An MPI-based Implementation of the Tascell Task-Parallel Programming Language

Daisuke Muraoka
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology
Masahiro Yasugi
Department of Artificial Intelligence, Kyushu Institute of Technology
Tasuku Hiraishi
Academic Center for Computing and Media Studies, Kyoto University
Seiji Umatani
Graduate School of Infomatics, Kyoto University

Abstract—**Tascell is a task parallel language that supports distributed memory environments. The conventional implementation of Tascell realizes inter-node communication with TCP/IP communication via Tascell servers. This implementation is suitable for dynamic addition of computation nodes and wide-area distributed environments. On the other hand, in supercomputer environments, TCP/IP may not be available for inter-node communication and there may be no appropriate places for deploying Tascell servers. In this study, we developed a server-less implementation of Tascell that realizes inter-node communication with MPI communication, while allowing highly portable Tascell programming by abstracting the underlying communication layer. Our server-less implementation realizes deadlock freedom, although it only requires the two-sided communication paradigm and the MPI_THREAD_FUNNELED support level. Our implementation exhibits good performance on four Xeon Phi coprocessors and the K computer.**

I. Introduction

With the growing popularity of parallel computing environments including multicore processors, high productivity languages for parallel computing have become important. Cilk [1] is such a language, which achieves good load balancing for many applications including irregular applications such as backtrack search. Cilk keeps all workers busy by generating plenty of "logical" threads and adopting the oldest-first strategy.

On the other hand, we proposed a logical thread-free framework called Tascell [2], [3]. A Tascell worker spawns a real task only when requested by another idle worker. The worker spawns a task after restoring its oldest task-spawnable state by temporarily backtracking. This mechanism eliminates the cost of spawning/managing logical threads. It also promotes the reuse of workspaces and improves the locality of reference since it does not need to prepare a workspace for each concurrently runnable logical thread. Furthermore, a single Tascell program can run efficiently on shared and distributed memory environments. In fact, we evaluated the performance of Tascell and confirmed that we can obtain reasonable speedups for irregular applications on computer cluster systems [2], [4].

In the conventional implementation of Tascell, inter-node communication is realized by TCP/IP communication via message routing servers called Tascell servers. This implementation is suitable for dynamic addition of computation nodes and wide-area distributed environments. On the other hand, Tascell servers often become communication bottlenecks. Furthermore, in recent supercomputer environments, there may be no appropriate places for deploying Tascell servers, and TCP/IP may not be available for inter-node communication; it is hard or impossible to run the conventional implementation in such an environment.

Therefore, we implemented inter-node communication in Tascell using MPI, which is supported by most practical supercomputer systems. At the same time, we adopted a Tascell server-less implementation in order to overcome the deployment and bottleneck problems, excluding the support of wide-area distributed environments. Note that programmers can write Tascell programs without concern for the underlying communication layer.

In order to enable our implementation to work with many MPI implementations, it only requires the `MPI_THREAD_FUNNELED` support level, in which only the main thread can make MPI calls, and the two-sided communication paradigm. Due to these limitations, techniques are required to implement efficient inter-node communication avoiding deadlocks.

We evaluated the performance of our MPI-based implementation on four Xeon Phi coprocessors and the K computer. On Xeon Phi coprocessors, we also compared its performance with those of the conventional TCP/IP-based implementation and an TCP/IP based implementation that does not use Tascell servers. Note that TCP/IP is not available for inter-node communication in the K computer.

The remainder of this paper is organized as follows. First, we provide a brief description of the Tascell framework in Section II. Then, we explain our MPI-based implementation of Tascell in Section III. In Section IV, we show the performance evaluations. Finally, we conclude this paper in Section VI.

II. Tascell framework

This section provides a brief description of the Tascell framework in the conventional TCP/IP-based implementation.

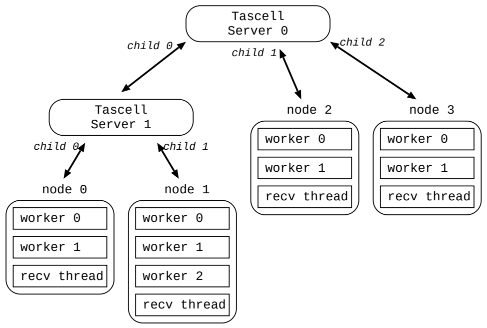


Fig. 1. Inter-node connection in the conventional Tascell implementation.

