PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

COSC 6374
Parallel Computation

One-sided communication

Edgar Gabriel
Spring 2008

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

Pt-2-pt communication in MPI

```
MPI_Init(int *argc, char ***argv);
MPI_Finalize ();

MPI_Comm_rank (MPI_Comm comm, int *rank);
MPI_Comm_size (MPI_Comm comm, int *size);

MPI_Send (void *buf, int count, MPI_Datatype dat,
          int dest, int tag, MPI_Comm comm);
MPI_Recv (void *buf, int count, MPI_Datatype dat,
          int source, int tag,  MPI_Comm comm,
          MPI_Status *status);
```

Parallel Software Technologies Laboratory PSTL

Two-sided communication in MPI

- `MPI_Send/MPI_Recv` : sending and receiving process are both active during communication
- A message consists of
  - The data to be sent consisting of
    - Buffer pointer
    - Data type
    - Number of elements
  - message header (message envelope)
    - Rank of sender process
    - Rank of receiver process
    - Communicator
    - Tag

Parallel Software Technologies Laboratory PSTL

Message matching (I)

- How does the receiving process know, that the message which has just been received is really the message which the `MPI_Recv` expects?
  - Rank of the sender process in the message received has to match the rank of the sender process in the expected message
  - The tag of the message received has to match the tag as specified in the `MPI_Recv` operation
  - The communicator of the message received has to match the communicator as specified in `MPI_Recv`
- Message length not used for message matching

What happens if one of the elements does not match

- An unexpected message is a message which does not match the message envelope of any currently posted `MPI_Recv` or `MPI_Irecv`
- Unexpected messages are stored temporarily in so-called *unexpected message queues*
- A good MPI implementation has a flow-control protocol
  - Avoids very long messages in the unexpected message queues
- Upon calling MPI_Recv, the *unexpected message queue* of that communicator has to be checked, since the message might have been received already
  - Copying the data from unexpected queue to user address space required
  - Might require multiple comparisons of message envelopes to find the right message in the unexpected message queue

One-sided communication in MPI

- MPI-2 defines so-called one-sided communication:
  - A process can put some data into the main memory of another process (`MPI_Put`)
  - A process can get some data from the main memory of another process (`MPI_Get`)
- Target process not actively involved in the communication

One-sided communication

- Historically coming from shared-memory machines
  - a process can access the main memory of another process (*Direct Memory Access – DMA*)
  - Today: some networking cards and graphic cards have direct access to the main memory, bypassing the processor
- *Remote Direct Memory Access – RDMA*: a process can access the main memory of a process on another processor
  - Advantage: processors might be relieved from work
  - Disadvantage: process must be made aware if somebody modified its data

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

RDMA in MPI

- Problems:
  - How can a process define **which** parts of its main memory are 'open' for RDMA?
  - How can a process define **when** this part of the main memory is available for RDMA?
  - How can a process define **who** is allowed to access its memory?
  - How can a process define which elements in the remote memory it wants to access?

COSC 6374 – Parallel Computation
Edgar Gabriel

The window concept of MPI-2 (I)

```
MPI_Win_create(void *base, MPI_Aint size, int
        disp_unit, MPI_Info info, MPI_Comm comm,
        MPI_Win *win);
```

- An `MPI_Win` defines the group of process allowed to access a certain memory area
- Arguments:
  - `base`: Starting address for the 'open' memory area
  - `size`: size of the 'open' memory area in bytes
  - `disp_unit`: offset from the base in bytes
  - `info`: Hint to the MPI how the window will be used (e.g. only reading or only writing)
  - `comm`: communicator defining the group of processes allowed to access the memory window
- A window does not define, when somebody is allowed to access data items in the memory window

Parallel Software Technologies Laboratory PSTL

The window concept of MPI-2 (II)

- Definition of a spatial window:
  - *Access Epoch*: time slot in which a process accesses remote memory of another process
  - *Exposure Epoch*: time slot in which a process allows access to a memory window by other processes
- Question: does a process have the possibility to control when other processes are accessing his memory window?
  - yes: *active target communication*
  - no: *passive target communication*

COSC 6374 – Parallel Computation
Edgar Gabriel

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

Active Target Communication (I)

```
MPI_Win_fence ( int assert, MPI_Win win);
```

- Synchronization of all operations within a window
  - collectiv across all processes of `win`
  - No difference between *access* and *exposure epoch*
  - If an access or exposure epoch is alredy initated, the subsequent call to `Win_fence` closes this epoch, else it start new epoch
- Arguments
  - `assert`: Hint to the library on the usage (default: 0)

PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

Data exchange (I)

```
MPI_Put (void *oaddr, int ocount, MPI_Datatype
        otype, int rank, MPI_Aint disp, int tcount,
        MPI_Datatype ttype, MPI_Win win);
```

- This function "puts" data described by the triple (`oaddr, ocount, otype`) into the main memory of the process defined by Rank `rank` in the window `win` at the position (`base+disp*disp_unit,tcount,ttype`)
  - `base` and `disp_unit` have been defined in `MPI_Win_create`
- A single process controls the data parameters of both processes

Data exchange (II)

```
MPI_Get (void *oaddr, int ocount, MPI_Datatype
        otype, int rank, MPI_Aint disp, int tcount,
        MPI_Datatype ttype, MPI_Win win);
```

- This function copies data items described by the triple (`base+disp*disp_unit, tcount, ttype`) from the main memory of the process defined by Rank `rank` in the window `win` to the position (`oaddr,ocount,otype`) in its own main memory
  - `base` and `disp_unit` defined in `MPI_Win_create`

Example

```
MPI_Win win;
int i, tmp, size, *rank;

MPI_Comm_rank (MPI_COMM_WORLD, &tmp);
MPI_Comm_size (MPI_COMM_WORLD, &size);
rank = (int *) malloc ( sizeof(int) * size);

MPI_Win_create (rank, size*sizeof(int), sizeof(int),
                MPI_INFO_NULL, MPI_COMM_WORLD, &win);

MPI_Win_fence (0, win);
for ( i=0; i < size; i++ ) {
       MPI_Put (&tmp, 1, MPI_INT, i, tmp, 1, MPI_INT,
                win);
}
MPI_Win_fence (0, win);

/* Now rank[0] is 0 ,rank[1]=1,…,
   rank[size-1]=size-1 on all processes */
```

Comments to the example

- According to the specification, the modifications of the data items might only be visible after closing the according epochs
  - No guarantee whether the data item is really moved during `MPI_Put` or during `MPI_Win_fence`
- If multiple processes access the very same memory address at the very same process, MPI does not make any guarantees which data item will be visible.
  - Responsibility of the user to get it right

Active Target Communication – (II)

```
MPI_Win_start (MPI_Group group, int assert,
               MPI_Win win);
MPI_Win_complete (MPI_Win win);
```

- Opens respectively closes an *access epochs*
- Only processes being part of the group `group` are allowed to access the memory of another process

```
MPI_Win_post (MPI_Group group, int assert, MPI_Win win);
MPI_Win_wait (MPI_Win win);
MPI_Win_test (MPI_Win win);
```

- Opens respectively closes an *exposure epochs*
- Only processes being part of the group `group` allow access to its main memory
- `MPI_Win_post` is non-blocking

Example

```
MPI_Win win;
MPI_Group group;
int i, tmp, size, *rank;

MPI_Comm_rank (MPI_COMM_WORLD, &tmp);
MPI_Comm_size (MPI_COMM_WORLD, &size);
rank = (int *) malloc ( sizeof(int) * size);

MPI_Win_create (rank, size*sizeof(int), sizeof(int),
                MPI_INFO_NULL, MPI_COMM_WORLD, &win);
MPI_Win_group (win, &group);

MPI_Win_post (group, 0, win);
MPI_Win_start (group, 0, win);
for ( i=0; i < size; i++ ) {
       MPI_Put (&tmp, 1, MPI_INT, i, tmp, 1, MPI_INT,
                win);
}
MPI_Win_wait (win);
MPI_Win_complete (win);
```

Passive Target Communication

```
MPI_Win_lock (int lock_type, int rank, int assert,
              MPI_Win win);
MPI_Win_unlock (int rank, MPI_Win win);
```

- `MPI_Win_lock` starts an *access epoch* in order to access the main memory of the process with rank `rank`
- `lock_type: MPI_LOCK_EXCLUSIVE` or `MPI_LOCK_SHARED`

Parallel Software Technologies Laboratory PSTL

One-sided vs. Two-sided communication

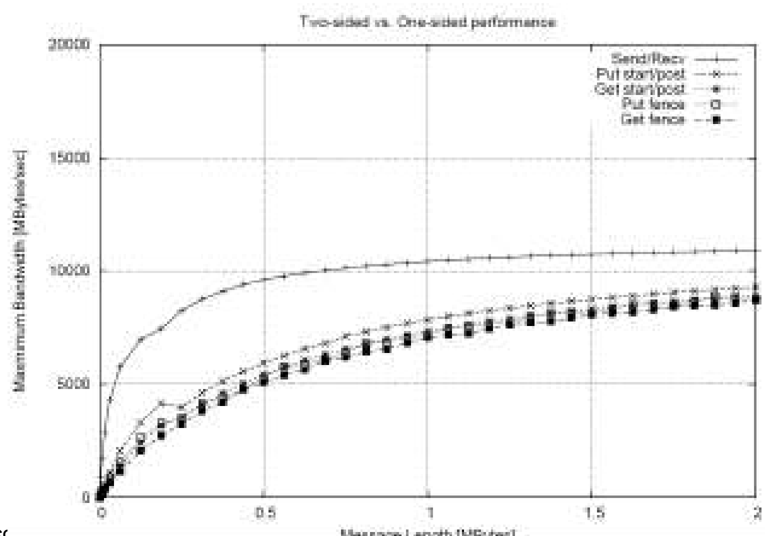
- One-sided communication doesn't need
  - *message matching*
  - *unexpected message queues*
  - Burdens (theoretically) only one processor

  ➔ potentially faster!

- One-sided communication in MPI can optimize potentially
  - multiple transactions
  - between multiple processeces

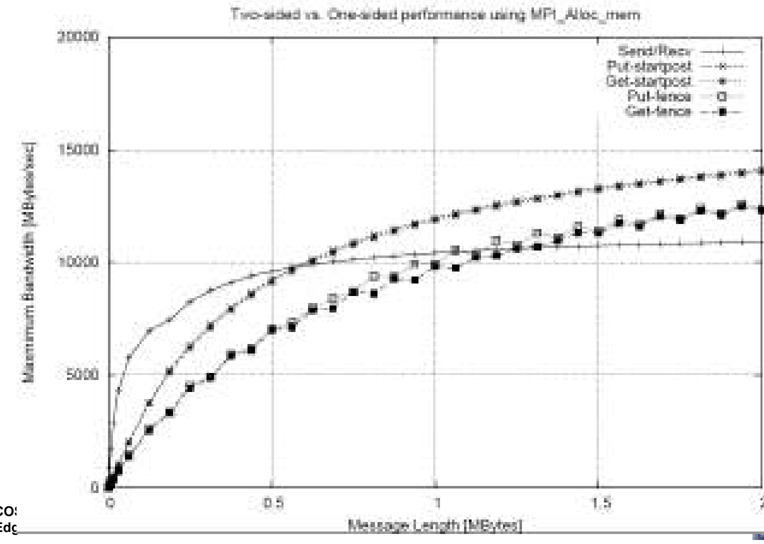
Parallel Software Technologies Laboratory PSTL

NEC SX6 intra-node performance



PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

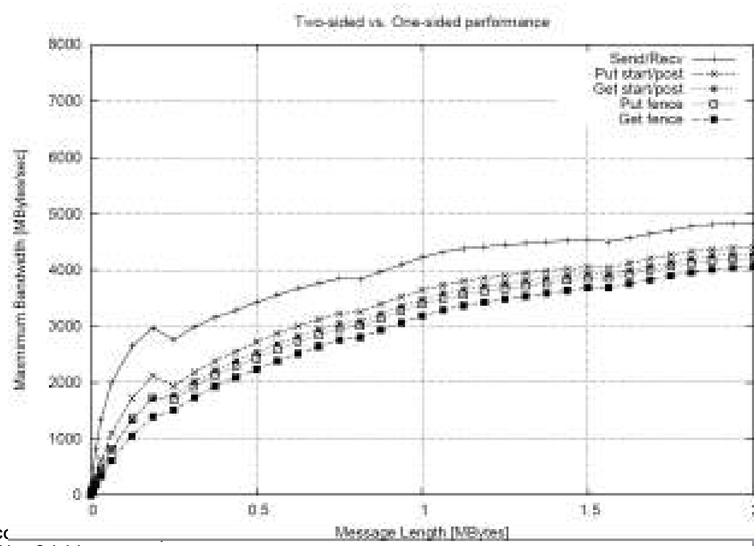
NEC SX6 intra-node performance (II)



