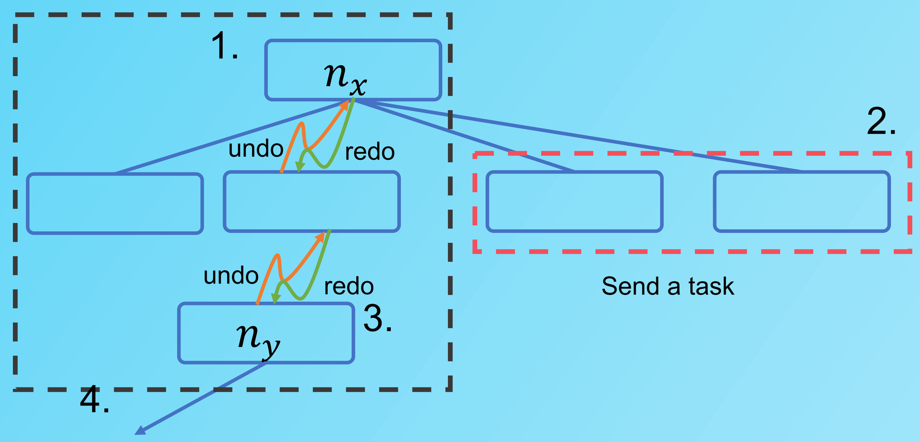
Implementing the Tascell Task-Parallel Language Tascell Using Multithreaded MPI

Daiki Kojima[1), Tasuku Hiraishi[2), Hiroshi Nakashima[2), Masahiro Yasugi[3)

1) Graduate School of Informatics, Kyoto University 2) Academic Center for Computing and Media Studies, Kyoto University 3) Department of Artificial Intelligence, Kyushu Institute of Technology

The Tascell Language

- Tascell
  - Extended C language that achieves high performance in irregular applications [T. Hiraishi et al., PPoPP 2009]
  - A worker executes its own task sequentially and does not create tasks until it receives task requests
  - When a worker (victim) receives a task request from another worker (thief),
    1. it temporarily backtracks to the past state
    2. spawns a task and sends it to the thief worker
    3. returns from the backtracking
    4. resumes its own task
  - A worker can delay copying workspaces and reuse it
  - Supports distributed memory environments using TCP/IP or MPI

