# Improving the Performance of MPI Derived Datatypes by Optimizing Memory Access Cost

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

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

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

# Outline

- What are MPI derived datatypes?
- Performance of MPI derived datatypes
- How can the performance be improved?
- Optimizing memory access cost
- Performance results
- Future work

# MPI Datatypes

- In MPI, the data to sent or received is described by a triple (address, count, datatype)
  - e.g. 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 MPI datatypes
  - a strided array of blocks datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
  - distributed arrays and subarrays

# Why Datatypes?

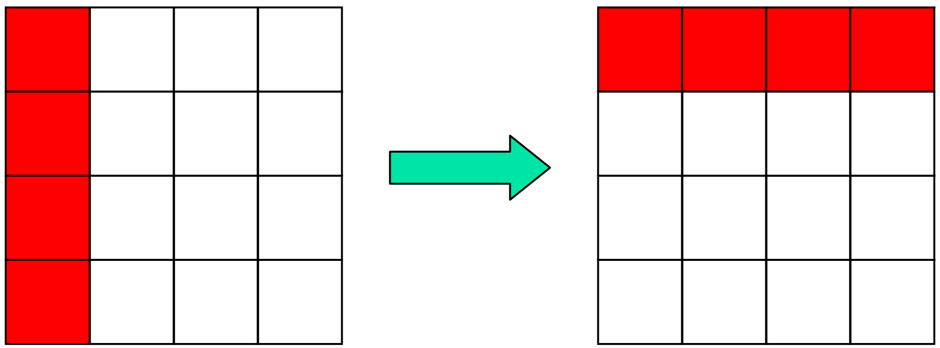
- To support communication between processes on machines with very different memory representations and lengths of elementary datatypes
- Specifying noncontiguous layout of data in memory
  - can reduce/optimize memory-to-memory copies in the implementation
  - allows the use of special hardware (scatter/gather)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

