Parallel Computing on RTEMS Operating System

Zeyu Liang[1] and Lei Wang[2]

[1] University of California, San Diego, La Jolla CA 92037, USA
[2] Beijing University of Posts and Telecommunications, Beijing 100876, China

Abstract. With the increasing complexity of computing tasks and the rise in data volume, current embedded and realtime systems, which are armed with multiple kinds of architectures, require parallel computing capability. However, the mainstream parallel programming models like OpenCL and MPI are too complex for these embedded and realtime systems. In this paper, we present a heterogeneous parallel programming framework Paor and implement it on RTEMS operating system. Paor provides several easy-to-use APIs to facilitate that computing devices of different architectures can collaborate in executing computing-intensive or data-intensive tasks on RTEMS operating system. Moreover, Paor provides supports for computational operators, including standard operators based on BLAS (Basic Linear Algebra Subprograms) and user-defined operators, thereby facilitating the parallelization of conventional mathematical computations. Experimental results in a heterogeneous computing environment show that Paor performs well on both computing-intensive and data-intensive tasks.

Keywords: Parallel computing · RTEMS · BLAS

1 INTRODUCTION

With the rapid advancement of information technology and the increasing trends towards artificial intelligence, embedded and networked devices are being widely deployed in various fields. On one hand, these devices are equipped with multiple and/or heterogeneous processors; On the other hand, these devices usually run with embedded and realtime operating systems, which lack of commonly used software infrastructure, such as parallel computing facilities, compared to mainstream desktop or server OS like Linux or Windows.

Cluster-based computing is the mainstream method for parallel computing in many application fields, usually with Linux as the most popular operating system and MPI [1] as the standard protocol for message passing between nodes. The use of standard messaging APIs simplifies the development and portability of distributed applications. However, current MPI implementations (such as MPICH [2] and OpenMPI [5]) rely on operating systems like Linux or Windows and require significant memory and computing capabilities, and hence unsuitable for embedded systems with limited computing power and small memory footprint.

To address the above issue, we propose a heterogeneous parallel computing framework called Paor, and implement it on RTEMS, an open source embedded and realtime operating system. Paor offers a promising solution for building clusters with embedded devices running on RTEMS. Paor consists of two kinds of nodes. Control nodes are responsible for task distribution, task control, and the collection and merging of task computation results. compute nodes can be of different processor architecture and each of them executes specific computations and returns results to control nodes. Moreover, Paor provides supports for computational operators, including standard operators based on BLAS (Basic Linear Algebra Subprograms) and user-defined operators, thereby facilitating the parallelization of conventional mathematical computations.

Our contributions are summarized as follows.

- We present a heterogeneous parallel computing framework called Paor and implement it on RTEMS operating system. Paor provides a promising solution for embedded devices running on RTEMS. To our best knowledge, Paor is the first to provide heterogeneous parallel computing capability for RTEMS.
- We enhance Paor with commonly used computational operators which contain both standard BLAS-based operators and custom logic-based operators. This allows for the utilization of multiple processors or cores in compute nodes and improves the computational efficiency of parallel programs.
- We evaluated Paor in a heterogeneous cluster and experimental results show that Paor performs effectively and efficiently on both computing-intensive and data-intensive tasks.

The remainder of this paper is structured as follows. Section 2 describes the background of parallel computing frameworks. We present the detail design of Paor in Section 3. Section 4 evaluates Paor's effectiveness and efficiency in a heterogeneous cluster. We discuss related work in Section 5, and conclude in Section 6.

2 BACKGROUND

In this section, we introduce the background of the parallel programming models widely used in server or desktop clusters, such as MPI, OpenMP, and OpenCL.

2.1 MPI

MPI [1, 18] is a widely used programming model in parallel computing that uses *message passing interfaces*. It provides a rich set of APIs that define a set of data types and communication operations, such as point-to-point communication, collection operations, synchronization operations. MPI supports multiple data transmission methods and allows efficient data transfer and communication between multiple devices or nodes. It is suitable for multiple parallel computing

requirements, such as clusters and supercomputers. There are several implementations of MPI for mainstream operating systems and parallel environments, such as MPICH [2], MPICH2 [3], MVAPICH [4], OpenMPI [5], LAM/MPI [14].