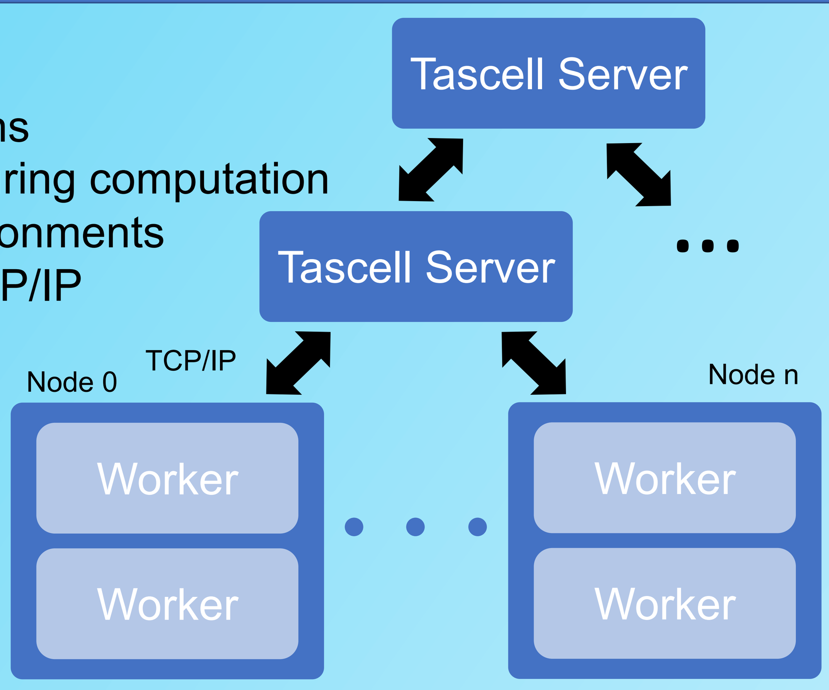


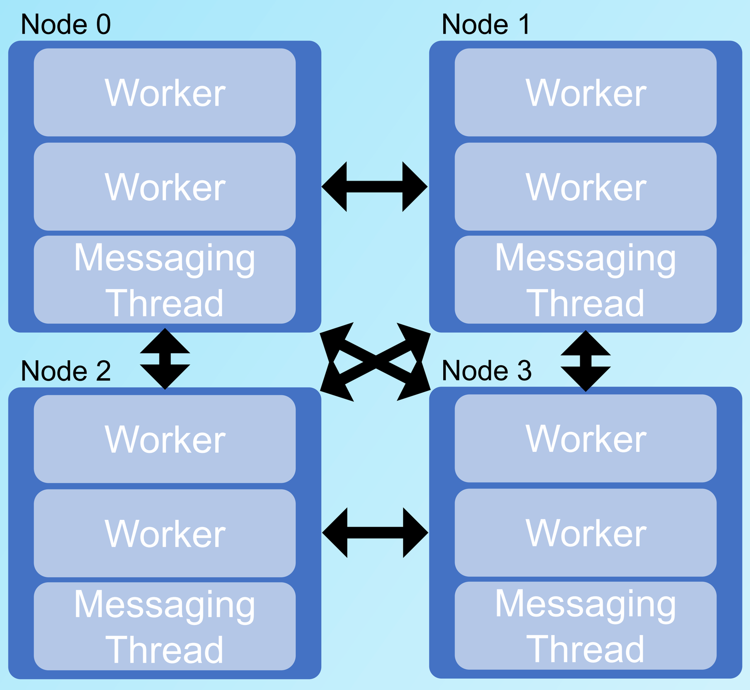
Implementation using Singlethreaded MPI

- Implementation using MPI with the MPI_THREAD_FUNNELED support [D. Muraoka et al., P2S2 2016]
- Computation nodes communicate directly with other nodes (serverless implementation)
- Each node employs a *messaging thread*
  - The messaging thread iterates the following operations
    1. checks an incoming message using MPI_Iprobe() and receives it using MPI_Recv()
    2. sleeps t_{slp} msec
    3. If the previous MPI_Isend() has finished, checks an incoming message in the *request queue* and sends it with MPI_Isend()
- A worker thread asks the messaging thread to send a message by adding it to the *request queue*
- Pros: works using MPI only with the MPI_THREAD_FUNNELED support
- Cons: a messaging thread uses busy-waiting for waiting both incoming and outgoing messages

Implementation using TCP/IP

- Each node is connected to *Tascell Server*
- Tascell Servers relay inter-node communications
- Pros: new computation nodes can be added during computation
- Pros: supports widely distributed memory environments
- Cons: supercomputers often do not support TCP/IP
- Cons: Tascell servers can become bottlenecks