# 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

# Example: Matrix Transpose

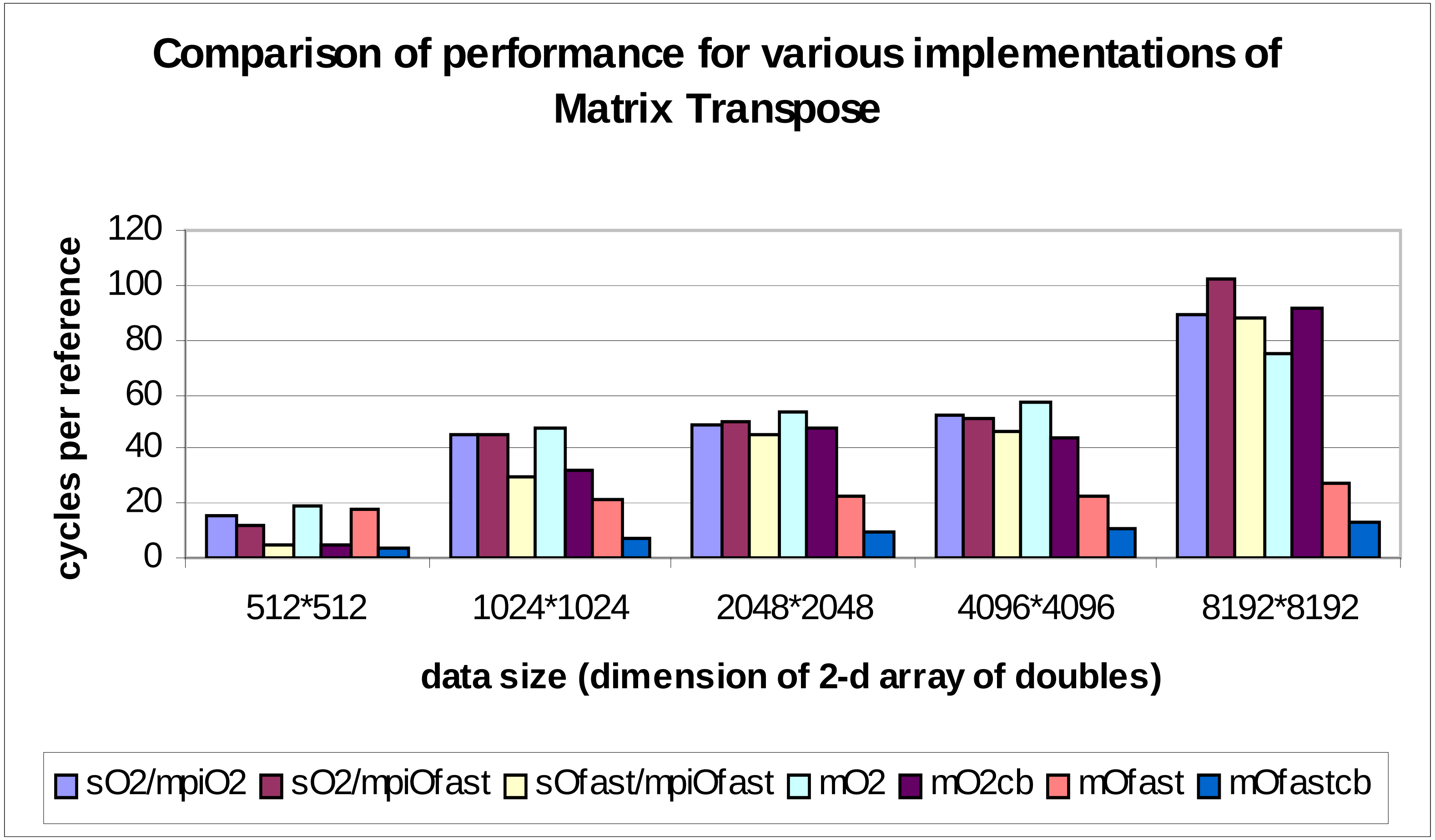
Row-major



# Matrix Transpose contd.

- We compared the performance of two programs:
  - An MPI program with derived datatypes. The MPI implementation does its usual (unoptimized) packing of data.
  - An MPI program with manually optimized packing of data using array padding and cache blocking
- Measured performance with various compiler optimizations on SGI MIPS R10000