CO
Edg

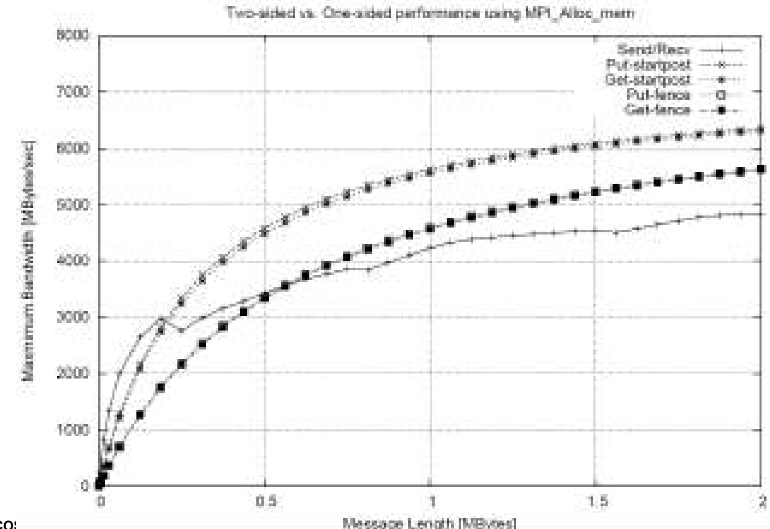
PARALLEL SOFTWARE TECHNOLOGIES LABORATORY PSTL

NEC SX6 inter-node performance



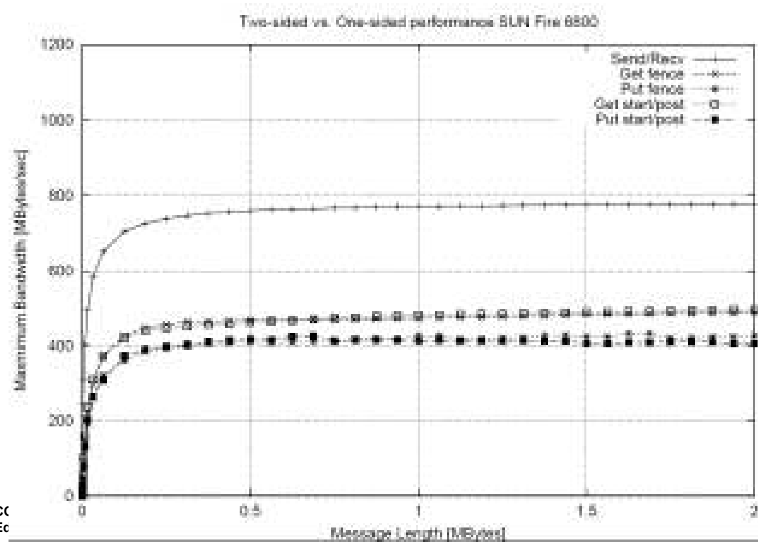
CC
Edgar Gabriel

NEC SX6 inter-node performance (II)



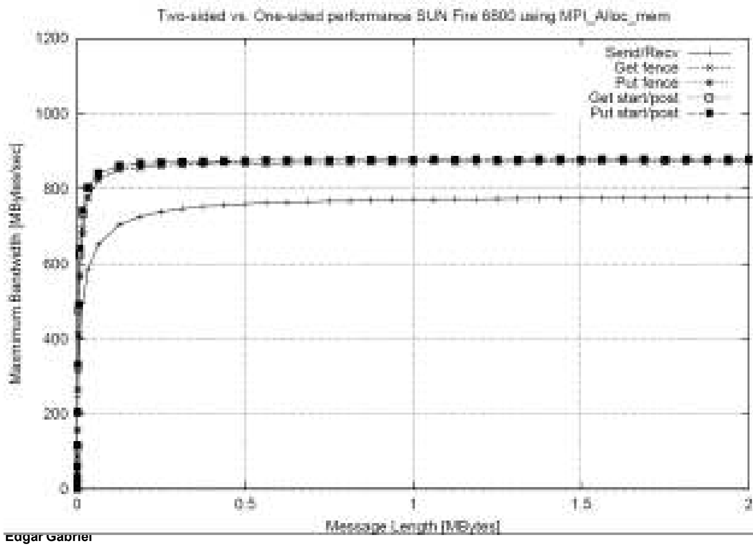
Parallel Software Technologies Laboratory PSTL