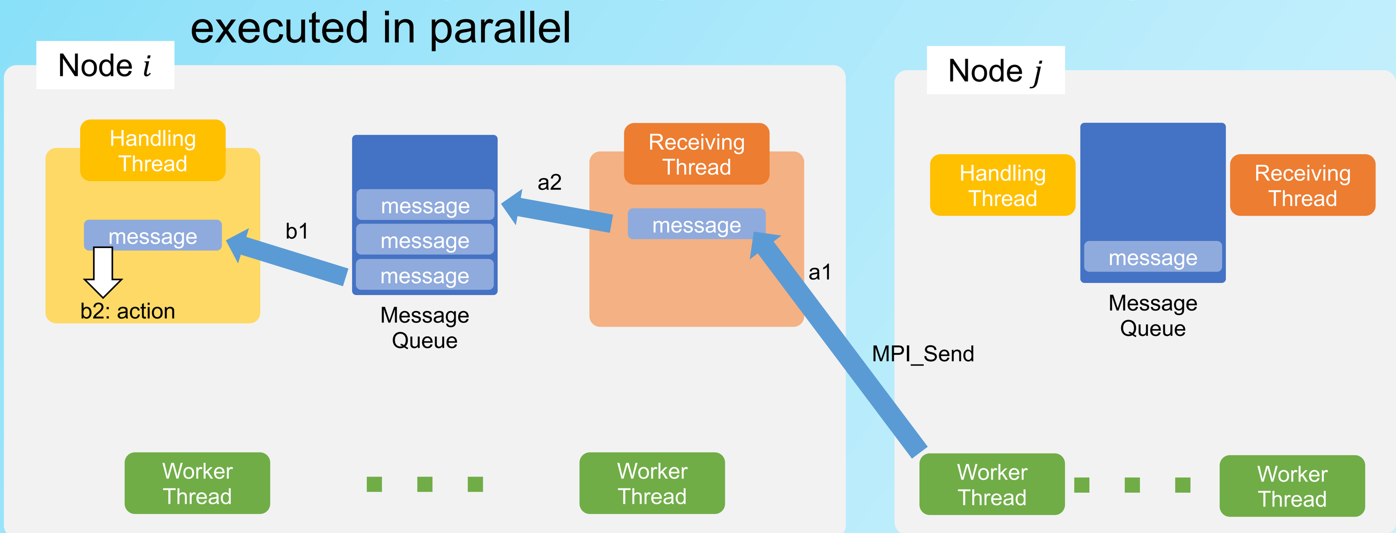


Psuedo code for the messaging thread

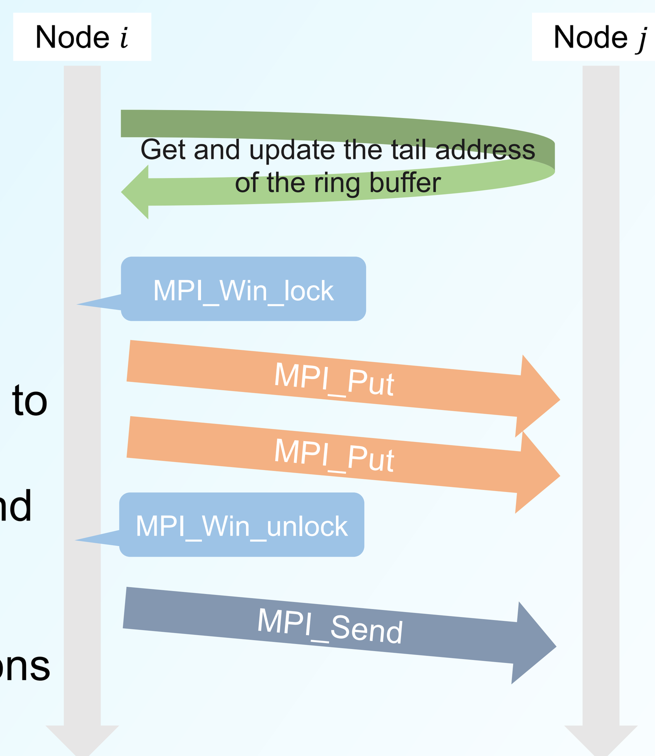
```
for(;;) {
  MPI_Iprobe(...);
  if(/* any incoming messages? */) {
    MPI_Recv(...);
    perform an action specified by the message
  }
  sleep(t_slp);
  if(sending_message) {
    MPI_Test(...);
    if(/* is previous MPI_Isend finished? */) {
      sending_message = false;
    }
  } else {
    if(/* any entries in the send queue */) {
      dequeue an entry
      MPI_Isend(...);
      sending_message = true;
    }
  }
}
```

Implementations using Multithreaded MPI

- Implementation using MPI with the MPI_THREAD_MULTIPLE support and two-sided communications
  - A worker thread sends messages directly to another node using MPI_Send()
  - Each node employs the two service threads:
    - The *receiving thread*
    - a1. waits an incoming message using MPI_Probe(), and receives it using MPI_Recv(), and
      a2. adds the received message to the *message queue*
    - The *handling thread*
      b1. takes a message from the message queue and
      b2. performs an action specified by the message
  - We cannot let the messaging thread perform actions, because that can result in deadlock if the thread sends a new message during the action
  - Pros: Busy-waiting free implementation
  - Pros: the delay for sending messages can be reduced
  - Pros: message receiving and actions for messages can be executed in parallel