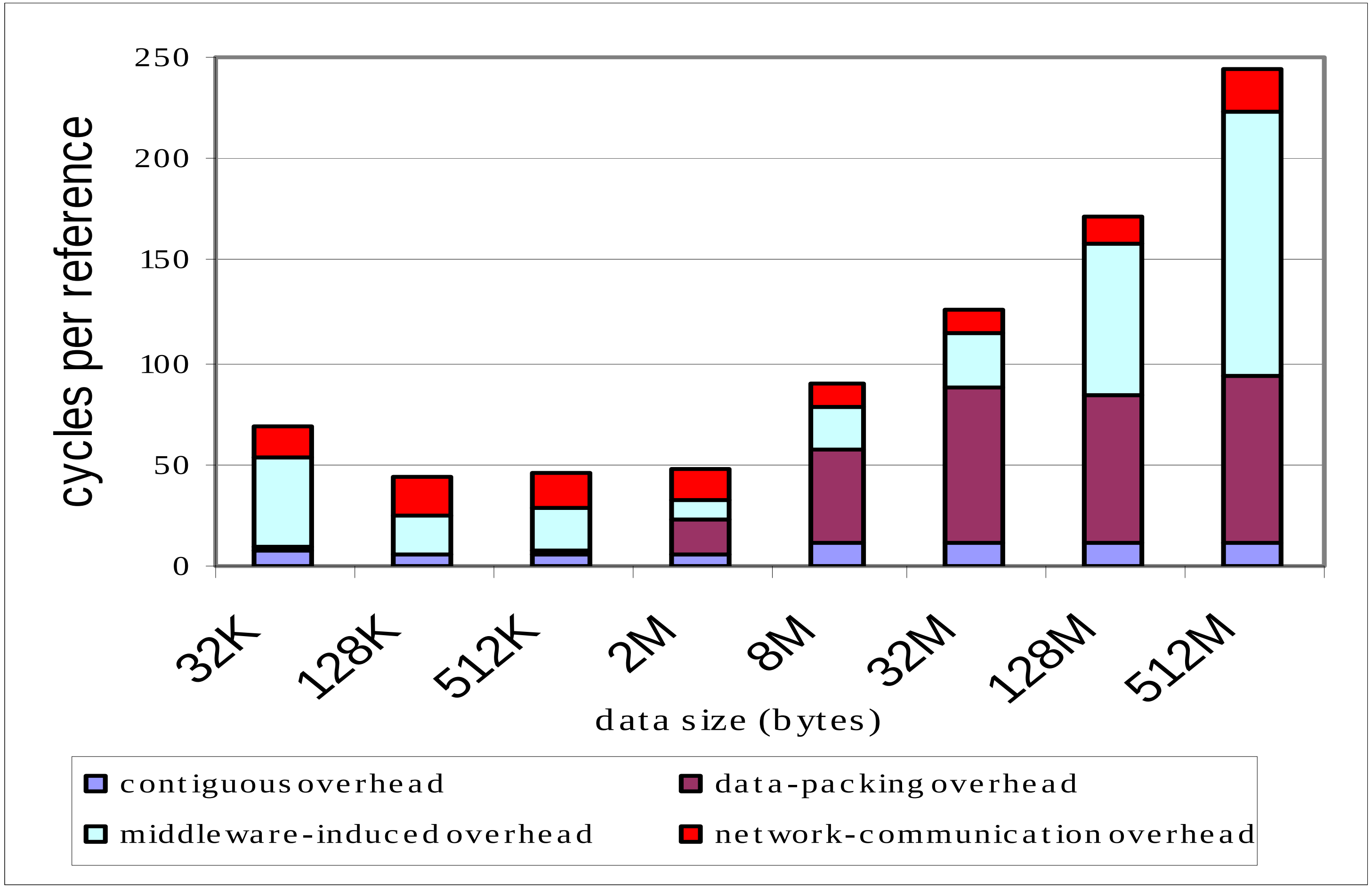
# Performance Comparison



# Observations

- Just leaving it to the compiler to optimize the packing code is not enough
- The compiler must be given memory-optimized code for best performance