Note that the framework remains basically the same also in the proposed MPI-based implementation except that inter-node communication is done by using MPI and without Tascell servers.

The Tascell framework consists of a Tascell server and a compiler for the Tascell language.

Compiled Tascell programs are executed on one or more computation nodes. As shown in Fig. 1, each computation node has one or more workers in the shared memory environment and is TCP/IP-connected to a Tascell server.

For good load balancing, idle workers should request tasks of loaded workers. An idle worker sends a task request to either a specific worker or any worker. Intra-node (Inter-node) messages are relayed by Tascell runtime systems (Tascell servers), which choose loaded workers (nodes) for "to any" task requests. Such a series of messages is exchanged automatically; programmers need not (and cannot) treat each message directly.

Each task or its result is transferred as a task object whose structure is defined in a Tascell program. If a request is from the same node, (the pointer to) the object can be passed quickly via shared memory, otherwise the object is transferred as a serialized message via Tascell servers.

As shown in Fig. 1, a Tascell server can be connectd to another Tascell server. Thus, computation nodes and Tascell servers generally form a tree. This capability of the conventional implementation enables us to run a Tascell program in a widely distributed environment [4].

III. Proposed implementation

In this section, we present the details of our MPI-based implementation of Tascell. As mentioned in Section I, we adopted a Tascell server-less implementation, that is, computation nodes directly transfer messages to each other as shown in Fig. 2. In addition, we decided to require only the `MPI_THREAD_FUNNELED` support level and the two-sided communication paradigm. Therefore, we cannot simply replace `send`/`recv` system calls in the conventional implementation to `MPI_Send`/`MPI_Recv` MPI calls, as described below.

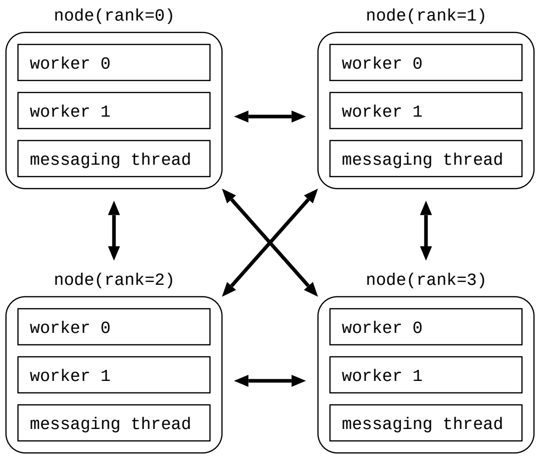


Fig. 2. Inter-node connection in the proposed Tascell implementation.

A. Node addressing

In the conventional implementation, the destination when sending a message to an external node is specified by a relative address that denotes a path from the source to the destination. In the proposed implementation, the destination node is directly specified by the rank of the MPI process that runs on the node. In addition, an actual destination node of a "to any" task request is determined randomly in the sender computation node.

B. Deadlock avoidance

In general, a program using inter-node communication can result in deadlocks when a blocking communication routine needs to wait for completion of another blocking routine [5]. Also in Tascell, careless implementation of message communication, for example, simply letting a messaging thread send and receive a message repeatedly using blocking communication routines, can result in deadlocks. Moreover, it is difficult to statically resolve such wait-for dependencies because a Tascell worker can request a task to any anther worker at any time to perform work-stealing-based dynamic load balancing.

The conventional implementation avoids such deadlocks by letting a Tascell server have sender and receiver threads for each connected node, i.e., a computation node or another Tascell server, so that any blocking communication is done asynchronously.

However, we cannot apply this implementation scheme to our MPI-based implementation since letting a computation node have sender and receiver threads for each external node requires the `MPI_THREAD_MULTIPLE` support level.

In order to avoid deadlocks using only a single messaging thread, we let it send a message using the non-blocking function and call the MPI receive function after confirming that there are incoming messages using the `MPI_Iprobe` a non-blocking MPI function. We describe the details below.