Sun Fire 6800 performance



Parallel Software Technologies Laboratory PSTL

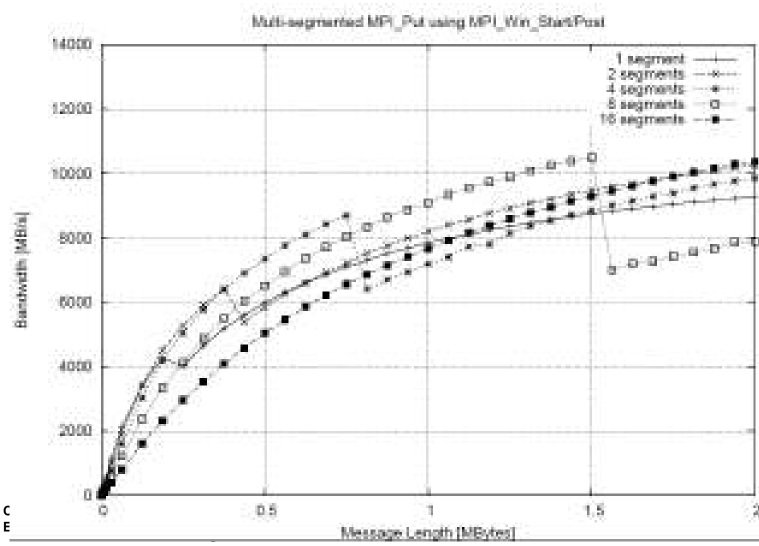
Sun Fire 6800 performance (II)



Edgar Gabriel

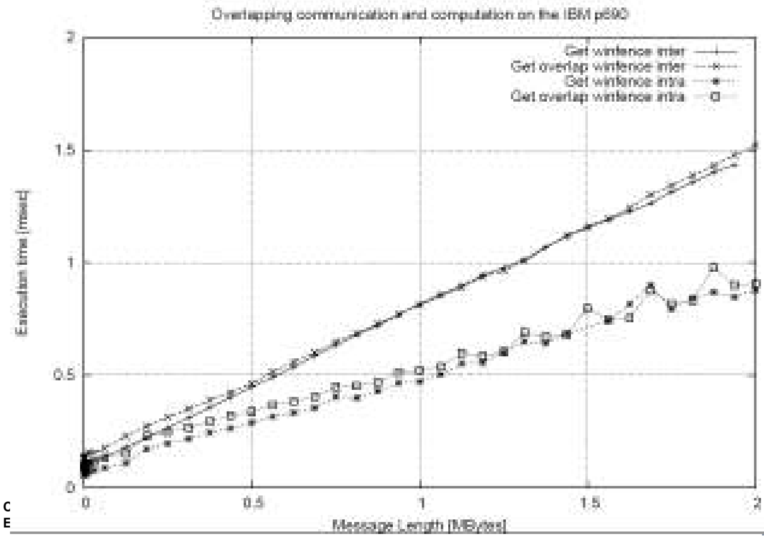
Parallel Software Technologies Laboratory PSTL

NEC-SX6 intra-node performance (III)



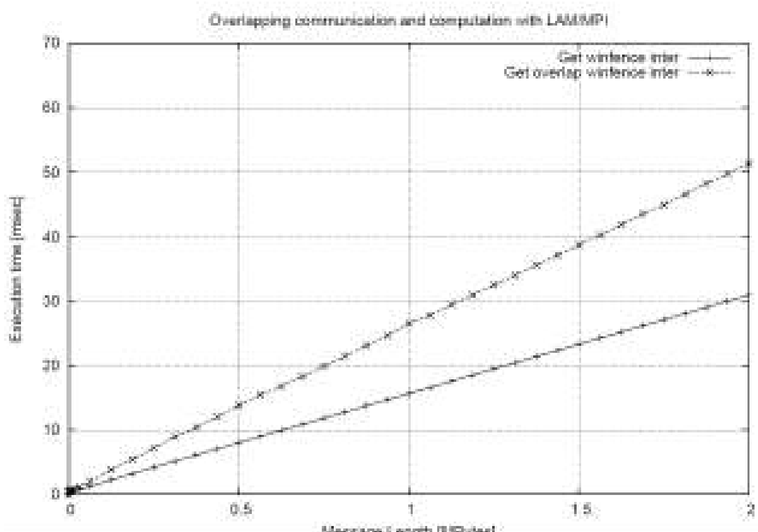
Parallel Software Technologies Laboratory PSTL

Performance IBM p690



Parallel Software Technologies Laboratory PSTL

Performance on a GEthernet cluster



Edgar Gabriel