Implementation of neighboring communication in QWS

Issaku Kanamori (RIKEN)

August 4th, 2020
Asia-Pacific Symposium for Lattice Field Theory (APLAT 2020)
Outline

1. Introduction specification of Fugaku
2. Algorithm and Implementation double buffering
3. As a communication library benchmark with a 2-dim Poisson equation
4. Summary and Outlooks
Outline

1. Introduction specification of Fugaku
2. Algorithm and Implementation double buffering
3. As a communication library benchmark with a 2-dim Poisson equation
4. Summary and Outlooks

Acknowledgments
This talk is based on discussion with the LQCD codesign team in flagship 2020 project:
RIKEN) Y.Nakamura, I.K, K.Nitadori, M.Tsuji
Fujitsu) I.Miyoshi, Y.Mukai, T.Nishiki
Hiroshima) K.-I.Ishikawa
KEK) H.Matsufuru

I.K. also thanks the MEXT as “Program for Promoting Researches on the Supercomputer Fugaku” (Simulation for basic science: from fundamental laws of particles to creation of nuclei) and JICFuS.
The software used for the evaluation, such as the compiler, is still under development and its performance, which is obtained by “performance estimation tool” and even actual execution on a prototype machine, may be different when the supercomputer Fugaku starts is operation.
Feature of Fugaku: TofuD interconnect

- Flops/node: 3TFlops[double] (×23 of K-computer)
- injection BW/node 40.8GB/s (only ×2 of K-computer)
- communication is important

- 6D torus/mesh network with 10 nearest neighbors
  (small 3d “torus” [2 × 3 × 2]) × (large 2d torus × 1d mesh),
- Each node (≠ process) can send data to 6 different directions simultaneously [QCD has 8 directions]
- Latency: 0.49μs, > 90% efficiency for the nearest neighbor put
- interface for TofuD: uTofu \(\leftrightarrow\) QWS uses uTofu for neighboring comm.

keywords

- TNI: Tofu Network Interface (RDMA engine)
- RDMA: Remote Direct Memory Access
- uTofu: Low Level Communication API for TofuD
QCD Wide Simd Library: see Y. Nakamura’s talk

- Clover solver designed for Fugaku
  also runs on other architectures s.t. intel
- https://github.com/RIKEN-LQCD/qws
Algorithm and Implementation
Double Buffering

**single buffering**
- sender: 1 send buffer
- receiver: 1 recv. buffer

**double buffering**
- sender: 1 send buffer
- receiver: 2 recv. buffers, used alternatingly
Double Buffering

**single buffering**
- sender: 1 send buffer
- receiver: 1 recv. buffer

**double buffering**
- sender: 1 send buffer
- receiver: 2 recv. buffers, used alternatingly

- smaller overhead: no need to check if the recv. buffer is ready
  the other buffer is always available
- robust against load imbalance
  no need to wait till the recv. buffer becomes available
Implementation: using uTofu interface

- smaller latency than calling MPI
- we use RDMA put write directly to the memory in the target process
- can (or must) tune the TNI assignment
  
  6 TNIs: 6 simultaneous RDMA put to different directions
  
  the load to each TNI should be balanced

- the boundary size
  
  depends on the direction: the local volume can be a hyper-rectangular

- the rank map
  
  6-dim Tofu coordinate → 4-dim QCD proc. coordinate → 1-dim MPI rank id
  
  a proper rank map is important
  
  the logical “neighbor” may not be a physical neighbor
Implementation in QWS

**rdma_utofu_comlib.c**: wrapper functions for calling uTofu

```c
rdma_comlib_data buf;

// buf is to send data (of size) to dst_rank and receive from rcv_rank using TNI of tni_id
// cf. MPI_Recv_init and MPI_Send_init
rdma_comlib_new(&buf, &tni_id, &dst_rank, &rcv_rank, &size);
// start sending with RDMA put
rdma_comlib_isendrecv(&buf);
rdma_comlib_irecv_check(&buf); // cf. MPI_Wait for receiving
rdma_comlib_jsend_check(&buf); // cf. MPI_Wait for sending
```

**rdma_comlib_2buf**: a class for double buffering

```c
... // initialize communication in x-direction
buff_rdma[0].init(tni_list[0], pxb, pxf, size);
buff_rdma[1].init(tni_list[1], pxf, pxb, size);

buff_rdma[req].isendrecv(); // in void xbound(int req, int prec)
buff_rdma[req].irecv_check(); // in void xbound_wait(int req, int prec)
```

**qws_xbound_rdma.cc**: communication routines in QWS (uTofu RDMA version)
Some details of the implementation

with a proper data alignment and suitable flags to uTofu interface

**sender**
- uTofu put is thread parallelized

**receiver**
- we monitor the last byte of the buffer to check the data has arrived
- received data goes directly to the cache  
  cache injection
As a Communication Library

Benchmark with a 2-dim Poisson equation on Fugaku
$$Mx = b \text{ with }$$

$$\begin{align*}
(Mx)(i,j) &= (4 + m^2)x(i,j) - x(i+1,j) - x(i-1,j) - x(i,j+1) - x(i,j-1) \\
&\equiv Dx \\
&\equiv Hx
\end{align*}$$

Jacobi method

$$x^{(k)} \rightarrow x^{(k+1)} = D^{-1}(b - Hx^{(k)})$$

Only the hopping $H$ contains the communication

- fixed number of iterations: 1000
- calculation of the residual norm (MPI_Allreduce) in every 10 iter.
- local lattice size: $60 \times 60$
- communication buffer (60 elements for each direction) in enlarged by 1–8192 (+ a flag as the end of buffer + alignment)
Theoretical Bandwidth for uTofu Communication

4 MPI ranks/node (2 × 2 ranks in each node)

**pattern 1**

- effective bandwidth: $40.8 \times \frac{16}{24} = 27.2 \text{GB/s}$

**pattern 2 (round robin)**

- effective bandwidth: $40.8 \times \frac{16}{18} = 36.3 \text{GB/s}$

- a proper TNI assignment is important to maximize the effective BW
- if (boundary size for $x$)\neq (boundary size for $y$), we can enjoy more games with TNI assignment
- (TNI assignments for MPI communication is unclear)
Elapsed Time vs. Amount of Communication: 96 nodes

- **Utofu double buffering: 96 nodes (384 ranks)**
  - Mult
  - Comm. non-overlap
  - Comm. overlap
  - Allreduce
  - 27.4 GB/s
  - 36.2 GB/s

- **Utofu double buffering: 96 nodes (384 ranks), TNI round robin**
  - Mult
  - Comm. non-overlap
  - Comm. overlap
  - Allreduce
  - 27.4 GB/s
  - 36.2 GB/s

- **MPI double buffering: 96 nodes (384 ranks)**
  - Mult
  - Comm. non-overlap
  - Comm. overlap
  - Allreduce
  - 27.4 GB/s
  - 36.2 GB/s

- **MPI single buffering: 96 nodes (384 ranks)**
  - Mult
  - Comm. non-overlap
  - Comm. overlap
  - Allreduce
  - 27.4 GB/s
  - 36.2 GB/s
Elapsed Time vs. Amount of Communication: 96 nodes

- uTofu double buffering: 96 nodes (384 ranks)
  - comm. is hidden

- uTofu double buffering: 96 nodes (384 ranks), TNI round robin
  - comm. is hidden

- MPI double buffering: 96 nodes (384 ranks)
  - comm. is hidden

- MPI single buffering: 96 nodes (384 ranks)
  - comm. is hidden
Elapsed Time vs. Amount of Communication: 96 nodes

I. Kanamori: APLA T 2020, Aug. 4, 2020
Elapsed Time vs. Amount of Communication: 96 nodes

- **utofu double buffering**: 96 nodes (384 ranks)
  - Good scaling
  - Smaller overhead
  - Comm. is hidden

- **utofu double buffering**: 96 nodes (384 ranks), TNI round robin
  - Good scaling
  - Comm. is hidden

- **MPI double buffering**: 96 nodes (384 ranks)
  - Comm. is hidden

- **MPI single buffering**: 96 nodes (384 ranks)
  - Good scaling

I. Kanamori: APLAT 2020, Aug. 4, 2020
Saturation of the bandwidth for large data size

I. Kanamori: APLAT 2020, Aug. 4, 2020
Saturation of the bandwidth for large data size

Good saturation of the theoretical band width
Saturation of the bandwidth for large data size

**MPI double buf. is also reasonable**

**MPI**

**good saturation of the theoretical band width**

**uTofu**
Summary and Outlooks
Neighboring Communication of QWS

- algorithm: double buffering
- implemented with uTofu  
  MPI version of QWS is available as well
- a proper TNI assignment is important  
  rank map is also important
- can be used as a library: ex. with 2-dim Poisson eq.
  - good saturation of the theoretical band width
  - good weak scaling
  - room (and/or freedom) for further optimization of TNI

Outlooks

- performance of QWS with practical system sizes
- [comm. part of] QWS + existing QCD code sets

TNI: Tofu Network Interface (RDMA engine)  
RDMA: Remote Direct Memory Access
uTofu: Low Level Communication API for TofuD
Neighboring Communication of QWS

- algorithm: double buffering
- implemented with uTofu (RDMA engine)
- MPI version of QWS is available as well
- a proper TNI assignment is important
- rank map is also important
- can be used as a library: ex. with 2-dim Poisson eq.
  - good saturation of the theoretical band width
  - good weak scaling
  - room (and/or freedom) for further optimization of TNI

Outlooks

- performance of QWS with practical system sizes
- [comm. part of] QWS + existing QCD code sets

TNI: Tofu Network Interface (RDMA engine)  RDMA: Remote Direct Memory Access
uTofu: Low Level Communication API for TofuD

Thank you.
Backup Slides
Double Buffering Algorithm

To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

\[ P \] : Packed
\[ P \] : Sending

recv. buf.

\[ R \] : Receiving
\[ R \] : Recv. done
\[ U \] : being Used

```plaintext
1 // 1st iter. send buffer
2 pack the boundary data
3 start sending
4 computation: bulk
5 wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 wait for sending is done
9 switch the buffer to send
// 2nd iter.
10 pack the boundary data
11 start sending
12 computation: bulk
13 wait for the boundary data comes
14 computation: boundary
15 clear the received flag
16 wait for sending is done
17 switch the buffer to send
...```
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

### send buf.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// 1st iter.</td>
<td>send buffer</td>
</tr>
<tr>
<td>2</td>
<td>pack the boundary data</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>start sending</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>computation: bulk</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>wait for the boundary data comes</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>computation: boundary</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>clear the received flag</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>wait for sending is done</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>switch the buffer to send</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>// 2nd iter.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>pack the boundary data</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>start sending</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>computation: bulk</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>wait for the boundary data comes</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>computation: boundary</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>clear the received flag</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>wait for sending is done</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>switch the buffer to send</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### recv. buf.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// 1st iter.</td>
<td>recv. buffers</td>
</tr>
<tr>
<td>2</td>
<td>pack the boundary data</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>start sending</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>computation: bulk</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>wait for the boundary data comes</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>computation: boundary</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>clear the received flag</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>wait for sending is done</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>switch the buffer to send</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>// 2nd iter.</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>pack the boundary data</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>start sending</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>computation: bulk</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>wait for the boundary data comes</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>computation: boundary</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>clear the received flag</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>wait for sending is done</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>switch the buffer to send</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

**send buf.**

- **P**: Packed
- **P**: Sending

**recv. buf.**

- **R**: Receiving
- **R**: Recv. done
- **U**: being Used

### Double Buffering Algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// 1st iter. send buffer</td>
</tr>
<tr>
<td>2</td>
<td>pack the boundary data</td>
</tr>
<tr>
<td>3</td>
<td>start sending</td>
</tr>
<tr>
<td>4</td>
<td>computation: bulk</td>
</tr>
<tr>
<td>5</td>
<td>wait for the boundary data comes</td>
</tr>
<tr>
<td>6</td>
<td>computation: boundary</td>
</tr>
<tr>
<td>7</td>
<td>clear the received flag</td>
</tr>
<tr>
<td>8</td>
<td>wait for sending is done</td>
</tr>
<tr>
<td>9</td>
<td>switch the buffer to send</td>
</tr>
<tr>
<td>10</td>
<td>// 2nd iter.</td>
</tr>
<tr>
<td>11</td>
<td>pack the boundary data</td>
</tr>
<tr>
<td>12</td>
<td>start sending</td>
</tr>
<tr>
<td>13</td>
<td>computation: bulk</td>
</tr>
<tr>
<td>14</td>
<td>wait for the boundary data comes</td>
</tr>
<tr>
<td>15</td>
<td>computation: boundary</td>
</tr>
<tr>
<td>16</td>
<td>clear the received flag</td>
</tr>
<tr>
<td>17</td>
<td>wait for sending is done</td>
</tr>
<tr>
<td>18</td>
<td>switch the buffer to send</td>
</tr>
<tr>
<td>19</td>
<td>...</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>// 1st iter. recv. buffers</td>
</tr>
<tr>
<td>2</td>
<td>pack the boundary data</td>
</tr>
<tr>
<td>3</td>
<td>start sending</td>
</tr>
<tr>
<td>4</td>
<td>computation: bulk</td>
</tr>
<tr>
<td>5</td>
<td>wait for the boundary data comes</td>
</tr>
<tr>
<td>6</td>
<td>computation: boundary</td>
</tr>
<tr>
<td>7</td>
<td>clear the received flag</td>
</tr>
<tr>
<td>8</td>
<td>wait for sending is done</td>
</tr>
<tr>
<td>9</td>
<td>switch the buffer to send</td>
</tr>
<tr>
<td>10</td>
<td>// 2nd iter.</td>
</tr>
<tr>
<td>11</td>
<td>pack the boundary data</td>
</tr>
<tr>
<td>12</td>
<td>start sending</td>
</tr>
<tr>
<td>13</td>
<td>computation: bulk</td>
</tr>
<tr>
<td>14</td>
<td>wait for the boundary data comes</td>
</tr>
<tr>
<td>15</td>
<td>computation: boundary</td>
</tr>
<tr>
<td>16</td>
<td>wait for sending is done</td>
</tr>
<tr>
<td>17</td>
<td>switch the buffer to send</td>
</tr>
<tr>
<td>18</td>
<td>...</td>
</tr>
</tbody>
</table>
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

\[ P \]: Packed
\[ P \]: Sending

recv. buf.

\[ R \]: Receiving
\[ R \]: Recv. done
\[ U \]: being Used

---

```
// 1st iter.
1 pack the boundary data
2 start sending
3 wait for the boundary data comes
4 computation: boundary
5 clear the received flag
6 wait for sending is done
7 switch the buffer to send
8 // 2nd iter.
9 pack the boundary data
10 start sending
11 wait for the boundary data comes
12 computation: boundary
13 clear the received flag
14 wait for sending is done
15 switch the buffer to send
...
```

---

```
// 1st iter.
1 pack the boundary data
2 start sending
3 wait for the boundary data comes
4 computation: boundary
5 clear the received flag
6 wait for sending is done
7 switch the buffer to send
// 2nd iter.
8 pack the boundary data
9 start sending
10 wait for the boundary data comes
11 computation: boundary
12 clear the received flag
13 wait for sending is done
14 switch the buffer to send
...
```
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

1. Pack

recv. buf.

1. Receiving

send buffer

1 // 1st iter.
2 pack the boundary data
3 start sending
4 computation: bulk
5 wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 wait for sending is done
9 switch the buffer to send
10 // 2nd iter.
11 pack the boundary data
12 start sending
13 computation: bulk
14 wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 wait for sending is done
18 switch the buffer to send
19 ...

recv. buffers

1 // 1st iter.
2 pack the boundary data
3 start sending
4 computation: bulk
5 wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 wait for sending is done
9 switch the buffer to send
10 // 2nd iter.
11 pack the boundary data
12 start sending
13 computation: bulk
14 wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 wait for sending is done
18 switch the buffer to send
19 ...

I. Kanamori: APLAT 2020, Aug. 4, 2020
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

\[P\] : Packed

\[P\] : Sending

recv. buf.

\[R\] : Receiving

\[R\] : Recv. done

\[U\] : being Used

```
// 1st iter.
1 send buffer
2 pack the boundary data
3 \[P\] start sending
4 computation: bulk
5 \[R\] wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 \[R\] wait for sending is done
9 \[U\] switch the buffer to send

// 2nd iter.
10 \[U\] switch the buffer to send
11 pack the boundary data
12 \[R\] start sending
13 computation: bulk
14 \[U\] wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 \[U\] wait for sending is done
18 \[P\] switch the buffer to send

1 // 1st iter.
2 pack the boundary data
3 \[P\] start sending
4 computation: bulk
5 \[R\] wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 \[R\] wait for sending is done
9 \[U\] switch the buffer to send