2.2 OpenCL

OpenCL [6] is an open industry standard for cross-platform, parallel programming of diverse accelerators found in supercomputers, cloud servers and personal computers. It supports multiple kinds of devices, such as CPU, GPU, and FPGA. It includes a language, API, libraries and a runtime system to support software development. The target of OpenCL is expert programmers wanting to write portable yet efficient code, hence it provides a low-level hardware abstraction plus a framework to support programming and many details of the underlying hardware are exposed, such as platform model, memory model, execution model and programming model [7].

2.3 OpenMP

The OpenMP API [8] supports multi-platform *shared-memory* parallel programming in C/C++ and Fortran. It defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the memory. A given storage location in the memory may be associated with one or more devices, such that only threads on associated devices have access to it [19].

Unfortunately, since their design purposes are different from the requirements in heterogeneous embedded and realtime devices, none of the above models was applied to embedded real-time systems, especially RTEMS operating system.

3 APPROACH

In this section, we describe the design of Paor for parallel computing in heterogeneous clusters.

3.1 Overview of Paor

The architecture of Paor is depicted in Figure 1. Paor employs a master-slave topology with a star-shaped configuration, comprising two distinct types of nodes: control nodes and compute nodes. The control node serves as the central component of the cluster, tasked with managing job distribution, overseeing task execution, and aggregating the computational results. Each slave node, referred to as a compute node, is dedicated to performing the computational

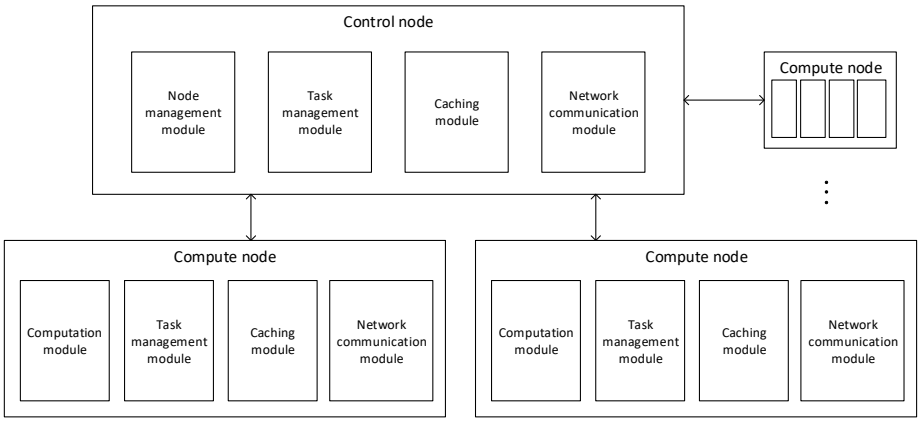

Fig. 1: The architecture of Paor

tasks assigned by the control node and returning the processed results. The compute nodes are designed to leverage multiple computational resources (e.g., CPU or GPU) through multi-threading, enabling efficient parallel processing.

Paor incorporates a range of features, including task states, caching mechanism, and communication protocol, which collectively enhance the system's functionality and adaptability. The subsequent sections provide a detailed examination of Paor, with a focus on the internal modules and their respective roles.

The control node is composed of several key modules: node management, task management, caching, and network communication. The node management module oversees the administration of active compute nodes, ensuring their availability and performance within the cluster. Task management is responsible for the allocation of tasks, aggregation of computational results, and synchronization of data across nodes. The caching module plays a critical role in storing function parameters, registering function, and retaining both inputs and their corresponding outputs. The network communication module is designed to optimize network I/O performance, thereby enhancing the efficiency of the parallel computing.

The compute node, similarly, is structured around modules for task management, computation, caching, and network communication. The task management module on each compute node handles the reception and execution of tasks, as well as the transmission of computed results back to the control node. The computation module is dedicated to executing the received tasks. The caching module is responsible for real-time monitoring of the node's operational status and for storing computational outputs.

