Incorporating MPI into Spiral WHT

Timothy A. Chagnon
Thomas A. Plick

December 13, 2007
The MPI Standard

- MPI is a network protocol and API standard with several popular implementations for FORTRAN, C, C++
  - Additional implementations exist for Python, Java, OCaml and others
- An implementation consists of:
  - library of calls to explicitly pass data between processes
    \texttt{MPI\_Send\ MPI\_Recv} etc.
  - programs to spawn and manage multiple processes
    \texttt{mpiexec\ mpirun}
- MPI provides many communication routines, but leaves it up to the programmer to specify the parallel execution and synchronization of distributed memory
History and Implementations

- **MPI-1 in 1994**
  - Started as MPI Forum at Supercomputing ’92
  - Broad collaboration from the parallel computing community
  - Defined most of the basic communication routines
  - Generally much more more popular than MPI-2

- **MPI-2 in 1997 added**
  - Parallel file I/O
  - Remote memory operations
  - Dynamic process management

- First implemented as MPI-CH from Argonne Nat’l Laboratory
- LAM/MPI was created by the Ohio Supercomputing Center
- Open MPI is a new project derived from LAM/MPI
Single-program, multiple data

- SPMD: MPI favors this structure of parallelism
- Written as a single program with conditional execution based on process number
- Conditionals provide a combined SIMD/MIMD like environment
- Consider the following example of forking in C:

```c
int pid = fork();
if (pid){
    /* I am the parent process ... */
} else {
    /* I am the child process ... */
}
```
Starting an MPI program

- Issue the command `mpiexec -n 10 myprog`
- `mpiexec` starts 10 processes and provides each to find its `rank` (its ID from 0 to \(n - 1\))
  ```c
  int n, myRank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &n);
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  /* Continue based on myRank */
  ...
  ```
- They can be spread across CPUs or hosts, as the MPI implementation allows
New dsplit node with method parameter

Initially same as DDL:
- Binary Split of $WHT_N = (W_R \otimes I_S)(I_R \otimes W_S)$
- Pseudo-Transpose on either side of the left factor
- Divide $I \otimes$ loops across multiple processors

Do MPI_Init for the first time here:
- non-MPI plans don’t need it
- No extra code in calling programs, e.g. verify

Since all MPI threads start at main(), they will all enter dsplit_apply() with some input vector.
- We assume that only process 0 has valid data, so we must Scatter before and Gather after computation
- This could be relaxed if the function calling wht_apply pre-arranges the data.
Pseudocode for dsplit_apply()

MPI_Init(NULL,NULL);
MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
MPI_Comm_size (MPI_COMM_WORLD, &totalRank);
localN = N/totalRank;

MPI_Scatter(x, localN, ...);

for i = 0 .. R/totalRank
    Ws[0]->apply(x+i*S);

mpi_transpose(x, min(S,R), method);

for i = 0 .. S/totalRank
    Ws[1]->apply(x+i*R);

mpi_transpose(x, min(S,R), method);

MPI_Gather(x, localN, ...);
Transform the strided access $\otimes I$ factor into block access:

$$WHT_N = M^N_S(I_S \otimes W_R)M^N_S(I_R \otimes W_S)$$

Identity allows us to only locally transpose in horizontal blocks.

Which corresponds to swapping bits at the ends of the addresses.
Distributed Pseudo-Transpose

- When the address space is distributed across multiple processes, the top bits become the Process ID (pid) and different cases of communication emerge.

- $np = s$ (Number of Processor bits = Swap bits)

  
  \[
  \begin{array}{c}
  \text{b}_0 \quad \text{b}_1 \quad \cdots \quad \text{b}_n \\
  \text{s} \quad \text{n-1} \quad \text{n-2} \quad \text{b} \\
  \end{array}
  \rightarrow
  \begin{array}{c}
  \text{b}_0 \quad \text{b}_1 \quad \cdots \quad \text{b}_n \\
  \text{s} \quad \text{n-1} \quad \text{n-2} \quad \text{b} \\
  \end{array}
  \]

  - Send to each processor $q$:
    $2^n-2^s$ words at stride $2^s + \text{offset } q$
  - Receive from each processor $q$:
    $2^n-2^s$ words at stride $2^s + \text{offset } q$
Distributed Pseudo-Transpose (cont.)