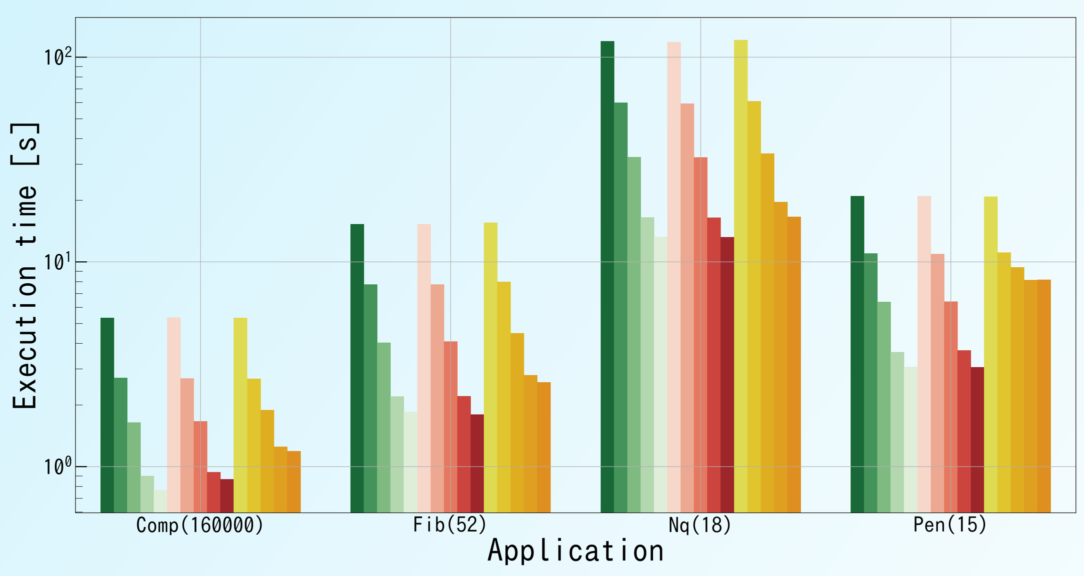


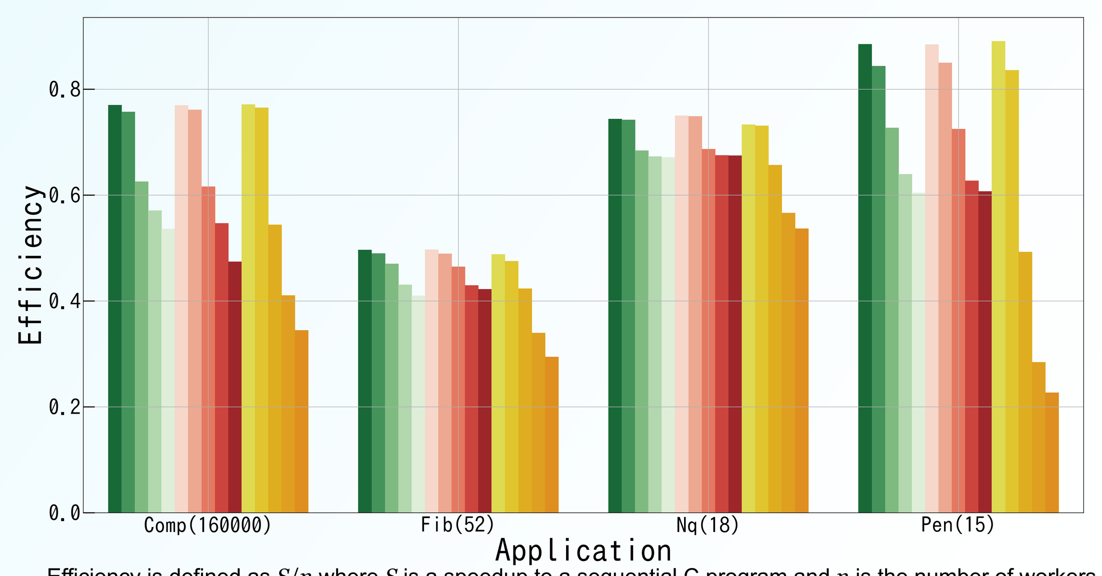
- Implementation using MPI with the MPI_THREAD_MULTIPLE support and one-sided communications
  - Each computation node has a *ring buffer*, to which workers in external nodes put messages
  - Each node employs two service threads:
    - The *handling thread* takes received messages from the *tail* of the ring buffer and performs actions specified by the messages
    - The *notification thread* waits for notifications from workers in external nodes using MPI_Recv() and notifies the handling thread that there are incoming messages
  - A worker thread performs the following operations when sending a message
    1. gets and updates the *tail* of the ring buffer using MPI_Get_accumulate()
    2. acquires the lock of the ring buffer using MPI_Win_lock()
    3. sends the msssage using MPI_Put() calls
    4. releases the lock of the ring buffer using MPI_Win_unlock()
    5. sends a notification to the notification thread of the recipient node using MPI_Send()
  - Pros: redundant memory copy operations can be eliminated
    - In the implementations using two-sided communications, a worker needs to pack an outgoing message into a buffer before sending it
      - because structures of sending data are defined in Tascell programs and not statically fixed. It is tough to send such data using two-sided communications without packing
    - In the implementation using one-sided communications, packing operations are not necessary because such data can be sent directly using multiple MPI_Put() calls
  - Cons: the number of MPI communications per message increases



Performance Evaluatinos

- Performance on Xeon Cluster
  - CPU: Xeon Broadwell 2.1GHz 18-core x 2 (36 workers / node)
    Interconnect: Omni-Path (injection BW = 12GB/s)
    Memory: 128GB, Intel Compiler 17.0.6, Intel MPI 2017.4 (-O2)
  - Applications:
    - Fib: recursively computes the n-th Fibonacci number
    - Nq: finds all solutions to the n-queens problem
    - Pen: finds all solutions to the Pentomino problem
    - Comp: compares array elements a_i and b_j in $0 \le i, j < n$
  - Results:
    - The implementation using the MPI_THREAD_MULTIPLE support and two-sided communications shows slightly better performance than the MPI_THREAD_FUNNELED based implementation except for Comp, probably due to shorter communication delays
    - The implementation using one-sided communications shows the worst performance in almost all the measurements. However, it shows the best performance in the 2-node executions of Comp, probably due to higher communication throughput when sending large array data





Efficiency is defined as S/n where S is a speedup to a sequential C program and n is the number of workers. (Efficiency = 1 means an ideal speedup.)