During system initialization, the control node is the first to activate, followed by the compute nodes, which come online and periodically report their capabilities, performance metrics, and current status. The control node continuously

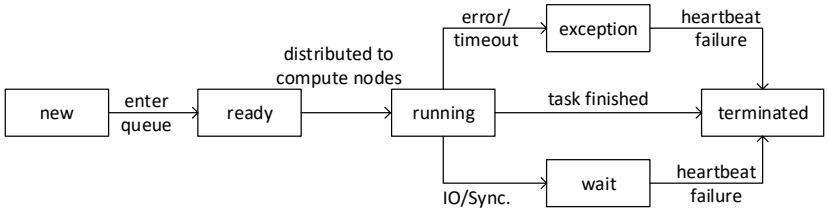

Fig. 2: The transition of task states

updates this information to maintain a comprehensive and up-to-date view of all nodes. If the control node fails to receive updates from a compute node within a predefined time interval, it will mark that node as offline.

3.2 Control nodes

From a task lifecycle perspective, tasks transition through several states, including new, ready, running, waiting, exception, and terminated. Initially, the control node creates the task and places it in the task queue, at which point the task is in the ready state. The control node then decomposes the task based on its type and distributes it to one or more appropriate compute nodes, moving the task to the running state. During computation, if multiple threads within a compute node are involved, the task may enter a waiting state. Once the compute node completes the assigned task, it returns the results to the control node, indicating the task's completion for that node. If a compute node encounters an issue, such as a timeout or going offline, the task transitions to an exception state. After the failure of (e.g., a user-defined count) successive heartbeat signals, the task is marked as in the terminated state. The state transition of a task is illustrated in Figure 2.

Paor includes control nodes and compute nodes, with computing devices encompassing hardware of various architectures. The control node assigns tasks to the most suitable computing device based on the nature of the task. Task distribution is handled through the *send_task* function, which first analyzes its parameters to verify the existence of a corresponding registered function. Subsequently, tasks are allocated to one or more nodes based on task priority and node performance. Additionally, space is reserved in the cache to store the results of the computations.

3.3 Compute nodes

3.3.1 Writing and registering computing tasks One of the primary responsibilities of compute nodes is executing parallel computing tasks, which necessitates the proper design and registration of these tasks. Users can tailor different types of computing functions according to the specific hardware characteristics of the compute nodes. For instance, if a compute node is equipped with

multi-core CPUs, it is particularly suited for executing parallel programs optimized for CPU architectures. Conversely, if the compute node includes additional devices such as GPUs, it becomes appropriate to incorporate GPU-accelerated programs. To facilitate this, we have developed a basic environment for compute nodes. This environment includes essential functionalities for writing and registering computing tasks, as well as a comprehensive compilation and runtime environment for executing parallel programs.

When developing parallel programs, users need to focus primarily on two key aspects: defining her computations in a function F and adding the function F into the registered function list. The implementation of these functions can rely on external environments, such as libraries and header files, which can be integrated into the program as needed.

For example, if the compute node hardware only has multi-core CPUs, it is suitable for parallel computing programs with CPUs. If the compute node contains devices such as GPUs, it is suitable to add GPU type computing programs.

Paor provides the registration basis and environment for computing programs. After adding the computation function, the user compiles it with the provided compilation tool until it succeeds. Generate an executable program after successful compilation. The compute node runs this program by first communicating with the control node to report computing power and other information, then registering the computing function and reporting the running status in real time. The compute node architecture aims to provide flexibility and scalability to adapt to different configurations and requirements. Subsequent chapters will elaborate on configuration, startup process, parallel computing, and communication mechanisms, laying the foundation for efficient system operation.

Paor offers a foundational mechanism for the registration and execution of parallel tasks. After the user adds a computational function, it must be compiled using the provided compilation tools. Upon successful compilation, an executable program is generated. The compute node executes this program by first establishing communication with the control node to report its computational capabilities, followed by registering the computational function and continuously reporting its operational status in real time. This design aims to to offer flexibility and scalability, enabling it to adapt to various hardware configurations and computational requirements.

Running tasks: After joining the cluster, the compute node continuously monitors the system in real time. Upon receiving a computational task, it invokes the relevant functions and utilizes the provided parameters to initiate computation, marking the compute node as "running". Once the computation is completed, the execution result is written to the specified location in the cache module as defined during task distribution, and the compute node is subsequently marked as "idle".

Task synchronization: In Paor, control nodes are responsible for decomposing computational tasks and distributing them across various compute nodes. This involves segmenting the input data into appropriate portions for parallel processing and implementing mechanisms for data segmentation, distribution,

and synchronization. The synchronization of function parameters is managed by the control node, which partitions the input data and sends it to the compute nodes before the computational task begins.