// 2nd iter.
10 \[U\] switch the buffer to send
11 pack the boundary data
12 \[R\] start sending
13 computation: bulk
14 \[U\] wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 \[U\] wait for sending is done
18 \[P\] switch the buffer to send
```

I. Kanamori: APLAT 2020, Aug. 4, 2020
Double Buffering Algorithm

To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

<table>
<thead>
<tr>
<th>P</th>
<th>: Packed</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>: Sending</td>
</tr>
</tbody>
</table>

recv. buf.

<table>
<thead>
<tr>
<th>R</th>
<th>: Receiving</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>: Recv. done</td>
</tr>
<tr>
<td>U</td>
<td>: being Used</td>
</tr>
</tbody>
</table>

```
// 1st iter.
1 // 1st iter.
2 pack the boundary data
3 start sending
4 computation: bulk
5 wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 wait for sending is done
9 switch the buffer to send
10 // 2nd iter.
11 pack the boundary data
12 start sending
13 computation: bulk
14 wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 wait for sending is done
18 switch the buffer to send
19 ...
```

```
// 1st iter.
1 // 1st iter.
2 pack the boundary data
3 start sending
4 computation: bulk
5 wait for the boundary data comes
6 computation: boundary
7 clear the received flag
8 wait for sending is done
9 switch the buffer to send
10 // 2nd iter.
11 pack the boundary data
12 start sending
13 computation: bulk
14 wait for the boundary data comes
15 computation: boundary
16 clear the received flag
17 wait for sending is done
18 switch the buffer to send
19 ...
```
To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

1. pack the boundary data
2. start sending
3. computation: bulk
4. wait for the boundary data comes
5. computation: boundary
6. clear the received flag
7. wait for sending is done
8. switch the buffer to send
9. // 2nd iter.
10. pack the boundary data
11. start sending
12. computation: bulk
13. computation: boundary
14. wait for the boundary data comes
15. clear the received flag
16. wait for sending is done
17. switch the buffer to send
18. // 2nd iter.
19. pack the boundary data
20. start sending
21. computation: bulk
22. wait for the boundary data comes
23. computation: boundary
24. wait for sending is done
25. switch the buffer to send
26. ...
Double Buffering Algorithm

To hide a possible load imbalance btw. nodes, and to minimize the latency, we use double buffering algorithm and implement it with uTofu.

send buf.

(send buf.)

recv. buf.

even if the 1st buffer is still busy but the 2nd is available

no need to check if the recv. buf. is available
Hopping (Mult of $H$)

1. packing the boundary data
2. start sending/receiving the boundary data
3. calculate: internal area
4. wait for receiving
5. calculate: boundary area
6. wait for sending finished
Hopping (Mult of $H$)

1. packing the boundary data
2. start sending/receiving the boundary data
3. calculate: internal area
4. wait for receiving
5. calculate: boundary area
6. wait for sending finished

\[
\text{non-overlap} = \text{comm.} - \text{overlap} = \text{start sending/receiving} + \text{wait for receiving}
\]
weak scaling with different data size

uTofu double buffering

uTofu double buffering (round robin TNI)
weak scaling with different data size

**uTofu double buffering**

- **uTofu double buffering: 60 byte/direction/rank**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

- **uTofu double buffering: 7680 byte/direction/rank**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

- **uTofu double buffering: 491520 byte/direction/rank**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

**uTofu double buffering (round robin TNI)**

- **uTofu double buffering: 60 byte/direction/rank (TNI: round robin)**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

- **uTofu double buffering: 7680 byte/direction/rank (TNI: round robin)**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

- **uTofu double buffering: 491520 byte/direction/rank (TNI: round robin)**
  - Multi: ×
  - Comm. non-overlap: ○
  - Comm. overlap: ▲
  - Allreduce: △
  - Theo: 27.4 GB/s
  - Theo: 36.2 GB/s

Good weak scaling if the communication is fully overlapped.
if comm. becomes visible, good weak scaling for large # of nodes

utofu double buffering

utofu double buffering (round robin TNI)
weak scaling with different data size

uTofu double buffering (upper) and mpi single buffering (lower)
weak scaling with different data size

I. Kanamori: APLAT 2020, Aug. 4, 2020

uTofu double buffering (upper) and mpi single buffering (lower)

mult: uTofu double buf. (42 msec.) < MPI single buf. (52 msec.)