# Quantifying Communication Cost for Matrix Transpose

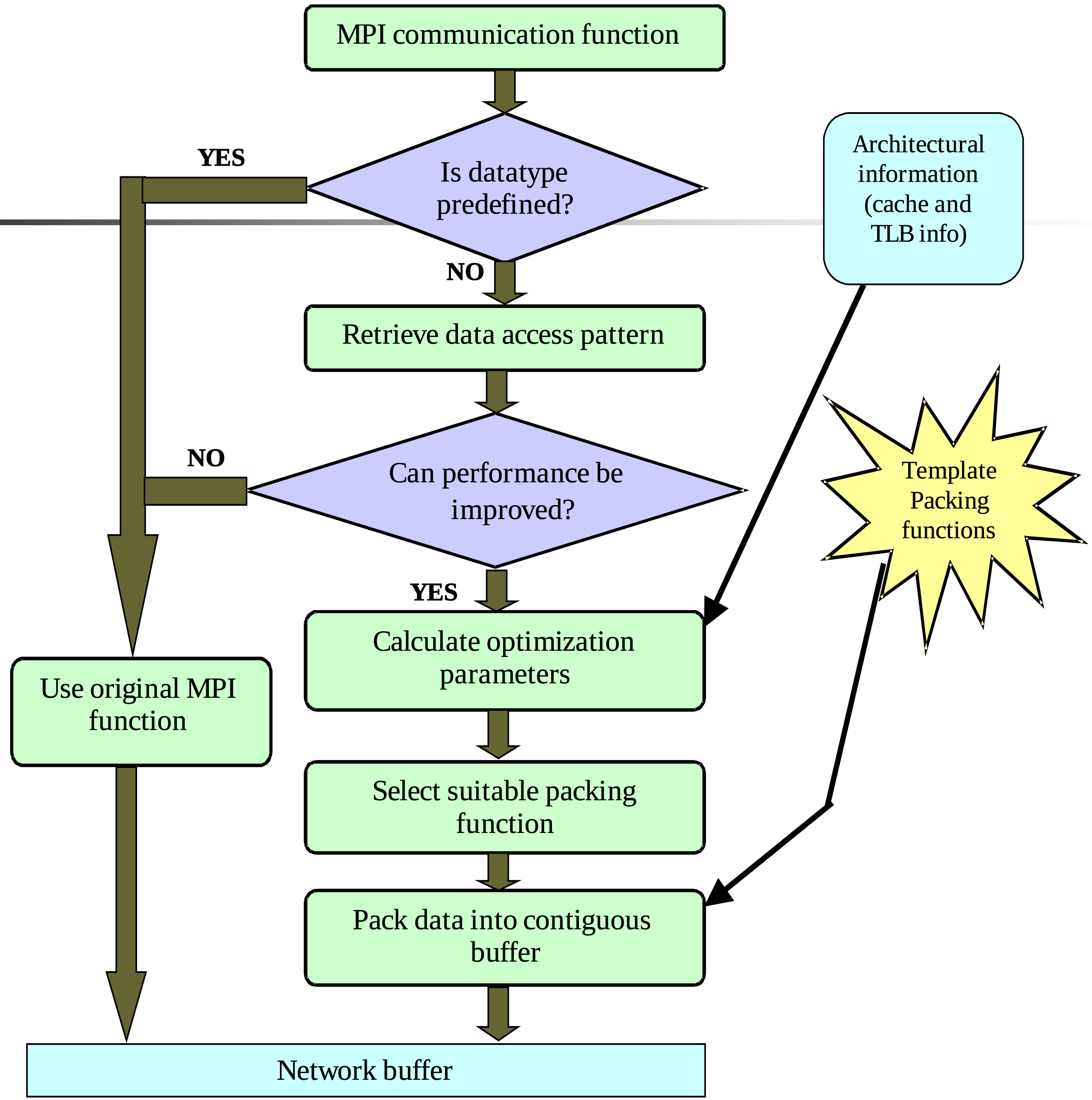


# Improving the Performance of Derived Datatypes

- The performance of derived datatypes can be improved in two ways:
  - Improving the data structures used to store derived datatypes internally in the MPI implementation (Traff ‘99, Gropp-Ross ‘03)
  - Optimized packing of noncontiguous data into a contiguous buffer for networks that support only contiguous messages (this paper)

# Reducing Memory Access Cost

- Optimize the performance based on the data access pattern and the memory architecture of the machine
- Improvement performed at two levels
  - At MPI_Type_commit – Determine the data access cost and possibility of optimization. If optimization is possible, find all the loop optimization parameters (Tile size, array padding etc.,)
  - At MPI_Send - If optimization is possible, use the loop optimizations to pack data into a contiguous buffer before sending


MPI communication function
Is datatype predefined?
YES
NO
Retrieve data access pattern
Can performance be improved?
NO
YES
Architectural information (cache and TLB info)
Template Packing functions
Calculate optimization parameters
Use original MPI function
Select suitable packing function
Pack data into contiguous buffer
Network buffer

# Is Performance Improvement Possible?

- Performance degrades drastically when
  - the data needed by a program cannot be reused from various levels of cache, and
  - when the pages referenced by a program do not have an entry in the TLB (TLB misses)

# Is Performance Improvement Possible? (2)

- The Translation Look-aside Buffer (TLB) stores the mapping from a virtual page address to a physical page address
- TLB size is typically small
- Cost of TLB misses is high

# Is Performance Improvement Possible? (3)

- We determine whether the unoptimized packing will result in TLB misses and whether the TLB misses can be reduced by blocking the code appropriately
- If yes, we use our optimized packing algorithm

# Choosing a Block Size

- Choose a block size that will minimize TLB misses
- We choose a block size that uses half the entries in the TLB for the block, leaving the other half for the contiguous buffer and other variables in the program

# Choosing a Packing Function

- We classify access patterns into combinations of contig/noncontig, fixed/variable block sizes, fixed/variable strides.
- For each combination we use a different packing function with architecture-dependent parameters

# Current Impl. Of MPI_Send in MPICH-1

```
MPI_Send (data, datatype, dest)
{
    if (datatype is basic datatype)
    {
       Send (data) to the network buffer.
    }
    else (datatype is derived datatype)
    {
       MPI_Pack (data, datatype, buffer);
        /* MPI_Pack () cost is very high for large data sets
          and powers-of-2 dimension arrays */
       Send (buffer) to the network buffer.
    }
}
```

# Memory Conscious Impl. Of MPI_Send (1)

```
MPI_Send (data, datatype, dest)
{
    if (datatype is basic)
    {
        PMPI_Send (data, datatype, dest);
    }
    else (datatype is derived and optimization possible)
    {
        packing_algorithm = Select_packing_algorithm
                            (data, datatype);
        pack (packing_algorithm, data, datatype, buffer);
        PMPI_Send (data, datatype, dest);
    }
}
```

# Memory Conscious Impl. Of MPI_Send (2)

```
Select_best_packing_algorithm (data, datatype)
{
    if (data fits into cache/TLB)
    {
        packing_algorithm = PMPI_Pack (data, datatype);
    }
    else
    {
        calculate_optimization_params (datatype,
                                   system_info, &params);
        choose_packing_algorithm (params, data, datatype,
                                          &packing_algorithm);
     }
     return (packing_algorithm);
}
```

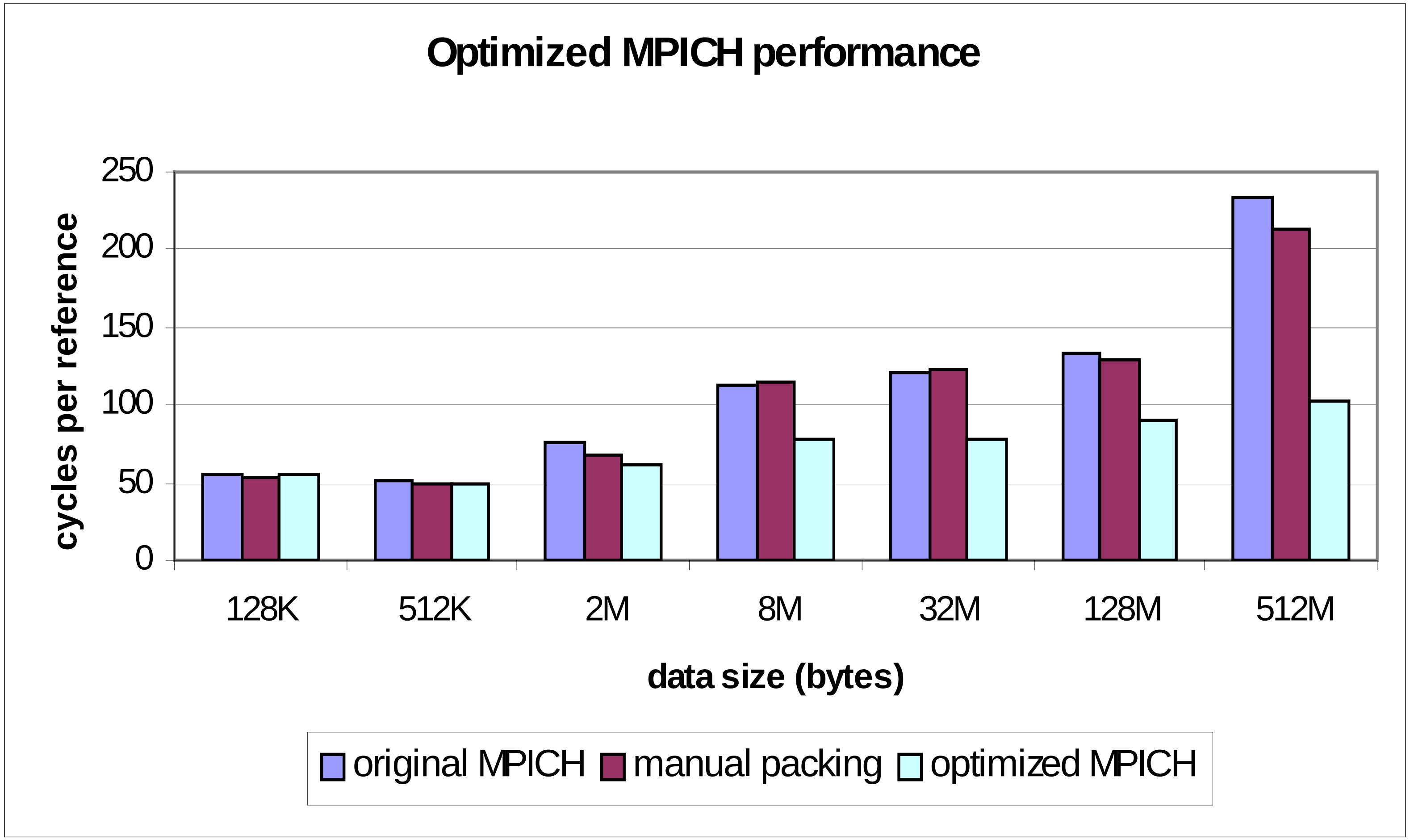
# Memory Conscious Impl. Of MPI_Send (3)

```
pack (packing_algorithm, data, datatype, buffer)
{
     if (packing_algorithm == PMPI_Pack)
     {
          PMPI_Pack (data, datatype, buffer);
     }
     else
     {
      /* Here come the template implementations for
         various data-access patterns with optimized
         parameters */
     }
}
```

