be a basic datatype, such as **`MPI_INT`**, **`MPI_FLOAT`**, or
- One can recursively define *derived datatypes* comprising:
  - a contiguous array of datatypes
  - a uniformly strided array of blocks of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
  - distributed arrays and subarrays

Argonne NATIONAL LABORATORY ILLINOIS INSTITUTE OF TECHNOLOGY

Why Datatypes?

- To support communication between processes on machines with very different memory representations and lengths of elementary datatypes
- Specifying noncontiguous layout of data in memory
  - can reduce/optimize memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather) when available
- Specifying noncontiguous layout of data in a file
  - can reduce system calls and physical disk I/O

Example of Noncontiguity: Matrix Transpose



Problems with Derived Datatypes

- Many MPI implementations perform poorly with derived datatypes
- Users resort to their own implementations of packing the data into a contiguous buffer and then calling MPI_Send
- Such usage clearly defeats the purpose of having derived datatypes in the MPI Standard
- Many advanced compilers are also not able to optimize noncontiguous data accesses in the user packing code

Improving the Performance of Derived Datatypes

- Goal: Improve the performance of derived datatypes to a level better than the average user can achieve with simple manual packing
- Our work focuses on using memory copying optimizations to do the buffer packing/unpacking efficiently based on knowledge of the memory architecture

- Other researchers have investigated
  - Using data structures that allow a stack-based approach to parsing a datatype, rather than making recursive function calls (Traff ’99, Ross-Miller-Gropp ’03)
  - Taking advantage of the features in InfiniBand to overlap packing & unpacking a message with network communication (Wu et al.)

Memory Performance Optimization

- Optimize the performance based on the data access pattern and the memory architecture of the machine
- Step 1 (in **`MPI_Type_commit`**)
  - Retrieve the data access pattern of a derived datatype from user's definition
  - Verify whether performance improvement is possible
  - Find memory optimization parameters
- Step 2 (in **`MPI_Send/Recv`**)
  - Call optimized pack/unpack templates and pass the optimization parameters
- Optimization is performed automatically within the source code

Retrieving Data Access Pattern

- We classify access patterns into combinations of contig/noncontig, fixed/variable block sizes, fixed/variable strides
- User's definition of a derived datatype contains data access pattern information
  - ```
    MPI_Type_vector(int count, int blocklength,
                    int stride,
                    MPI_Datatype oldtype,
                    MPI_Datatype *newtype);
    ```
- We use datatype decoder functions (**`MPI_Type_get_envelope`** and **`MPI_Type_get_contents`**) to retrieve the pattern
    ```

Is Performance Improvement Possible?

- Use a series of heuristics to determine whether a datatype is optimizable or not
  - Is the datatype contiguous or noncontiguous?
  - Is data size larger than cache size?
  - Is cache and TLB reuse too low?
- Set a flag (**`is_optimizable`**) if the datatype is deemed optimizable

Optimization Parameters

- Various memory optimization methods
  - cache blocking, loop unrolling, array-padding optimizations, and software-level prefetching
- Select optimization parameters with the goal of reducing cache misses
  - Cache blocking parameters: block size fits into the cache memory and improve cache and TLB reuse
  - Array padding to avoid cache thrashing
  - Adjust *prefetch distance* to make sure that data is prefetched in time
- **`MPI_Send/Recv`** functions call optimized templates that are passed these parameters

Experiments

Benchmarks

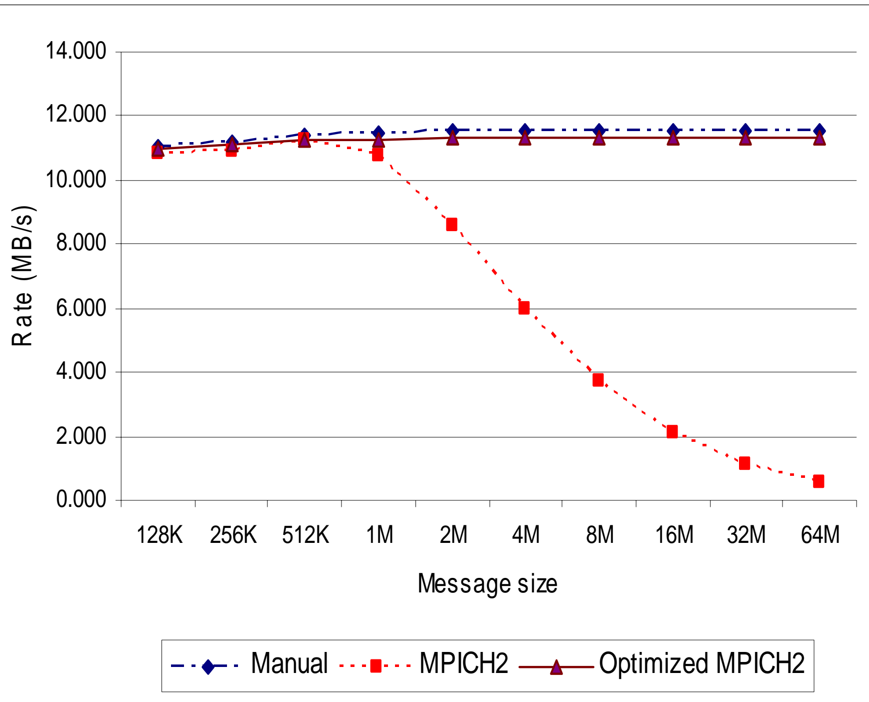
- Simple derived datatypes – vector and indexed datatypes from SKaMPI benchmark
- Nested derived datatypes - 3D array is stored in row-major order, vector of YZ planes in an array
- NAS benchmarks – MG, Matrix Transpose from FFT

Machine Architectures

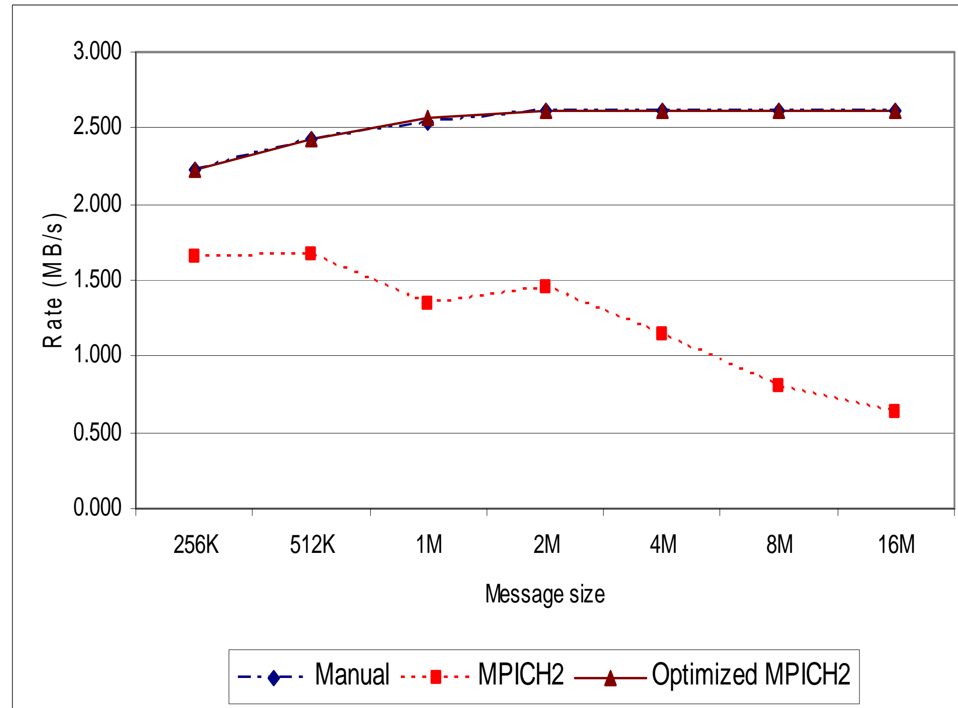
- Argonne's 350 node *jazz* cluster
  - 2.4 GHz Pentium-4 processor with 1 GB of memory
  - 512KB L2 cache, 64 byte line, 8-way set associative
  - TLB of 128 entries, and a page size of 4 KB
  - Interconnect used: Fast Ethernet

- IIT's 84 node *sunwulf* cluster
  - 500MHz UltraSparc-IIe CPU
  - 8MB L2 cache, 64 byte line, 4-way set associative
  - TLB of 48 entries with 4 KB page size
  - Interconnect: Gigabit Ethernet

Vector and Indexed Datatypes (jazz cluster)

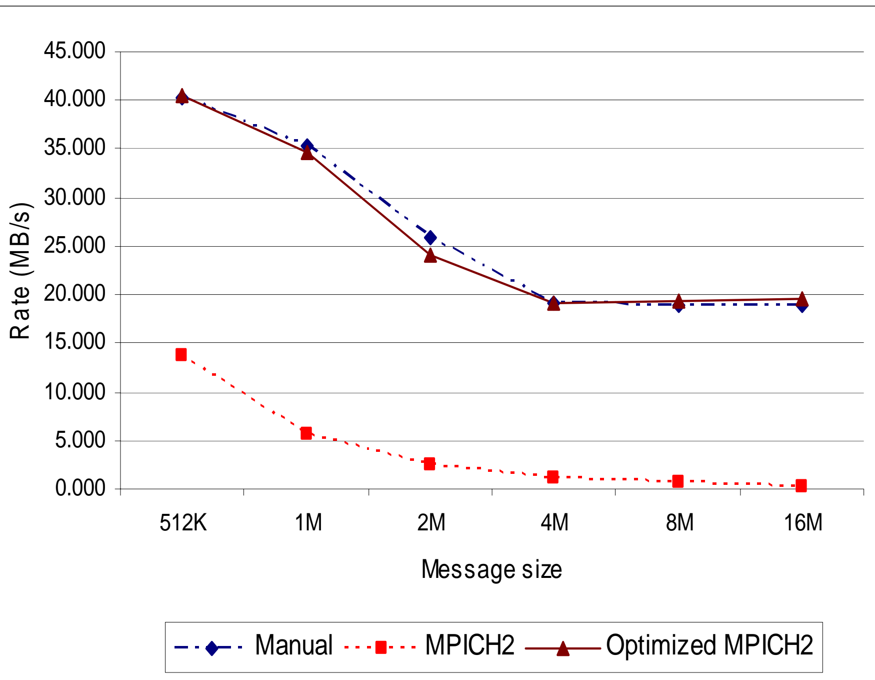


Vector

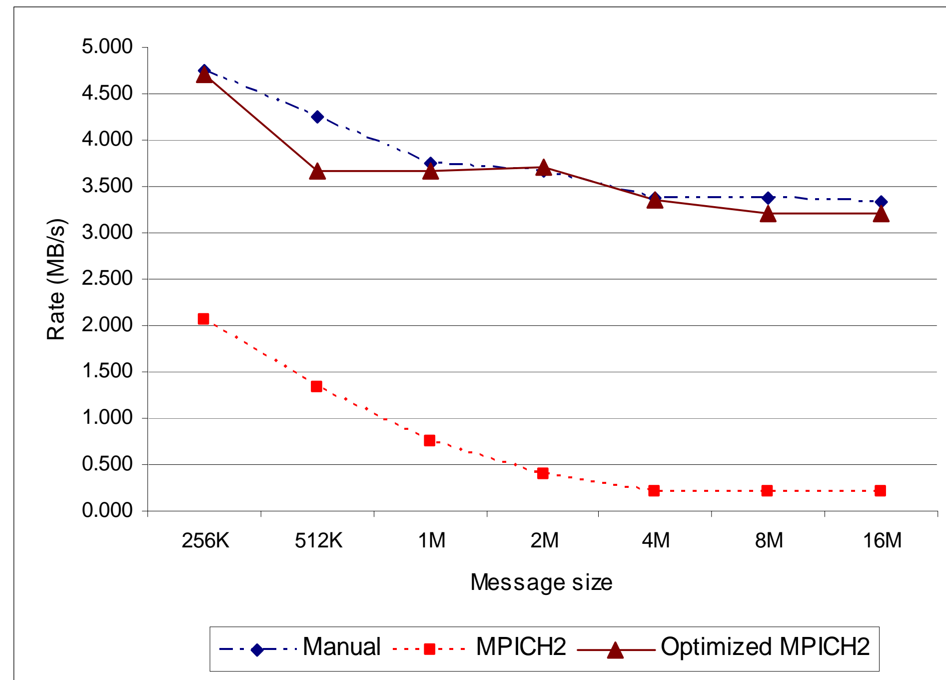


Indexed

Vector and Indexed Datatypes (sunwulf cluster)

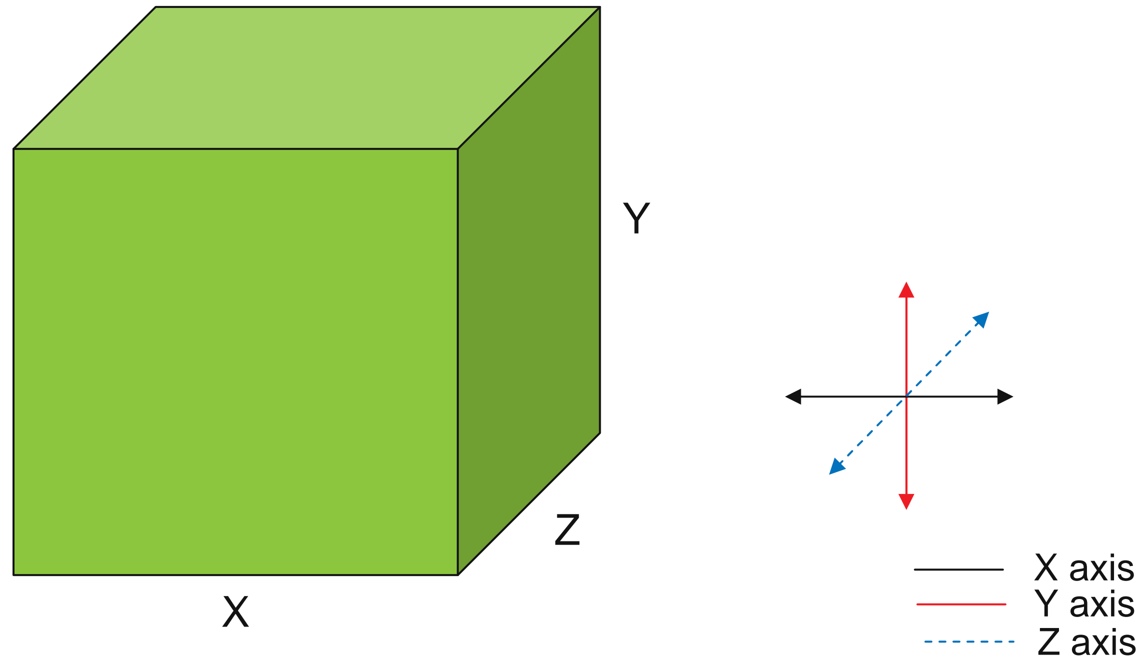


Vector



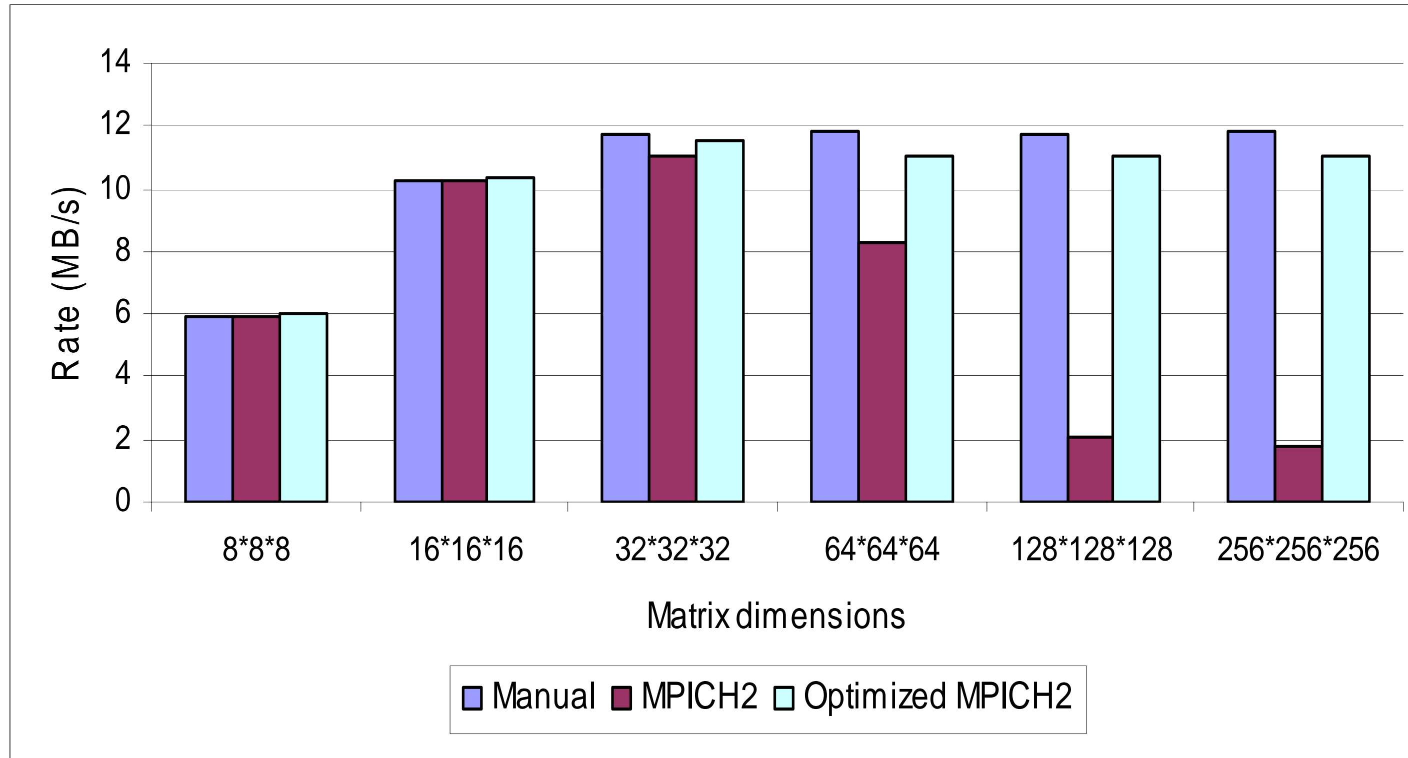
Indexed

3D Cube Test

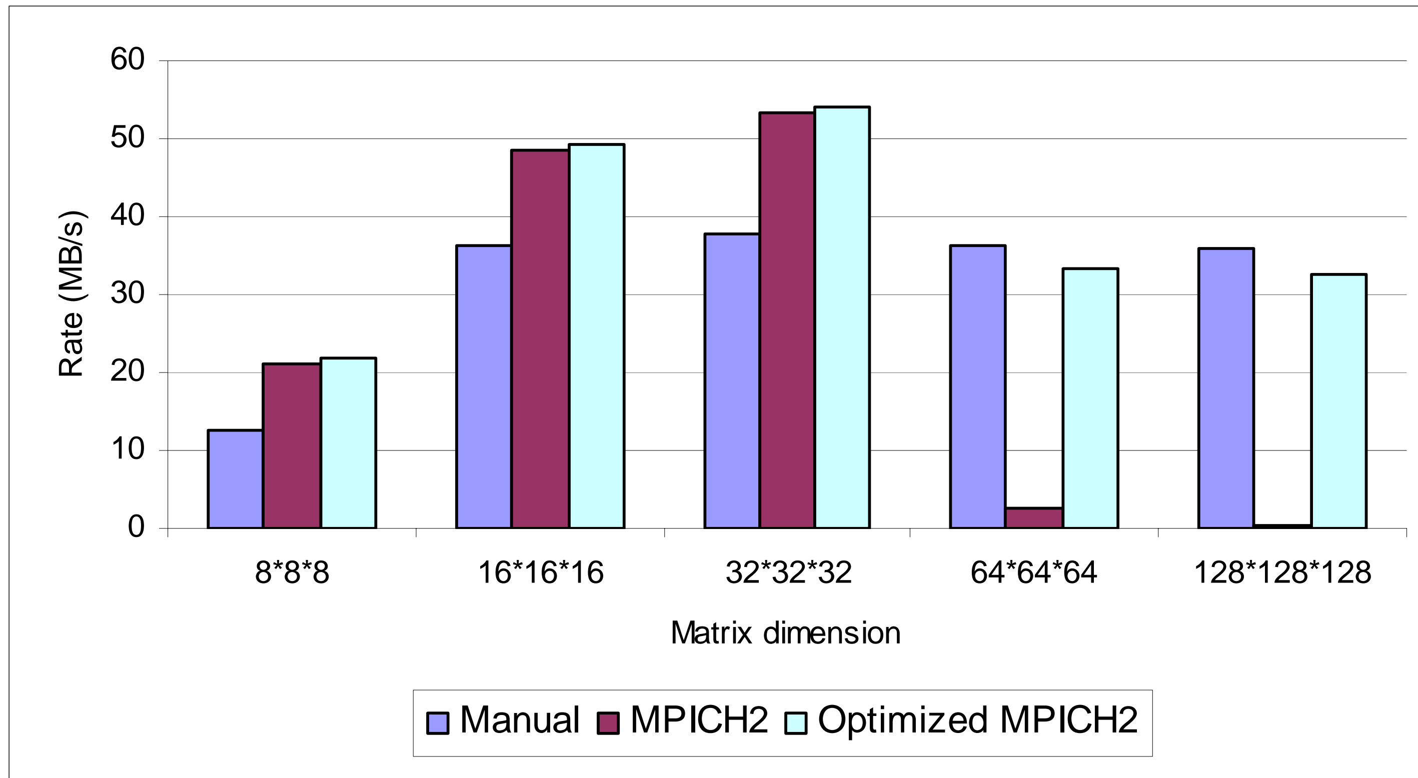


Accessing the cube along YZ slices is noncontiguous and has poor locality when the size of the YZ face is more than the cache or TLB sizes

3D Cube on Jazz Cluster