- **$np > s$**

  
  - Send to each processor $q$ at processor-stride $2^{np-s}$: $2^{n-2s}$ words at stride $2^s + \text{offset } q/2^{np-s}$
  
  - Receive to each processor $q$ at processor-stride $2^{np-s}$: $2^{n-2s}$ words at stride $2^s + \text{offset } q/2^{np-s}$

- **$np < s$**

  
  - Send to each processor $q$: contiguous blocks of size $2^{s-p}$ $2^{n-np-s}$ blocks at stride $2^s + \text{offset } q2^{s-np}$
  
  - Receive to each processor $q$: $2^{s-p}$ words at stride $2^{n-s}$ $2^{n-2s}$ blocks at stride $2^s + \text{offset } q2^{s-np}$ $2^{s-p}$ times incrementing by 1 each time
We can generalize the above cases into parameters describing what data to send or receive.

First we define a zero-bound function:

$$zb(x) = \begin{cases} 
  x, & x \geq 0 \\
  0, & x < 0 
\end{cases}$$

And then define the following parameters:

- Processor Stride
  \[ PS = 2^{zb(p-s)} \]
- Count number of Blocks
  \[ C = 2^{n-np-s} \]
- Block Size
  \[ B = 2^{zb(s-p)} \]
- Stride of Blocks
  \[ S = 2^s \]

\texttt{mpi.transpose()} generates these parameters to describe what must be communicated and passes them to the desired communication method.
Communication Methods

- **Alltoall**
  - Scatters and Gathers equal contiguous chunks of data to each processor in a communication group.
  - A Temporary Communicator/Group can be used to stride processors.
  - Unfortunately the MPI Strided Vector type doesn’t get overlapped as required, so data must be Packed and Unpacked.

- **Sendrecv 3-way**
  - $NP/PS$ rounds of communication.
  - $sendTo = (pid + round \times PS) \mod (NP/PS)$
  - $recvFrom = (pid - round \times PS) \mod (NP/PS)$

- **Sendrecv 2-way**
  - $NP/PS$ rounds of communication.
  - $recvFrom = sendTo = (pid \text{ XOR } round \times PS)$

- **Other variations include**
  - Use copy instead of MPI for local communication.
  - Sendrecv replace using a single buffer.
  - Many small Alltoalls for each chunk.
Our test machines

- Tom’s laptop: Intel T2080 @ 1.73GHz, 1 MB cache; 1 dual-core proc
- Tux: AMD Opteron(tm) Processor 244 @ 1.8 GHz, 1 MB cache; 1 dual-core proc
- Tim’s desktop: AMD Athalon 3800+ @ 2.0GHz, 1 MB cache; 1 dual-core proc
Comparison of communication methods in MPICH

Test of communication methods in MPICH

Time for 1 execution relative to method 2

size $2^n$

0: MPI sendrecv, p2p; 1: MPI sendrecv, 3way; 2: MPI nonblocking send and recv using p2p; 3: MPI gather with loop; 4: MPI Alltoall method; 5: MPI sendrecv, p2p replace
Comparison of communication methods in LAM

Test of communication methods in LAM

0: MPI sendrecv, p2p; 1: MPI sendrecv, 3way; 2: MPI nonblocking send and recv using p2p; 3: MPI gather with loop; 4: MPI_Alltoall method; 5: MPI sendrecv, p2p replace
 MPI transpose

Time for MPI transpose

- mpich 0
- lam 0
- mpich 1
- lam 1

Time for 1 execution relative to mpich 0

Size $2^n$
New MPI Transpose Code

New MPI Transpose, various swap and methods
koala AMD Athlon(tm) 64 X2 Dual Core Processor 3800+ 512 KB L2 Cache

- s=np Alltoall
- s=np 3way
- s=np 3way+copy
- s=np 2way
- s=n/2 Alltoall
- s=n/2 3way
- s=n/2 3way+copy
- s=n/2 2way