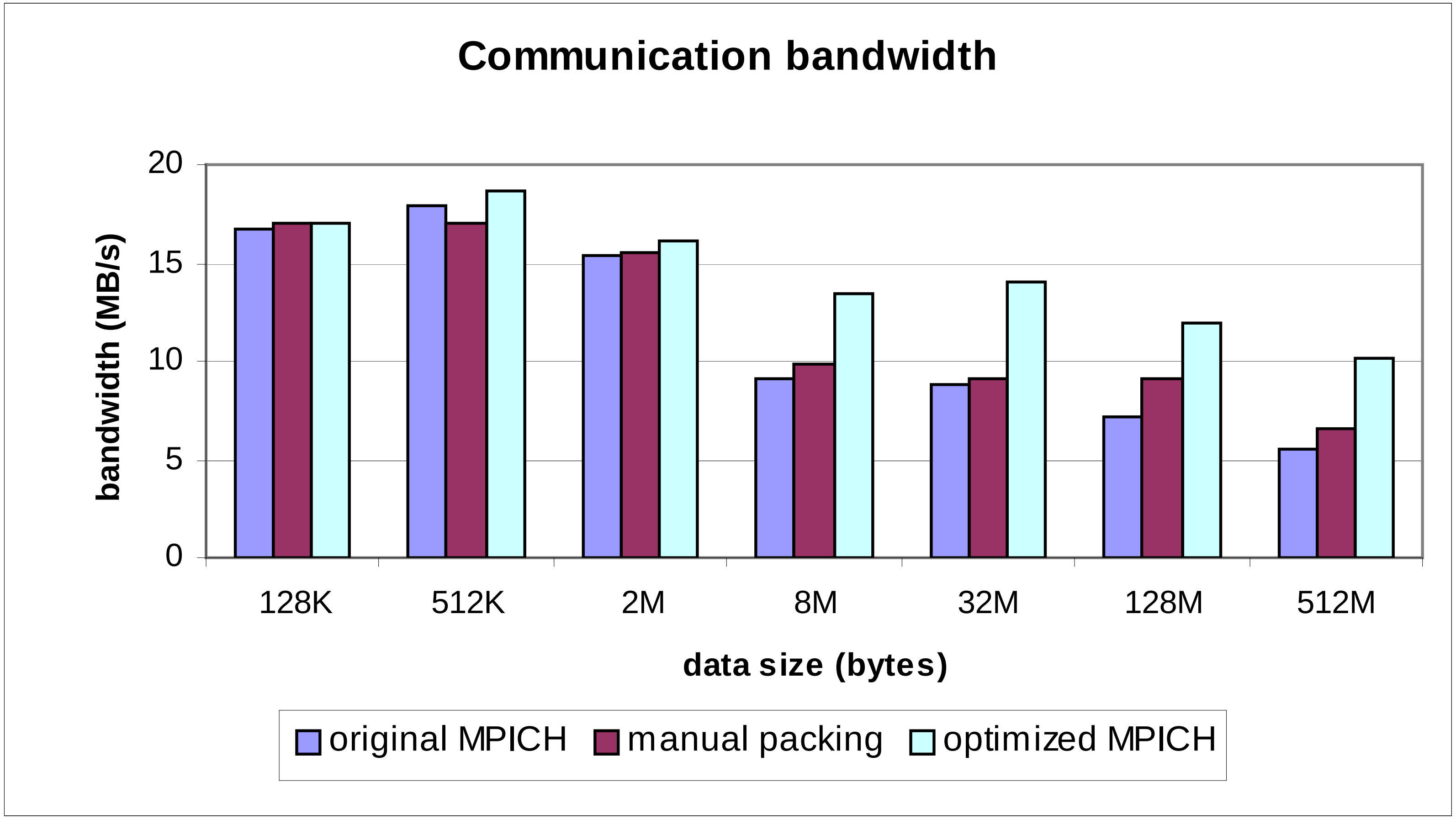
# Performance Evaluation

- SGI Origin 2000
  - MIPS R10000 Processor, 195MHz
  - 32KB, 2-way set associative, L1 cache
  - 4MB, 2-way set associative, off-chip L2 cache
  - IRIX 6.5.14 operating system
  - MPI implementation: MPICH 1.2.5 with shared memory device
  - Hardware counters are used to measure the performance
- Performance results are compared to implement Matrix Transpose for three cases:
  - Original MPICH 1.2.5 with derived datatypes
  - Original MPICH 1.2.5 with manual (unoptimized) packing (no derived datatypes)
  - Derived datatypes with our optimized packing algorithm