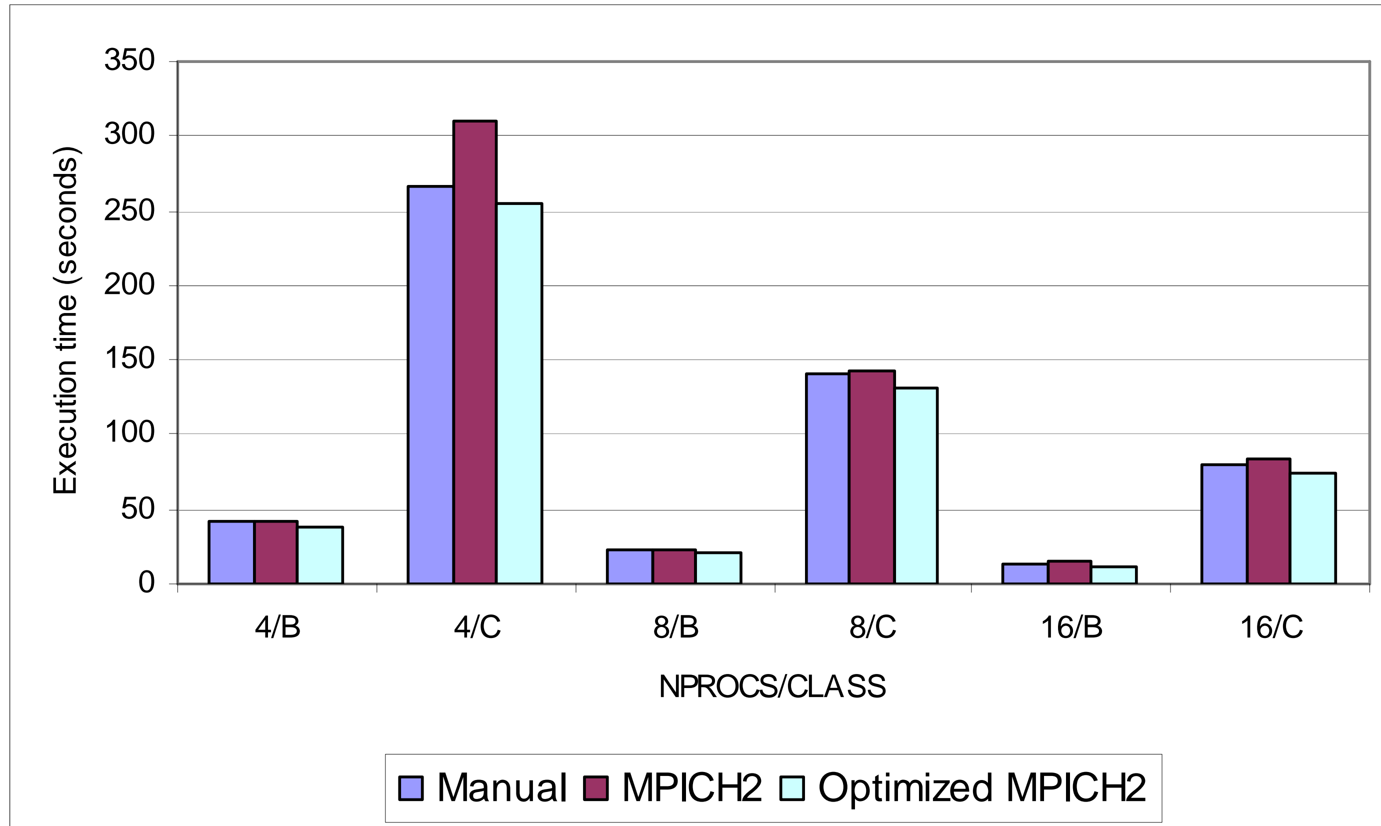
3D Cube on Sunwulf Cluster



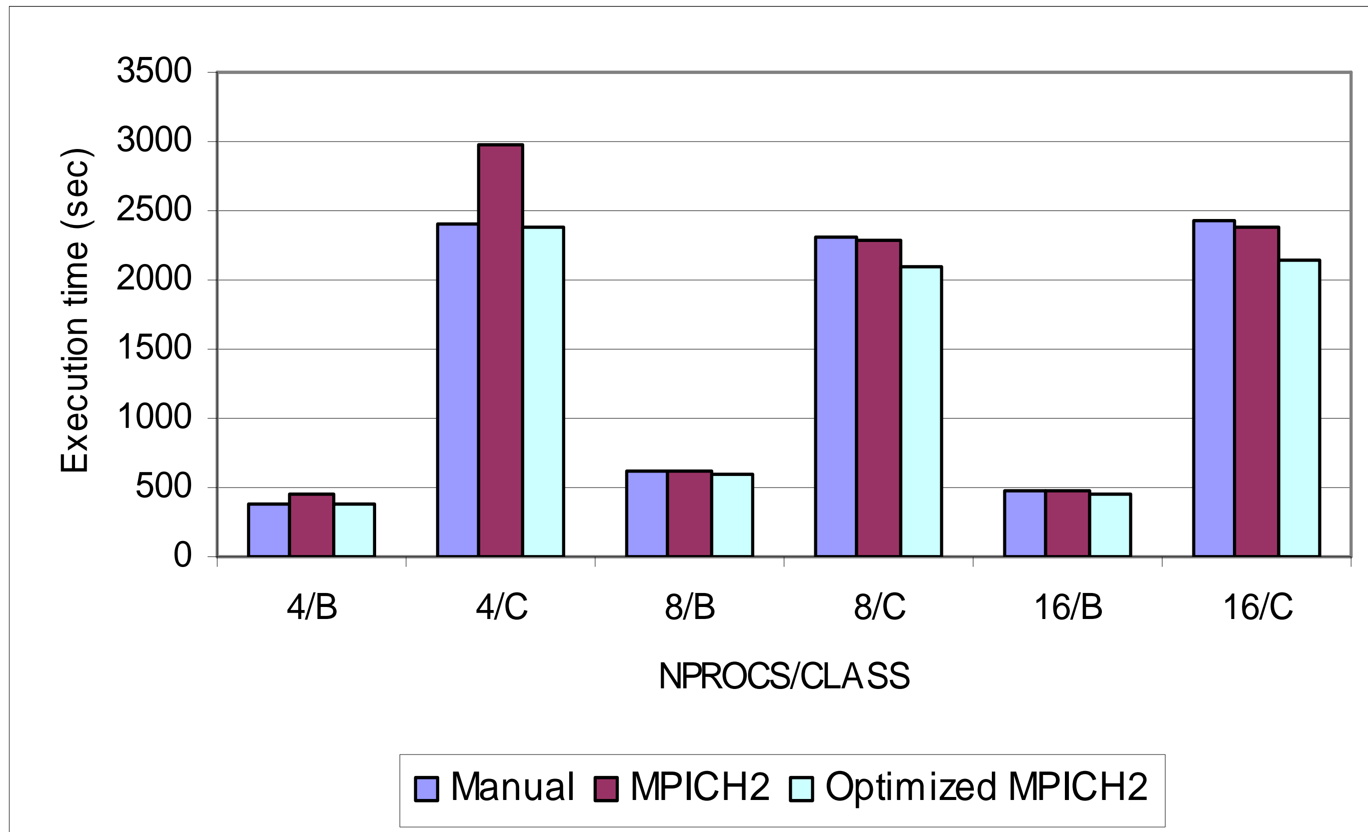
Argonne NATIONAL LABORATORY

ILLINOIS INSTITUTE OF TECHNOLOGY

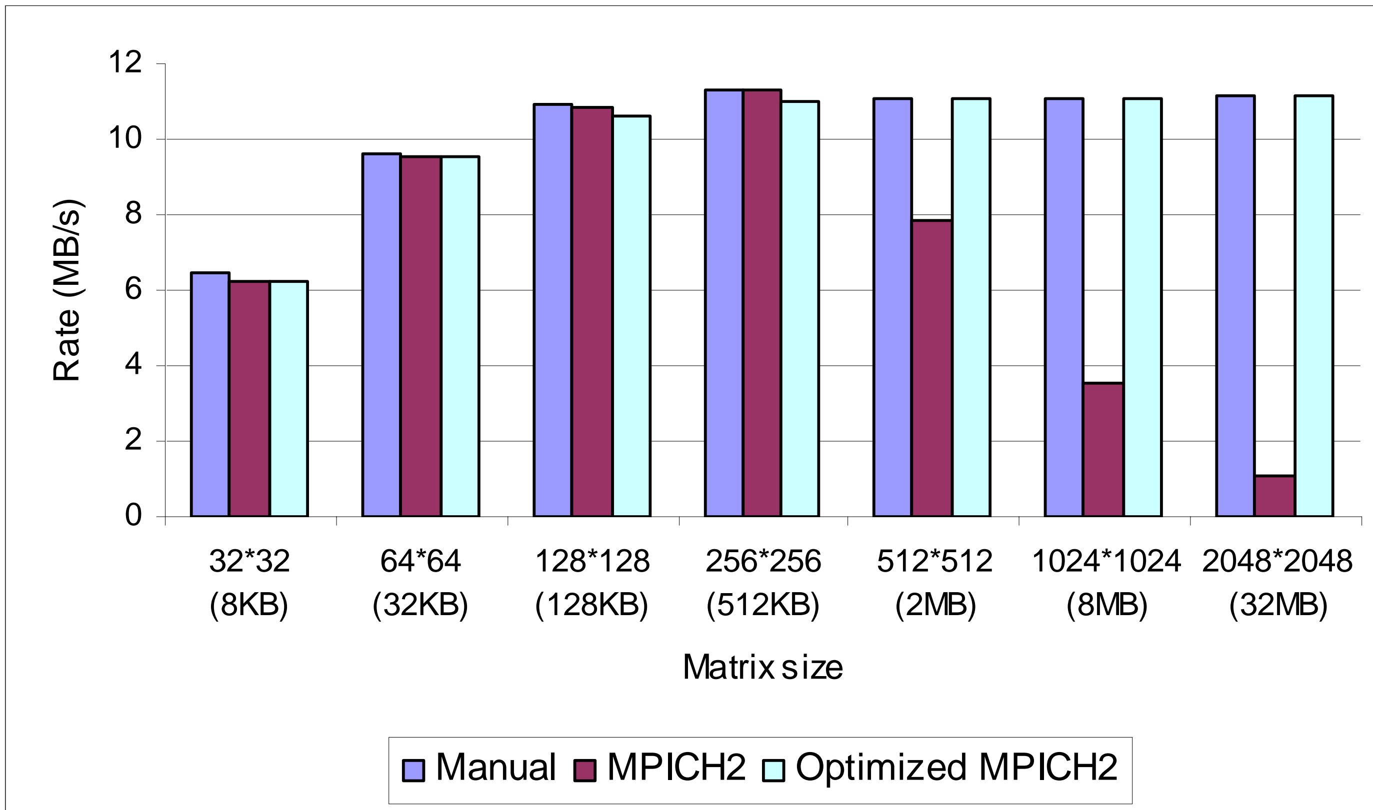
NAS MG Benchmark on Jazz Cluster



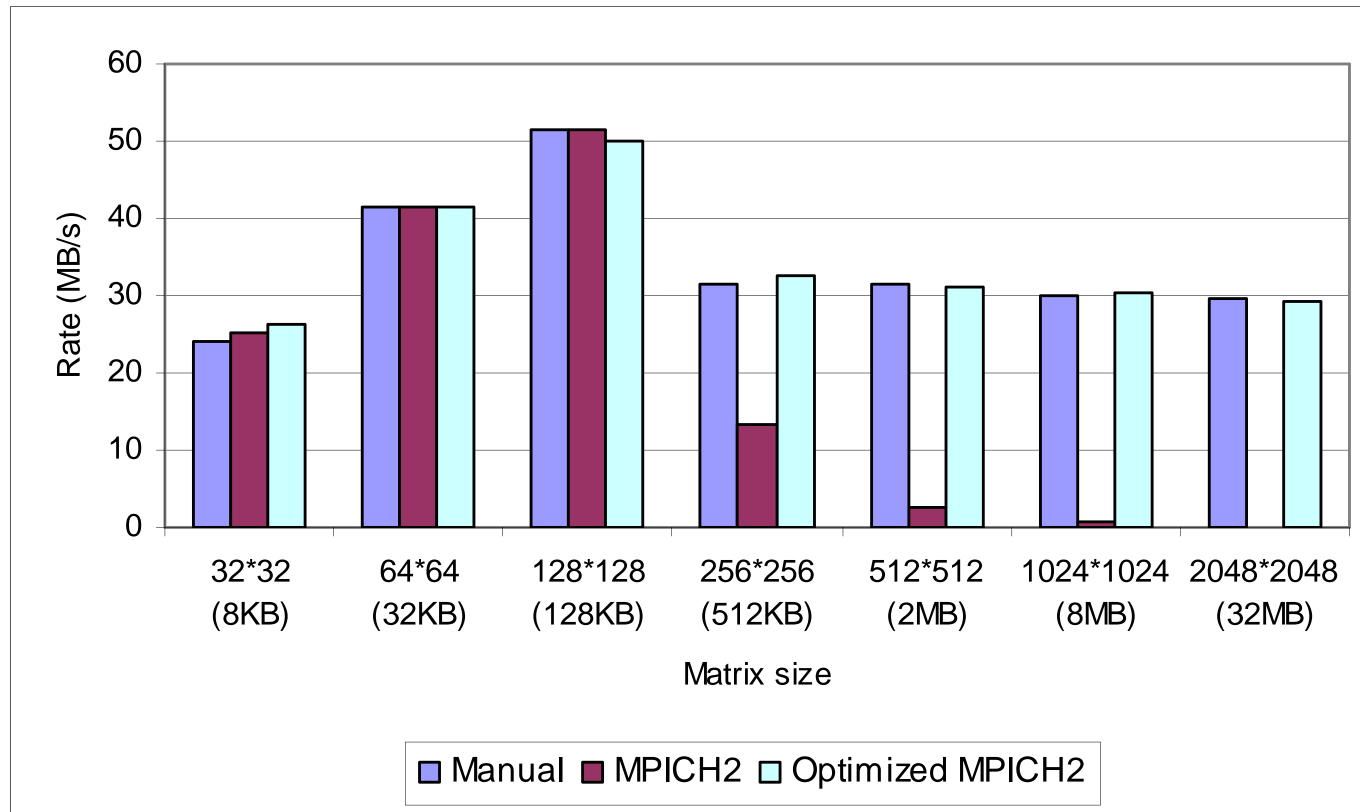
NAS MG Benchmark on Sunwulf Cluster



Matrix Transpose on Jazz Cluster



Matrix Transpose on Sunwulf Cluster



Conclusions

- We introduced an approach to automatically applying memory optimizations to derived datatype packing
- Automatic selection of optimized templates based on datatype
- Achieved significant performance improvement for simple, nested and indexed derived datatypes
- We plan to incorporate this work into the production MPICH2 source