In our implementation, a computation node has one or more worker threads and a messaging thread, which is responsible

```
for (;;)
{
  // Check whether there are any incoming messages without blocking.
  MPI_Iprobe(...)
  if ( any incoming messages? )
  {
    MPI_Recv(...) // Receive a message.
  }
  sleep(t_slp);
  if ( sending_message = true )
  {
    MPI_Test(...)
    if ( Is a previous sending operation complete? )
    {
      sending_message = false;
    }
  }
  else
  {
    if ( any entries in the send request queue? )
    {
      // Send a message using a nonblocking function.
      MPI_Isend(...)
      sending_message = true
    }
  }
}
```

Fig. 3. Flow of processing of the messaging thread

for communications with external nodes. Fig. 3 shows pseudo code for the messaging thread. The messaging thread repeatedly executes the following steps:

1) checks whether there are any incoming messages using MPI_Iprobe, and if any, receives a message using MPI_Recv,
2) sleeps for a specified time, and
3) checks whether a previous send operation is complete, and if so, pops an entry from the send request queue and sends the message in it using MPI_Isend.

Note that a worker thread can request the messaging thread to send a message to an external node by adding an entry to the send request queue.

This implementation can avoid deadlocks because it sends a message using a non-blocking function and starts a receive operation only after confirming that there is a corresponding send request using a non-blocking function.

One obvious problem with this implementation is that it uses busy-waiting for incoming or outgoing messages. However, we cannot help use busy-waiting in order to avoid deadlocks without requiring the MPI_THREAD_MULTIPLE support level for an underlying MPI implementation. As discussed below, busy-waiting is unavoidable even when we require the MPI_THREAD_SERIALIZED support level, which allows any thread to make MPI calls but only one at time (weaker than MPI_THREAD_MULTIPLE but stronger than MPI_THREAD_FUNNELED).

Suppose an implementation that uses a send request queue as the proposed implementation. In order to avoid busy-waiting, we need to use a blocking operation that can wait for completion of send/receive operations and send requests from worker threads at the same time. The messaging thread can wait for completion of send and receive operations at the same time using the MPI_Waitany blocking function, but there is no way to wait for these operations and addition of an entry to the send request queue at the same time.

In an implementation that does not use without a send request queue, a worker thread needs to send messages by itself by making MPI calls. In this case, the messaging thread would be responsible at least for receiving messages from external nodes. Such an implementation can result in deadlocks because, due to the limitation of MPI_THREAD_SERIALIZED, a send operation by a worker thread is blocked while the messaging thread waits for completion of receive operations using a blocking function.

	Xeon Phi	The K computer (in 1 node)
Host processor	Intel Xeon E5-2697 v2 12-core $\times$ 2	SPARC64 VIIIfx 2GHz 8-core
Co-processor	Intel Xeon Phi 3120P 57-core $\times$ 4	-
Memory	64GB	16GB
Co-processor Memory	6GB for each coprocessor	-
OS	CentOS 6.5 (64bit)	-
Compiler	Intel Compiler 13.1.3 with -O2 optimizers	Fujitsu C/C++ Compiler 1.2.0 with -O2 optimizers
Closure	Trampoline-based implementation (compatible with the GCC extension [6])	LW-SC [7]
MPI library	Intel MPI 4.1 Update 3 Build 20140124	-
Tascell server	Steel Bank Common Lisp 1.2.6 (runs on the host node)	-

TABLE I
EVALUATION ENVIRONMENT

IV. Performance evaluation

We evaluated the performance of the proposed implementation of Tascell on Xeon Phi and the K computer using the following programs:

Fib(n) recursively computes the n-th Fibonacci number.
Nq(n) finds all solutions to the n-queens problem.
LU(n) computes the LU decomposition of an $n \times n$ matrix using a cache-oblivious recursive algorithm.
Pen(n) finds all solutions to the Pentomino problem with n pieces (using additional pieces and an expanded board for $n > 12$).

The evaluation environments are summarized in Table I. The check interval for incoming and outgoing messages (t_slp in Fig. 3) is set to $20\,\mu s$.

A. Xeon Phi coprocessors

On Xeon Phi coprocessors, we compared the performance of the proposed implementation with those of the conventional implementation, an implementation that exploits the MPI_THREAD_MULTIPLE support, and a Tascell server-less implementation that uses TCP/IP for inter-node communication. The five implementations are summarized as follows.

Tascell/SVR is the conventional implementation. Inter-node communication is done by using TCP/IP via a Tascell-server.

Tascell/SVR-Noisy is Tascell/SVR in which the capability of the Tascell server to avoid sending a task request to nodes that do not have any tasks is disabled.

Tascell/MPI is the proposed implementation. Inter-node communication is done by using MPI without Tascell servers.

Tascell/MPI-MT is an MPI-based implementation (without Tascell servers) that exploits the `MPI_THREAD_MULTIPLE` support.

Tascell/TCP is an implementation in which inter-node communication is done by using TCP/IP without Tascell servers.

In the Tascell/TCP implementation, a computation node has a receiver thread, which waits for incoming messages using the `poll` system call and receives a message using the `recv` system call. In addition, each worker thread sends a message using the `send` system call without using a send request queue.

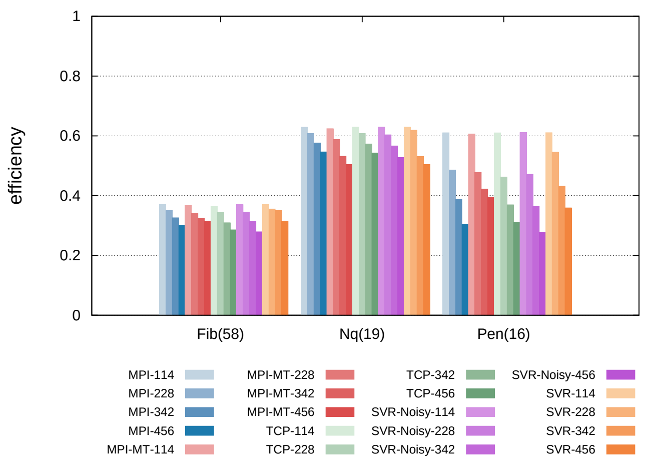
In the Tascell/MPI-MT implementation, a computation node has a receiver thread, which receives a message using the `MPI_Recv`. In addition, each worker thread sends a message using the `MPI_Send` without using a send request queue.

Each Xeon Phi 3120P coprocessor has 57 cores and each core can run four hardware threads. We only show the measurement results when each core runs two hardware threads (i.e., each coprocessor runs 114 hardware threads) because we could get the best performance with this setting.

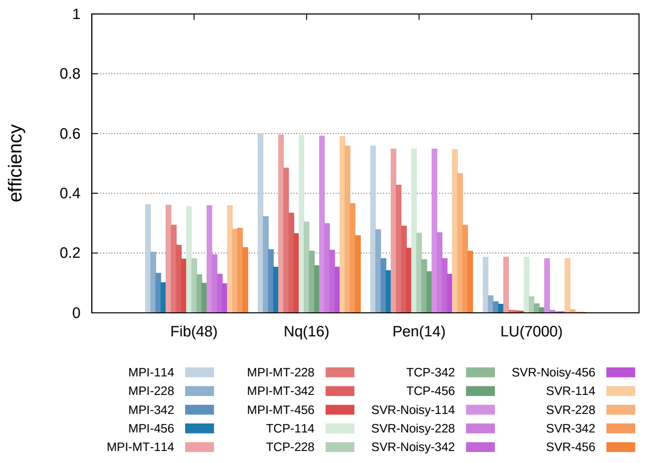
The results of the performance measurements are shown in Table III, Fig. 4. In Table III, t_{C} denotes the execution time of the sequential C programs. $t_{\mathrm{M}n}$, $t_{\mathrm{P}n}$, $t_{\mathrm{T}n}$, $t_{\mathrm{N}n}$ and $t_{\mathrm{S}n}$ denote the execution time of Tascell/MPI, Tascell/MPI-MT, Tascell/TCP, Tascell/SVR-Noisy and Tascell/SVR with n workers (114 workers in each coprocessor), respectively. In Table II, NONE/TASK denotes the average time between sending a task request and receiving a NONE/TASK message as a response. A worker that receives a task request (victim) generates and sends a TASK message to send a spawned task. If the victim cannot spawn a task, it sends a NONE message as a response. NONE+TASK denotes the average time between sending a task request and receiving a NONE or TASK message.

We can see from Table III that there is little difference among the performances of the five implementations with 114 workers because they perform almost the same computation in shared memory environments except that there is a messaging thread that uses busy-waiting in Tascell/MPI; we can conclude that the cost of the busy-waiting is ignorable.

From Fig. 4, we can see that we can get good speedups with all the implementations show good speedups for Fib(58), Nq(19) and Pen(16) when the number of coprocessors increases with four coprocessors. This is because the total loads for Fib(58), Nq(19) and Pen(16) are sufficiently large and their computation times are sufficiently long relative to the communication costs among coprocessors. On the other hand, speedups for Fib(48), Nq(16) and Pen(14) are limited because the total loads for these benchmarks are too small.



(a) Large size benchmarks



(b) Small size benchmarks

Fig. 4. Efficiency with multiple nodes. Efficiency is defined as $S/worker$ where S is speedup of a sequential C program and *worker* is the number of workers. (Efficiency = 1 means an ideal speedup.)

Although we cannot get speedups for LU(7000) with all the implementations when the number of coprocessors increases with four coprocessors, Tascell/TCP and Tascell/MPI show better performances than Tascell/MPI-MT, Tascell/SVR-Noisy and Tascell/SVR. This is because coprocessors in the Tascell/MPI and Tascell/TCP executions directly communicate with each other without Tascell servers. In Tascell/SVR and Tascell/SVR-Noisy, a Tascell server becomes a bottleneck when messages of large size are transferred among coprocessors. Note that it is difficult to obtain sufficient speedups in applications with large shared data such as LU with work-stealing-based dynamic load balancing in distributed memory environments.

From the comparison between $t_{\mathrm{M}456}$ and $t_{\mathrm{T}456}$ in Table III, we can see that Tascell/MPI shows a little better performance than Tascell/TCP for all the benchmarks. This difference reflects the difference of the performances between MPI and TCP/IP communication.

	Tascell/MPI			Tascell/MPI-MT			Tascell/TCP			Tascell/SVR		
Return message	NONE	TASK	NONE+TASK	NONE	TASK	NONE+TASK	NONE	TASK	NONE+TASK	NONE	TASK	NONE+TASK
Fib(48)	68.22ms	70.13ms	68.99ms	3.45ms	2.38ms	2.63ms	49.13ms	50.92ms	49.75ms	52.49ms	186.77ms	112.44ms
Nq(16)	65.53ms	69.18ms	66.76ms	2.85ms	4.39ms	3.60ms	51.66ms	53.48ms	52.27ms	49.31ms	142.40ms	90.18ms
Pen(14)	63.68ms	80.89ms	68.79ms	2.46ms	3.55ms	3.03ms	45.32ms	54.45ms	48.30ms	43.77ms	130.51ms	78.32ms
LU(7000)	119.97ms	313.75ms	146.56ms	43.94ms	141.07ms	66.11ms	70.86ms	333.20ms	118.74ms	134.80ms	289.74ms	180.28ms

TABLE II
THE AVERAGE RESPONSE TIME TO TASK REQUESTS ON XEON PHI.

	C	Tascell/MPI				Tascell/MPI-MT				Tascell/TCP			
# of workers	1	114	456	speedup		114	456	speedup		114	456	speedup	
	t_{C}[s]	$t_{\mathrm{M}114}$[s]	$t_{\mathrm{M}456}$[s]	$t_{\mathrm{C}}/t_{\mathrm{M}456}$	$t_{\mathrm{M}114}/t_{\mathrm{M}456}$	$t_{\mathrm{P}114}$[s]	$t_{\mathrm{P}456}$[s]	$t_{\mathrm{C}}/t_{\mathrm{P}456}$	$t_{\mathrm{P}114}/t_{\mathrm{P}456}$	$t_{\mathrm{T}114}$[s]	$t_{\mathrm{T}456}$[s]	$t_{\mathrm{C}}/t_{\mathrm{T}456}$	$t_{\mathrm{T}114}/t_{\mathrm{T}456}$
Fib(48)	218.67	5.29	4.76	45.94	1.11	5.33	2.66	82.21	2.00	5.40	4.86	44.99	1.11
Fib(58)	26897.45	637.80	196.86	136.63	3.24	642.72	187.82	143.21	3.42	647.63	206.49	130.26	3.14
Nq(16)	190.90	2.79	2.75	69.42	1.01	2.81	1.58	120.82	1.78	2.82	2.66	71.77	1.06
Nq(19)	79434.11	1107.71	318.76	249.20	3.48	1116.00	345.33	230.02	3.23	1107.34	320.91	247.53	3.45
Pen(14)	239.47	3.76	3.70	64.72	1.02	3.83	2.42	98.95	1.58	3.83	3.82	62.69	1.00
Pen(16)	24337.73	349.65	175.56	138.63	1.99	352.11	135.07	180.19	2.61	349.85	171.89	141.59	2.04
LU(7000)	276.58	13.01	21.20	13.05	0.61	12.97	98.26	2.81	0.13	13.07	34.32	8.06	0.38

	Tascell/SVR-Noisy				Tascell/SVR			
# of workers	114	456	speedup		114	456	speedup	
	$t_{\mathrm{N}114}$[s]	$t_{\mathrm{N}456}$[s]	$t_{\mathrm{C}}/t_{\mathrm{N}456}$	$t_{\mathrm{N}114}/t_{\mathrm{N}456}$	$t_{\mathrm{S}114}$[s]	$t_{\mathrm{S}456}$[s]	$t_{\mathrm{C}}/t_{\mathrm{S}456}$	$t_{\mathrm{S}114}/t_{\mathrm{S}456}$
Fib(48)	5.34	4.94	44.27	1.08	5.34	2.19	99.85	2.44
Fib(58)	637.89	211.27	127.31	3.02	637.84	187.41	143.52	3.40
Nq(16)	2.83	2.73	69.93	1.04	2.84	1.62	117.84	1.75
Nq(19)	1107.21	330.21	240.56	3.35	1107.34	345.55	229.88	3.20
Pen(14)	3.83	4.06	58.98	0.94	3.84	2.54	94.28	1.51
Pen(16)	349.34	192.15	126.66	1.82	349.48	148.66	163.71	2.35
LU(7000)	13.43	146.33	1.89	0.09	13.40	309.72	0.89	0.04

TABLE III
EXECUTION TIME (AND RELATIVE TIME TO SEQUENTIAL C PROGRAMS) ON XEON PHI.

	Tascell/MPI		
Return message	NONE	TASK	NONE+TASK
Fib(50)	15.46ms	19.86ms	15.82ms
Nq(17)	13.85ms	20.29ms	14.44ms
Pen(15)	5.29ms	16.54ms	5.85ms
LU(7000)	1.94ms	38.26ms	2.06ms

TABLE IV
THE AVERAGE RESPONSE TIME TO TASK REQUESTS ON THE K COMPUTER.

B. The K computer

On the K computer, we evaluated the performance only of the proposed implementation. The results of the performance measurements are shown in Table V and Fig. 5. In Table V, t_{C} denotes the execution time of the sequential C programs and $t_{\mathrm{K}n}$ denotes the execution time of Tascell/MPI with n workers (8 workers in each node). Table IV shows the average response time to task requests. The meanings of NONE, TASK, and NONE+TASK in Table IV are the same to those in Table II.

As on Xeon Phi coprocessors, we cannot get good speedups for LU(7000) but can get good speedups for Fib(50), Nq(17) and Pen(15). Moreover, we should note that our Tascell implementation efficiently works on the K computer, in which TCP/IP communication among computation nodes is not available.

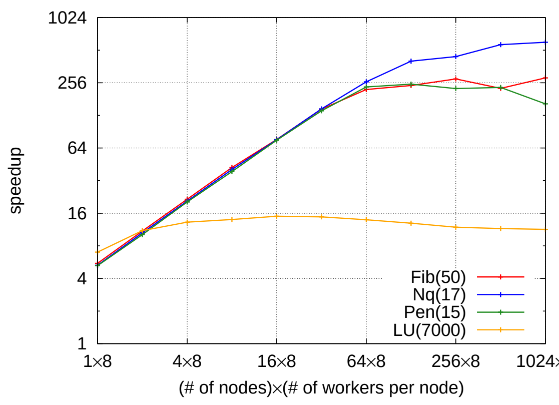


Fig. 5. Speedups of Tascell/MPI with multiple computation nodes on the K computer (relative to sequential C).

	C	Tascell/MPI			
# of workers	1	8	8192	speedup	
	t_{C}[s]	$t_{\mathrm{K}8}$[s]	$t_{\mathrm{K}8192}$[s]	$t_{\mathrm{C}}/t_{\mathrm{K}8192}$	$t_{\mathrm{K}8}/t_{\mathrm{K}8192}$
Fib(50)	316.11	57.30	1.11	284.78	51.62
Nq(17)	629.29	118.89	1.04	605.09	114.32
Pen(15)	1550.07	295.36	9.47	163.68	31.19
LU(7000)	378.89	54.08	33.34	11.36	1.62

TABLE V
EXECUTION TIME (AND RELATIVE TIME TO SEQUENTIAL C PROGRAMS) ON THE K COMPUTER.

V. RELATED WORK

A. Parallel Language

The X10 parallel language [8] is implemented using MPI and works with the `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` thread support level. With an MPI implementation with up to `MPI_THREAD_SERIALIZED` thread support level, X10 performs exclusive control for MPI function calls. X10 uses nonblocking functions of MPI to send and receive, and adds handles of MPI to a pending queue. X10 programmers need to call the `x10rt_probe` function explicitly to progress inter-node communications.

B. MPI Libraries

The MPI library installed on the K computer is an Open MPI based implementation [9]. We have to implement Tascell/MPI that works under the `MPI_THREAD_SERIALIZED` thread support level because Open MPI is recommended not to be used with the `MPI_THREAD_MULTIPLE` thread support level.

Intel MPI on Xeon Phi work with the `MPI_THREAD_MULTIPLE` thread support level by adding `-mt_mpi` option when compiling programs.

VI. Conclusion and Future work

We proposed an implementation of the task-parallel language Tascell in which computation nodes directly communicate with each other using MPI without Tascell servers.

In order to realize deadlock freedom requiring only the two-sided communication paradigm and the `MPI_THREAD_FUNNELED` thread support level, we use busy-waiting for incoming and outgoing messages. We also discussed that such busy-waiting is unavoidable without requiring the `MPI_THREAD_MULTIPLE` support level.

Our Tascell implementation showed better performance than the conventional TCP/IP-based implementation and a TCP/IP-based server-less implementation on four Xeon Phi coprocessors. Moreover, our implementation works efficiently on the K computer, in which TCP/IP communication among computation nodes is not available.

Future work will include exploring the feasibility of improving the MPI-based implementation of inter-node communication requiring more strong supports by MPI implementations, such as the `MPI_WIN_UNIFIED` memory model for one-sided communications. We will also evaluate our implementation using more practical applications such as graph mining.

Acknowledgment

This work was supported in part by JSPS KAKENHI Grant Numbers 25730041 and 26280023. Part of the results is obtained by using the K computer at the RIKEN Advanced Institute for Computational Science. This work was conducted in part in collaboration with RIKEN AICS for "Research and development to improve programming environment of the K computer".

References

[1] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *ACM SIGPLAN Notices (PLDI '98)*, vol. 33, no. 5, pp. 212–223, May 1998. [Online]. Available: http://doi.acm.org/10.1145/277652.277725

[2] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa, "Backtracking-based Load Balancing," *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2009)*, pp. 55–64, 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504187

[3] D. Muraoka, M. Yasugi, and T. Hiraishi, "An MPI-based Implementation of the Tascell Task-Parallel Programming Language," in *the 17th JSSST Workshop on Programming and Programming Languages (PPL 2015)*, March 2015, (In Japanese).

[4] T. Hiraishi, M. Yasugi, and S. Umatani, "Evaluation of the Tascell Dynamic Load Balancing Framework in Widely Distributed and Many-Core Environments," in *Symposium on Advanced Computing Systems and Infrastructures (SACSIS 2011)*, May 2011, pp. 55–63, (in Japanese).

[5] G. Luecke, Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva, "DEADLOCK DETECTION IN MPI PROGRAMS," 2002.

[6] T. Breuel, "Lexical Closures for C++," in *Usenix Proceedings, C++ Conference*, 1998.

[7] T. Hiraishi, M. Yasugi, and T. Yuasa, "A Transformation-Based Implementation of Lightweight Nested Functions," vol. 47, no. 29, pp. 50–67, may 2006. [Online]. Available: http://ci.nii.ac.jp/naid/110006390851/

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[9] N. Shida, S. Sumimoto, and A. Uno, "MPI library and low-level communication on the K computer," *FUJITSU Scientific & Technical Journal*, vol. 48, no. 3, pp. 324–330, July 2012.