# Improved performance of MPI derived datatypes



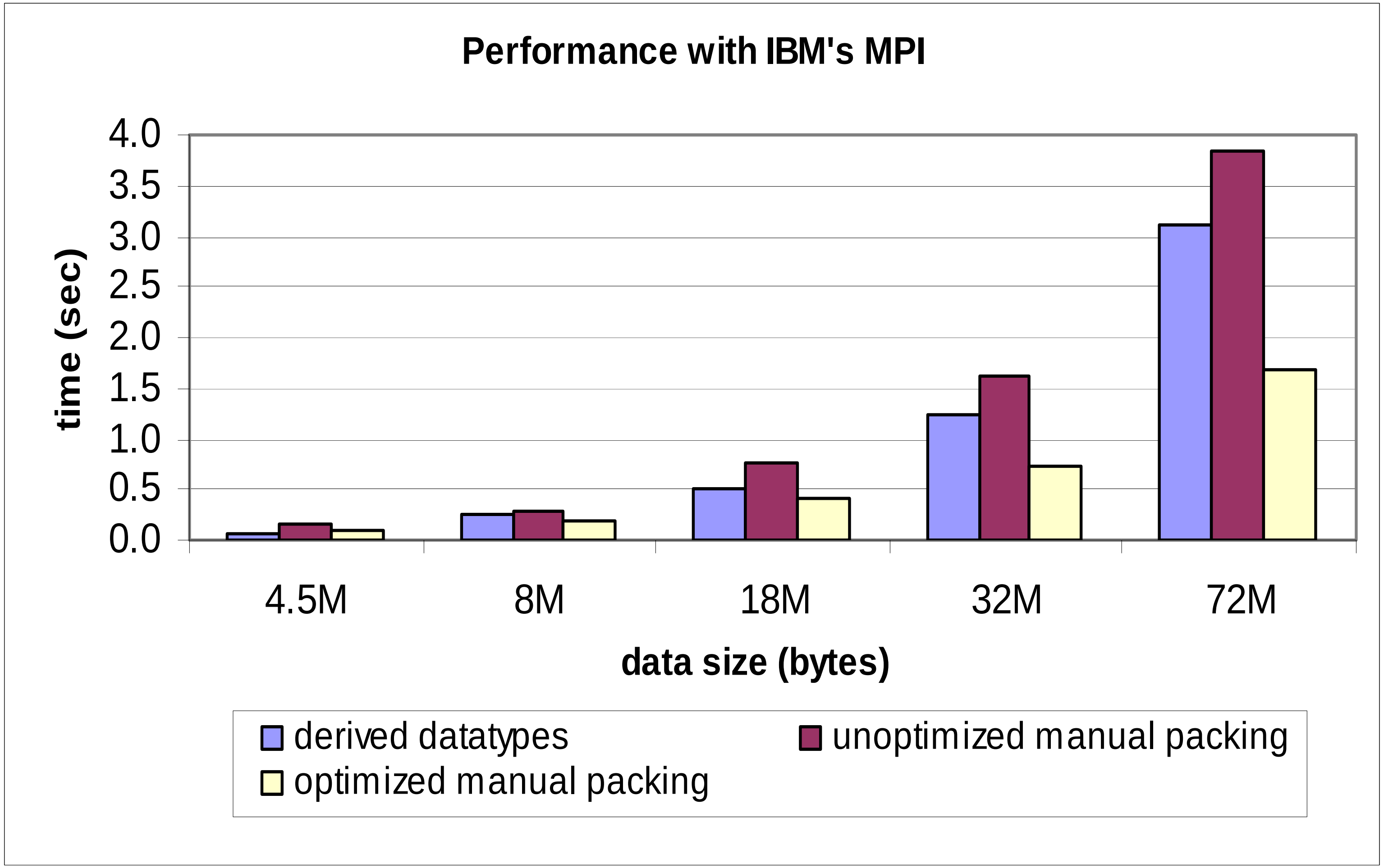
# Bandwidth Improvement



# Performance Tests with IBM's MPI

- IBM Blue Horizon at San Diego Supercomputing Center
  - Power3 processors run at 375 MHz
  - 64KB, 128-way set associative L1 cache
  - 8MB, 4-way set associative L2 cache
  - MPI implementation: IBM MPI

# Improved bandwidth of MPI derived datatypes



# Conclusions and Future Work

- This work shows that the performance of derived datatypes can be improved by optimizing memory copies
- We plan to incorporate this work into MPICH-2 and its new improved implementation of derived datatypes.
- We plan to extend the optimizations to include more loop optimizations, such as loop interchange and loop unrolling
- A model to predict the memory access cost based on data access pattern is under development