After the computation task is dispatched via the *send_task* function, the main program continues to execute without blocking until it encounters the *wait* function. This approach allows multiple computational tasks to be dispatched asynchronously, facilitating parallel execution. Such asynchronous execution enhances Paor's flexibility and efficiency, making it particularly suitable for large-scale parallel computing clusters. It maximizes resource utilization by enabling the simultaneous execution of multiple tasks, thereby improving overall computational efficiency.

Upon the completion of computation by the compute node, the *wait* function is employed to synchronize and aggregate the results. The main program remains blocked until the results of all tasks have been successfully received, ensuring the readiness of the computational outputs. The *wait* function monitors the return results of the tasks, retrieving them based on their locations within the cache module. If a task times out, the *wait* function alerts the control node, which then decides whether to retransmit the task or take alternative action.

3.3.2 Computational Operators Paor provides supports of two types of computational operators: standard operators based on the Basic Linear Algebra Subprograms (BLAS) library and operators built on custom logic. These operators serve distinct functions: standard BLAS-based operators are primarily used for fundamental mathematical operations, while custom logic operators are tailored for more complex computational tasks.

Standard Operators Based on BLAS: Paor utilizes the BLAS library to provide a standardized computational interface that is compatible with heterogeneous compute nodes. BLAS offers three levels of operations: vector-to-vector (Level 1), matrix-to-vector (Level 2), and matrix-to-matrix (Level 3), all of which support four types of data precision. Paor leverages cBLAS and OpenBLAS, thereby enabling parallelization of conventional mathematical operations. By employing the computational tasks of a parallel program across different nodes, Paor improves the program's performance.

Operator based on custom logic: In order to meet more complex computational requirements, Paor supports user-defined or custom functions to be registered as computational operators. By using Paor's APIs, Users can provide function pointers and summaries to register their own functions. This design decouples complex logic from the underlying framework, enhancing Paor's scalability and adaptability.

3.4 Communication Module

The communication module is a critical component of Paor. The control node establishes listening ports, enabling compute nodes to report their status and other relevant information to the control node in real time. Communication

Fig. 3: The format of an application protocol package

Table 1: The meanings of the protocol header and data for each function

Function	Protocol header	Protocol Data
send_task	0x0001	function_name, para-1, ..., para-n
heartbeat	0x0002	node_info
send_result	0x0003	taskid, result, type
get_compute_node	0x0004	node_list
get_node_status	0x0005	node_detail
node_online	0x0006	node_info
node_offline	0x0007	node_info
wait	0x0008	para-1, ..., para-n
get_task_result	0x0009	task_id, result, type
send_result_complete	0x000A	task_id
apply_for_node	0x000B	node_info

between these nodes is facilitated through a custom protocol built on top of the TCP protocol. This design ensures real-time communication between the control and compute nodes, thereby enabling the entire system to collaborate effectively in executing distributed computing tasks.

3.4.1 Communication protocol Paor leverages the existing communication infrastructure library of the RTEMS operating system to construct the foundational protocol at the IP layer. This IP layer protocol enables the identification of each node's address within the network, facilitating communication between nodes based on their IP addresses.

In addition, a TCP layer protocol is built atop the IP layer protocol. Given the presence of a caching module (see next section), Paor can support one-to-many communication, allowing the server to initiate real-time calls to client software for executing computational tasks, thereby minimizing task latency associated with network delays. The use of TCP as the communication method enhances reliability. A custom application layer protocol is constructed on top of the TCP layer protocol. This application layer protocol is designed with several API functions, which are encapsulated by the TCP layer protocol and further encapsulated by the IP layer protocol. The format of the custom protocol is illustrated in Figure 3.

For each API function offered by the custom protocol, Table 1 illustrates the meanings of the protocol header and protocol data which are transmitted in JSON format.

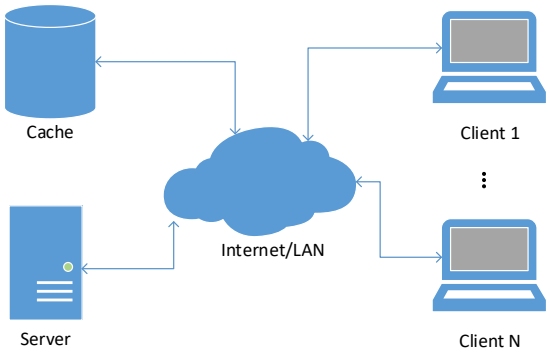

Fig. 4: Communication model of Paor

3.4.2 Communication model The communication model of Paor is shown in Figure 4. Using the above protocol, the server program and cache module in the control node can communicate directly with the client program in the compute nodes, and the server program can also communicate with the cache module. However, the client programs are not allowed to communicate directly with each other.

At startup, the server begins by listening on a specified port, while clients establish connections through this port. The server retrieves information about the compute nodes by invoking the API of the cache module, subsequently allocating computational tasks based on the requirements of the tasks and the status of the nodes. The clients periodically send heartbeat packets to the cache module, reporting their status to monitor node health and store computation results. The server then retrieves these results from the cache module as needed. This centralized coordination mechanism streamlines the communication architecture and enhances overall system control.

3.5 Cache module

The cache module provides an interface for compute nodes to report information and real-time statistics on various performance metrics of the compute nodes. It offers the control node access to information about compute nodes for selecting suitable nodes for computation. The cache module also provides an interface for the control node to access information about registered functions, enabling the control node to understand the available functions and their parameters on the compute nodes. Additionally, the cache module provides interfaces for the control node to allocate storage for function parameters and return results, facilitating data exchange between the control node and compute nodes. It also offers an interface for compute nodes to write their computation results.

The cache module serves as a critical interface for compute nodes to report information and provide real-time statistics on performance metrics. It enables the control node to access detailed information about compute nodes, allowing

Table 2: Experiment setup

	Control node	Compute nodes
CPU architecture	x86 i386	ARM
Host OS	Linux 4.15.0	Linux 4.15.0
Emulator	qemu-system-i386	qemu-system-arm
Guest OS	RTEMS 5.1	RTEMS 5.1

for the selection of the most suitable compute nodes for computation tasks. Additionally, the cache module provides an interface for the control node to know about registered functions, thereby enabling the control node to understand the available functions and their parameters on the compute nodes.

Furthermore, the cache module facilitates data exchange between the control node and compute nodes as it offers methods for the control node to allocate storage for function parameters and computation results. It also provides a method for compute nodes to write their computation results back into the cache, ensuring efficient data handling and parallel computing.

4 EVALUATION

In this section, we aim to answer the following research questions through empirical evaluation.

- RQ1: How does Paor perform on computing-intensive tasks?
- RQ2: How does Paor perform on data-intensive tasks?
- RQ3: Are the computational operators implemented in Paor correctly?

To test the correctness and effectiveness of the implementation of heterogeneous parallel computing framework based on RTEMS operating system, two testing programs are commonly used : the π value computation program `cpi.c` and the matrix multiplication operation program `matrix.c`. These programs served as benchmark tests to evaluate the performance of parallel computing frameworks on computationally intensive and data-intensive tasks. The simplicity and well-defined mathematical models of these programs make it easier to verify the accuracy of the results produced by the parallel computing framework.

We conducted experiments in a cluster which consists of several machines. The control node runs on an Intel i7 8750h CPU with 16 cores and 16G memory. Each compute node runs on a xilinx-zynq-a9 CPU with 1G memory. Their hardware and software setup are shown in Table 2.

4.1 Answer to RQ1

The π value computation program is highly parallelizable, as each iteration can be computed independently without direct data dependencies. This program

Table 3: Performance on π computation

#CN	Runtime (s)			Parallel Speedup		
	100	1000	10000	100	1000	10000
1	0.00613	0.05448	0.54156	1	1	1
2	0.00321	0.02789	0.27818	1.909	1.953	1.947
3	0.00216	0.01957	0.19242	2.737	2.783	2.814

involves a large number of computational operations, making it ideal for evaluating the performance of parallel computing frameworks on computing-intensive tasks that require high computational power.

The control node sends data to each compute node through send_task, and each node obtains its own computing task. After the computing task is completed, the result is returned to the control node. Using clusters of 1∼3 compute nodes (Column #CN) and loop iterations of 100, 1000 and 10000 for each test respectively, the runtime and parallel speedup of the π computing program are shown in Table 3.

Table 3 shows that Paor can run well on RTEMS and exposes nice parallel efficiency. For each loop iteration case, the π programs best when using three compute nodes and obtains over 2.7× parallel speedup.

4.2 Answer to RQ2

Matrix multiplication is a classic parallel computing problem, where multiplication operations can be decomposed into independent sub-problems, each of which can be computed in parallel. Matrix multiplication involves a large amount of data exchange and memory access, demanding high memory and communication performance. Therefore, matrix multiplication is commonly used to evaluate the performance of parallel computing frameworks on data intensive tasks, and it can be tested and compared on dataset of different scales.

To test the performance of matrix multiplication `matrix.c` on the RTEMS operating system,, we evaluate Paor's performance using the *CTRMM* function (a Blas matrix operation) in the GSL library which is a widely used mathematics library in real-world software. The function computes the following matrix-matrix products, using the scalar α, rectangular matrix B, and triangular matrix A, i.e., $B = \alpha AB$. We leverage three sets of matrices of 50*50, 500*500, and 3000*3000, respectively. The experimental results, i.e., runtime and parallel speedup, are shown in Table 4.

The results that 1) with the increase of matrix size, the program requires more runtime as expected; 2) with the increase of compute nodes, the program runs faster; 3) the program obtains the largest speedup (i.e., 14.120) at the case of 6 compute nodes and 3000*3000 matrix.

Table 4: Performance on the CTRMM operator in GSL library

Matrix size	Runime (s)			Parallel Speedup		
	1CN	3CN	6CN	1CN	3CN	6CN
50*50	0.201	0.115	0.109	1	1.748	1.844
500*500	24.690	6.530	5.946	1	3.781	4.153
3000*3000	4355.385	725.405	308.463	1	6.004	14.120

```
-----------RTEMS App Init------------
Init Network ...
Ethernet address 0:0:1:1:1:1
PCI IDs: 0x8086 0x1229 0x1af4 0x1100 0x2
Chip Type: 1
Network init successful!
******************Net Client main******************
input <-- A: 3 * 3
4.000000 0.000000 0.000000
0.000000 0.000000 0.000000
1.000000 0.000000 1.000000
cblas_ctrsm result --> B: 3 * 3
0.500000+0.000000*I    2.500000+0.000000*I   -7.500000+0.000000*I
2.750000+0.000000*I   -4.750000+0.000000*I   10.750000+0.000000*I
0.250000+0.000000*I    0.750000+0.000000*I    3.250000+0.000000*I
```

Fig. 5: The test result of CTRMM operator in BLAS

4.3 Answer to RQ3

To verify the correctness of the computational operators in Paor, we constructed test inputs for each operator using various data types, including single precision, double precision, real numbers, and complex numbers. The experimental results confirm the correctness of all 142 BLAS operators integrated within the Paor framework. As an example, Figure 5 shows the test result of `CTRMM` operator in BLAS.

Building on the BLAS operators, we developed and registered a new operator function named `user-blas-sgemmvv`, defined as $Y = X \times A \times B$, where A and B are matrices, X and Y are vectors. The test result for this operator is confirmed correct and presented in Figure 6. Overall, 142 BLAS operators and user-defined operator mechanism are correctly implemented, whose results are not shown here due to page limit.

```
--------RTEMS App Init---------
Init Network ...
Ethernet address 0:0:1:1:1:1
PCI IDs: 0x8086 0x1229 0x1af4 0x1100 0x2
Chip Type: 1
Network init successful!
****************Net Client main*****************
user-defined: user-blas-sgemmvv
input <-- A: 4 * 3
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
8.000000 7.000000 6.000000
input <-- B: 3 * 2
5.000000 4.000000
3.000000 2.000000
1.000000 2.000000
result --> C: 4 * 2
14.000000 8.000000
41.000000 26.000000
68.000000 44.000000
67.000000 46.000000
user-defined: user-blas-sgemmvv result --> Y: 4
30.000000 93.000000 156.000000 159.000000
```

Fig. 6: The test result of user-defined operator user-blas-sgemmvv

5 RELATED WORK

MPI [18] was originally designed for high-performance communication on large-scale parallel machines. It first provides the basic communication function interface used for inter process communication, and then communication mode extension, process creation management, environment management, language constraints, and parallel I/O [1]. Recently it adds support of C++ types and unilateral communication functions etc. MPICH [2] is a high-performance, standard compatible MPI implementation. Open MPI [5] is an open-source implementation of MPI. Jin et al. [15] studied the mixed programming model of MPI and OpenMP in multi-core parallel systems. Agbaria et al. [12] proposed an MPI implementation LMPI for Linux operating systems. However, the above implementations are quite complex for embedded and realtime devices.

Moreover, some research teams have proposed several parallel programming models for heterogeneous systems. For example, NVIDIA launched CUDA for GPGPUs [10]. PGI and multiple companies developed OpenACC [11]. Microsoft released C++ AMP [9]. However, these parallel models are not universal, as they

are designed for those heterogeneous systems with specific requirements, thereby leading to low scalability and portability.

For systems based on multi-CPU and multi-GPU architectures, OpenCL [6] provides strong supports. For instance, based on OpenCL, Samsung Electronics released SNUCL [16] framework that supports Cell processors, ARM processors, and digital signal processors. Kronos [13,16] released a high-level programming model SYCL based on OpenCL, which greatly simplifies the interfaces of OpenCL by using upper layer interfaces, and enhances the flexibility of programming. Kim et al. [17] proposed a framework that treats GPGPUs as device side in the OpenCL model. By writing applications for a single GPU and then porting them to a GPGPU system containing multiple GPUs, these applications can fully utilize the computing power of all GPUs resources.

It's worthwhile to notice that the methods or implementations outlined above were not applied to or unsuitable for embedded and realtime devices with heterogeneous architectures.

6 CONCLUSION

We present and implement a heterogeneous parallel programming framework Paor on the RTEMS operating system, which is optimized and suitable for embedded and real-time devices. Paor provides several easy-to-use APIs to facilitate that computing devices of different architectures can collaborate in executing computing-intensive or data-intensive tasks on RTEMS operating system. Moreover, Paor provides supports for computational operators, including standard operators based on BLAS (Basic Linear Algebra Subprograms) and user-defined operators, thereby facilitating the parallelization of conventional mathematical computations. Experimental results in a heterogeneous computing environment show that Paor performs well on both computing-intensive and data-intensive tasks.

References

1. The mpi-2: Extensions to the message passing interface. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
2. Mpich. http://www.mcs.anl.gov/mpi/mpich1/
3. Mpich2. http://www.mcs.anl.gov/mpi/mpich2/
4. Mvapich and mvapich2 project. http://mvapich.cse.ohio-state.edu/
5. Open mpi: Open source high performance computing. http://www.open-mpi.org
6. Opencl: Open computing language. https://www.khronos.org/opencl/
7. Opencl: Open computing language version 3.0. https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_the_opencl_architecture
8. Openmp: Open multi-processing. https://www.openmp.org/
9. Microsoft corporation. c++ amp: Language and programming modes. https://www.openacc-standard.org/ (2013)

10. Nvidia cuda toolkit. https://www.developer.nvidia.com/cuda-downloads/ (2017)
11. Openacc: Directives for accelerators. https://www.openacc-standard.org/ (2017)
12. Agbaria, A., Kang, D.I., Singh, K.: Lmpi: Mpi for heterogeneous embedded distributed systems. In: International Conference on Parallel & Distributed Systems (2006)
13. Alpay, A., Heuveline, V.: Sycl beyond opencl: The architecture, current state and future direction of hipsycl. In: IWOCL '20: International Workshop on OpenCL (2020)
14. Burns, G., Daoud, R., Vaigl, J.: LAM: An open cluster environment for MPI (1994)
15. Haoqiangjin, Jespersen, D., Mehrotra, P., Biswas, R., Huang, L., Chapman, B.: High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing **37**(9), 562–575 (2011)
16. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In: International Conference on Supercomputing (2012)
17. Kim, J., Seo, S., Lee, J., Nah, J., Lee, J.: OpenCL as a unified programming model for heterogeneous CPU/GPU clusters. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012 (2012)
18. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.1 (Nov 2023), https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf
19. OpenMP Group: OpenMP Application Programming Interface Version 5.2 (Nov 2021), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf