# ABSS: An Adaptive Batch-Stream Scheduling Module for Dynamic Task Parallelism on Chiplet-based Multi-Chip Systems

QINYUN CAI, GUOQING XIAO, SHENGLE LIN, and WANGDONG YANG,
Hunan University, Changsha, China
KEQIN LI, State University of New York, NY, USA and Hunan University, Changsha, China
KENLI LI, Hunan University, Changsha, China

Thanks to the recognition and promotion of chiplet-based High-Performance Computing (HPC) system design technology by semiconductor industry/market leaders, chiplet-based multi-chip systems have gradually become the mainstream. Unfortunately, programming such systems to achieve efficient computing is a challenge, especially when considering dynamic task parallelism. This paper presents an Adaptive Batch-Stream Scheduling (ABSS) module for dynamic task parallelism on chiplet-based multi-chip systems. To this end, we propose an adaptive batch-stream scheduling method based on Graph Convolution Network (GCN) classifier to select the appropriate scheduling scheme. We further design a chiplet-based core-cluster binding mechanism, which establishes the affinity between threads and core-clusters on CPU-compute die. Moreover, to achieve dynamic workload balance, we propose a chiplet-based nearest task stealing method. We implement our ABSS module on the HiSilicon Kunpeng-920 chiplet-based multi-chip system. Experiments show that it outperforms state-of-the-art parallelism solutions, such as Intel Threading Building Blocks.

CCS Concepts: • **Theory of computation → Scheduling algorithms**; • **Computer systems organization → Parallel architectures;**

Additional Key Words and Phrases: Adaptive batch-stream scheduling, chiplet-based core-cluster binding, chiplet-based multi-chip system, chiplet-based nearest task stealing, task parallelism

**ACM Reference Format:**
Qinyun Cai, Guoqing Xiao, Shengle Lin, Wangdong Yang, Keqin Li, and Kenli Li. 2024. ABSS: An Adaptive Batch-Stream Scheduling Module for Dynamic Task Parallelism on Chiplet-based Multi-Chip Systems. *ACM Trans. Parallel Comput.* 11, 1, Article 6 (March 2024), 24 pages. https://doi.org/10.1145/3643597

New Paper, Not an Extension of a Conference Paper.
The research was partially funded by the Key-Area R&D Program of Guangdong Province (2021B0101190004), the Programs of National Natural Science Foundation of China (62321003, U21A20461, 62172157, 92055213, 62227808), the Major Projects of Xiangjiang Laboratory (22xj01011), the Key R&D Program of Hunan Province (2023GK2002), and the Shenzhen Science and Technology Program (JCYJ20210324135409026).
Authors' addresses: Q. Cai, S. Lin, W. Yang, and K. Li, College of Computer Science and Electronic Engineering, Hunan, University, Changsha, Hunan, 410082, China; e-mails: hnutsai@hnu.edu.cn, lsl036@hnu.edu.cn, yangwangdong@hnu.edu.cn, lkl@hnu.edu.cn; G. Xiao (Corresponding author), College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, 410082, China and Shenzhen Research Institute, Hunan University, Keyuan Road S., Nanshan District, Shenzhen, 518000, China; e-mail: xiaoguoqing@hnu.edu.cn; K. Li, College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan, China, 410082, and State University of New York, Science Hall 249, 1 Hawk Drive, New Paltz, New York, USA; e-mail: lik@newpaltz.edu.

ACM 2329-4949/2024/03-ART6
https://doi.org/10.1145/3643597

## 1 INTRODUCTION

With the development of low-power, low-cost, and high-performance processors, multi-core RISC chips and multi-chip systems have gradually become the mainstream architecture of **high-performance computing (HPC)** systems. The **non-uniform memory access (NUMA)** architecture is generally adopted to address memory bandwidth bottlenecks [9, 16]. Communication between on-chip cores is faster than that between off-chip cores, which can be quantified by the NUMA distance in a hierarchical NUMA distance matrix [29]. Although the **QuickPath Interconnect (QPI)** [17] and HyperTransport [5] technologies can narrow the difference between local and remote memory access latency to within 30%, reducing memory access overhead in multi-chip systems is still critical [12]. Additionally, large monolithic **System-on-Chips (SoCs)** typically result in low yields and tremendous costs in verification effort and advanced manufacturing [27].

Various market leaders within the industry have used/recognized a chiplet technology, which can divide a traditional System-on-Chip (SoC) into smaller functions and then implement them into different dies. It can significantly increase the yield of a single die and create a better balance between cost and performance. Emerging commercial chiplet-based multi-core RISC chips are being released, such as HiSilicon Kunpeng 920 and Phytium FT-2000+ [31]. Such chiplet-based multi-core RISC chip contains several **CPU-Compute Dies (CCDs)**, and each CCD contains dozens of computing cores (form a core-cluster) and a **last-level cache (LLC)**. Therefore, the remote LLC and the corresponding local caches of the core-cluster form a NUMA relationship throughout the SoC. When considering parallel programming in a chiplet-based multi-chip system, we face challenges not only with the impact of NUMA architecture, but also with the NUMA relationship among CCDs in a single chip.

Dynamic task parallelism is a parallel programming style, in which the workload is divided into tasks as units of computation, and is applied to familiar CPU programming models, such as Intel Cilk Plus [14], Intel **Threading Building Blocks (TBB)** [19], and OpenMP [3]. There are two scheduling solutions for task parallelism, including batch scheduling and stream scheduling. Among those, batch scheduling is the most common method with a long history and is widely adopted by multi-core architecture. However, batch scheduling may cause core idleness or inter-core workload imbalance in applications that have different scales of tasks or non-negligible inter-arrival times. The stream scheduler can immediately respond to each arriving task, and the work-stealing method can make it insensitive to the load imbalance of the task. However, stream scheduling leads to frequent access to state variables of computing units. Therefore, it is necessary to consider a way to identify the most suitable scheduling strategy for dynamic task parallelism and achieve the overall optimal performance.

In this paper, we focus on the optimization of dynamic task parallelism on chiplet-based multi-chip systems, and propose an **Adaptive Batch-Stream Scheduling (ABSS)** module to optimize the execution performance of tasks. Our ABSS module provides efficient parallelism services through adaptive batch-stream scheduling, chiplet-based core-cluster binding, and chiplet-based nearest task stealing. Our ABSS module is successfully implemented on the real-world chiplet-based multi-chip system of HiSilicon Kunpeng-920 server, and is delivered to **Kunpeng Math Library (KML)** to accelerate sparse matrix decomposition [11]. It yields competitive performance advantages over the state-of-the-art of parallelism solutions such as Intel TBB. The main contributions of this work are summarized as follows:

— We design an adaptive batch-stream scheduling method, which automatically and adaptively selects the optimal scheduling strategy by using a **Graph Convolutional Network (GCN)** classifier.

Table 1. Terminology used in this Paper

| Terminology | Description |
|---|---|
| HPC | High-performance computing |
| OS | Operating system |
| NUMA | Non-uniform memory access |
| CCD | CPU-compute die |
| LLC | Last level cache |
| ABSS | Adaptive batch-stream scheduling |
| deque | double-ended queue |
| MCU | Memory control unit |
| SLIT | System locality information table |
| BLAS | Basic linear algebra subprograms |
| GEMM | General matrix multiplication |
| SpQRF | Sparse QR factorization |
| SpLQF | Sparse LQ factorization |
| SpLUF | Sparse LU factorization |
| SpCHF | Sparse Cholesky factorization |
| TBB | Threading building blocks |

— To further address the problems of thread overhead and resource idleness, we propose a chiplet-based core-cluster binding mechanism. Threads are bound to the core cluster of CCDs instead of a single core, and multiple cores in each CCD share the same task queue. The mechanism can narrow the frequent context switching and remote memory access, and significantly improve the utilization of the cores.
— We propose a chiplet-based nearest task stealing mechanism for stream scheduling, which steals tasks from the nearest busy CCDs of each idle core. The mechanism not only maintains load balance, but also avoids frequent context switching caused by thread sleep and wake-up. It effectively minimizes the heavy overhead of remote memory access caused by the diverse NUMA distance and NUMA relationship between CCD and LLC.

The rest of the paper is organized as follows: Section 2 introduces the background and dynamic task parallelism of chiplet-based multi-chip systems. Section 3 presents the proposed ABSS module, and details the adaptive batch-stream scheduling method, the chiplet-based core-cluster binding for threads, the chiplet-based nearest task stealing, and the implementation. Section 4 evaluates the performance of the ABSS module by comparing it with the state-of-the-art solutions. Relevant work is discussed in Section 5. Section 6 discusses the scalability of the proposed module. Finally, the paper is summarized in Section 7.

## 2 BACKGROUND

In this section, we introduce the chiplet-based multi-chip HPC system and the corresponding parallelism programming of tasks. Table 1 gives the abbreviations and descriptions of the terminology methods used in this paper.

### 2.1 Chiplet-based Multi-chip System

A chiplet-based multi-chip system contains several chips and each chip is assembled by multiple chiplets (i.e., dies) which are stacked on an interposer via advanced packaging technologies [30]. There are several types of primitive dies in an SoC, such as CPU-Compute die, AI-Compute die, Compute-IO die, and Wireless-**accelerate (ACC)** die. These dies can be selected and combined

together to meet various requirements. Such a system can be seen on modern commercial HPC systems, such as HiSilicon Kunpeng 920 server and Phytium FT-2000+ server. A CCD contains a series of computing cores and their corresponding L1 cache, as well as the private L2 cache. The cores in the same CCD share the **last-level cache (LLC)**, and all CCDs in the same chip support full consistency. Each CCD independently accesses a memory node through its own **memory control unit (MCU)**, that is, all available memory in a node has the same access characteristics for all cores in a CCD. Based on the NUMA architecture support, each CCD in the chiplet-based multi-chip system is tightly connected to a NUMA node. The communication speed between on-chip CCDs is faster than off-chip, which can be quantified by the NUMA distance [29] which forms a hierarchical NUMA distance matrix. The NUMA distance is the relative distance between nodes, which is determined by the **system locality information table (SLIT)** and is given at the time of manufacturing. An example of the NUMA distance matrix $D_{\text{NUMA}}$ is expressed as:

$$D_{\text{NUMA}} = \begin{pmatrix} 10 & d_{0,1} & \dots & d_{0,N} \\ d_{1,0} & 10 & \dots & d_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ d_{N-1,0} & d_{N-1,1} & \dots & 10 \end{pmatrix}. \tag{1}$$

The value range of $D_{\text{NUMA}}$ is 10 to 254, and 255 represents that there is no connection between NUMA nodes. A value of 10 indicates how fast a thread can access the memory in the same NUMA node, and a value of 254 will be 25.4 times slower.

### 2.2 Dynamic Task Parallelism

Dynamic task parallelism is a style of parallel programming where the workload is divided into tasks as units of computation and dependencies among tasks are generated at runtime. This parallel programming style becomes popular in many modern programming frameworks, such as Intel Cilk Plus [14], Intel TBB [19] and OpenMP [4]. It can provide a flexible data-flow execution to exploit dynamic, irregular and nested parallelism of workloads, especially in scientific applications. The dependencies of the tasks and data-flow execution in task parallelism can be represented by a **Directed acyclic graph (DAG)**, where nodes represent tasks and edges represent data flow among tasks. A common parallelism solution for tasks parallelism is the fork-join mechanism [26] and the corresponding DAG is an up-down symmetrical structure.

In general, there are two types of scheduling methods for task parallelism that dynamically map tasks to hardware at runtime, including batch processing and stream processing. Batch processing is a static scheduling method and is widely used on multi-core architecture. The scheduler dispatches all tasks in the queue to the same number of idle cores as the tasks in the queue for execution. This method is suitable for applications in which the number of parallel tasks is predictable, arrives in batches, or the arrival time interval is negligible. For non-continuously arriving tasks, parallel tasks need to be added to the set batch scheduling queue after arriving at the system. They wait for other tasks in the queue until the queue is filled, and the scheduler will be scheduled to the same computing core. This method is more sensitive to task workload balancing. In contrast, stream processing is a dynamic scheduling method. After the task arrives at the system, the scheduler responds immediately and puts the task on an idle core for execution or joins the task buffer queue. The method is suitable for situations where the number of parallel tasks is unpredictable and the arrival interval cannot be ignored. It schedules tasks to the computing core for execution for the first time, reduces task waiting time, and improves task responsiveness, which is of great help to tasks with responsive requirements.

Work-stealing is a common and effective method for parallel programs on multi-core computers for load balancing among dynamic tasks [1]. Cilk Plus and TBB use a work-stealing mechanism to automatically and efficiently balance a load of forkjoin, which migrates tasks to idle processors [6]. When work-stealing is running, each thread is associated with a task queue to store tasks available for execution. The task queue is a **double-ended queue** (called **deque**). When a thread is available, it first attempts to fetch a task from the top of its own deque. If the thread's own deque is empty, it will try to steal a task from the bottom of another thread's deque. When the number of threads increases, the total number of deques maintained by threads increases, which will significantly increase the overhead of threads in work-stealing. The overhead primarily arises from the "stealing" aspect, as threads need to retrieve available tasks from other deques when carrying out task stealing. This overhead will sharply increase with the total number of threads and deques. The traditional work-stealing method where each thread maintains a task deque is not suitable for chiplet-based multi-chip systems. The prerequisite for the system cores executing in parallel is that the number of threads is not less than the number of cores, so an increase in the number of cores directly leads to an increase in the overhead incurred by threads during task stealing. The number of cores of the chiplet-based multi-chip system far exceeds that of traditional multi-core architecture. The former can reach hundreds of cores, while the latter has only dozens of cores, as shown in Table 3.

The carrier of task execution depends on threads, which are closely related to the architecture. The conventional optimization for threads on multi-core computing systems takes the strategy of binding threads to individual cores. This conventional optimization can reduce cache misses and migration time between CPUs and works well on architecture with a small number of CPU cores. However, adopting this strategy completely limits the ability of the **operating system (OS)** to schedule threads, which may cause significant performance degradation. Because threads bound to a core may concurrently undergo blockage, stemming from either explicit or implicit causes, and the strategy impedes the OS's ability to redistribute threads from other cores to the affected one. Explicit blockage occurs when a task is programmed to await the completion of other tasks, necessitating the suspension of the executing thread. Implicit blockage occurs when a task attempts to access a critical section and must wait for a mutex, leading to an involuntary suspension of the executing thread. When it happens, it can lead to resource underutilization unless the number of threads per core is increased. For example, there are 4 cores $A$, $B$, $C$, $D$, and each core is bound to 2 threads for executing tasks. When the 2 threads on core $A$ are both blocked due to the mutex lock, while cores $B$, $C$, $D$ still have spare threads available, but since these spare threads are bound to non-$A$ cores, even though there are spare threads available in the system, core $A$ still has no threads available. To mitigate this issue, we can either increase the number of threads bound to each individual core, or unbind the spare threads on non-$A$ cores, so the operating system can participate in thread scheduling. In a 4-core system, the overhead of increasing the number of threads bound to each single core is acceptable. However, in chiplet-based multi-chip systems with a large number of cores, this approach results in a substantial increase in the total number of threads system-wide. Such an escalation can excessively consume resources, which is unacceptable. The non-binding strategy is not a good choice either. Adopting a no-binding strategy may cause threads to frequently migrate across CCDs. In task execution, the computing core necessitates not only the loading of task data but also the loading of thread contexts in task execution. The absence of a thread binding mechanism during task scheduling results in indeterminate thread assignment to tasks. This uncertainty becomes problematic when the thread executing the task comes from a remote node, requiring the core to perform remote access for loading the thread context. This situation leads to reduced task execution efficiency, especially involving task stealing. Accordingly, applications are suffering from adverse effects of the NUMA relationship among CCDs/chips.

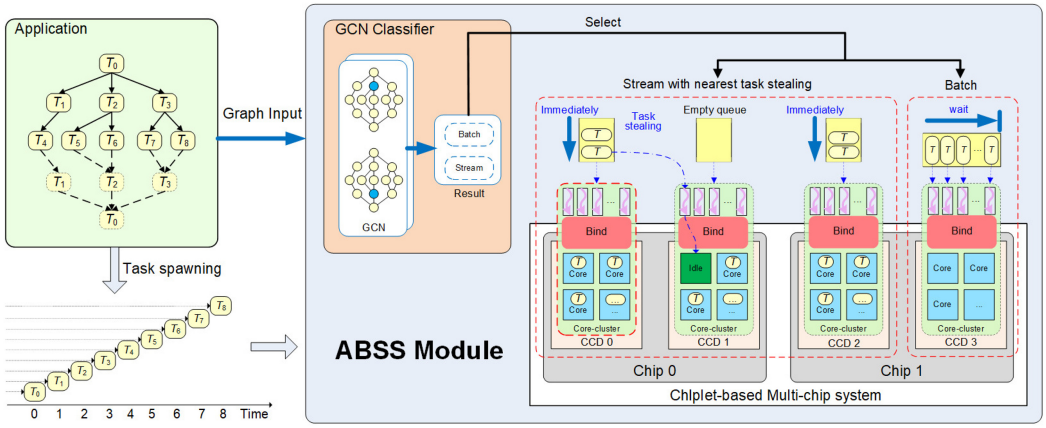



Fig. 1. The overall workflow of the proposed Adaptive Batch-Stream Scheduling (ABSS) module on the chiplet-based multi-chip systems. The ABSS module includes three optimization approaches: the adaptive batch-stream scheduling, the chiplet-based core-cluster binding, and the chiplet-based nearest task stealing.

The above challenges can be briefly summarized as:

(1) How to choose a suitable scheduling strategy for dynamic task parallelism on the chiplet-based multi-chip system?
(2) How to optimize thread execution behaviors on the chiplet-based multi-chip system to obtain the appropriate tradeoff between the reduced overhead of remote memory access and the utilization improvement of computing resources?
(3) How to adjust the placement of tasks in queues on the chiplet-based multi-chip system to reduce the impact of workload imbalance at run-time?

## 3 OUR PROPOSED ABSS MODULE

### 3.1 Overall Architecture

In this section, we design an **Adaptive Batch-Stream Scheduling (ABSS)** module on the chiplet-based multi-chip system to provide efficient parallelism services for dynamic tasks. ABSS provides performance optimization from three perspectives: adaptively batch-stream scheduling, chiplet-based core-cluster binding, and chiplet-based nearest task stealing. A GCN-based classifier is used to learn the execution features of the task parallelism, and adaptively select the suitable scheduling scheme between batch and stream scheduling for a given DAG of application. In addition, the chiplet-based core-cluster thread binding mechanism is designed to bind threads to a core cluster on a CCD instead of a core, which can improve the utilization of CCD's computing units, effectively reduce remote memory access, and narrow frequent context switching. On this basis, the chiplet-based nearest task stealing method for stream scheduling is further proposed, which steals tasks from the nearest CCDs for each idle core, achieves workload balance, and further reduces context switching. The overall workflow of the proposed ABSS module is shown in Figure 1.

### 3.2 Adaptive Batch-stream Scheduling

There are two task scheduling schemes for task parallelism, one is batch scheduling and the other is stream scheduling, as shown in Figure 2.

They have their own advantages and limitations in different scale applications and task types. The batch scheduling method is suitable for large-scale tasks, and the number of tasks is an integral

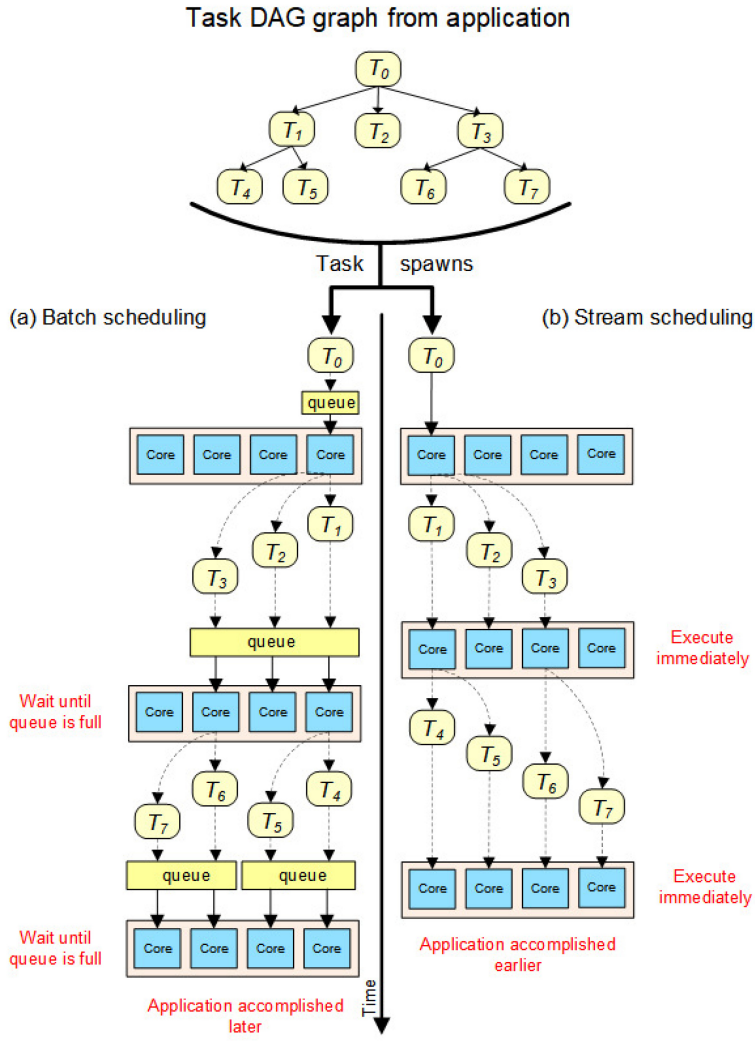



Fig. 2. Examples of batch scheduling and stream scheduling for tasks.

multiple of the number of cores. However, in applications with different task sizes or unignorable intervals, it may cause core idleness or unbalanced workloads among cores. In contrast, in a stream scheduling scheme, the scheduler will immediately respond to each arrived task and put it to an idle core or add it into the buffer queue. However, during scheduling each task, it is necessary to query and modify the available state of computing units, which leads to frequent reads and writes of the state variables of computing units. We experimented with a typical task parallelism application by using both batch and stream scheduling with several sparse matrices randomly selected from the UF datasets as the input and each of these datasets formed a corresponding DAG. We found that there are three situations from results summarized in Table 2: 1) stream scheduling is better than batch scheduling, 2) contrary to the former, and 3) both scheduling methods are pretty close (performance gap within 2%). According to our observation, we expect that tasks can be scheduled by the most appropriate scheduling method in parallel execution to achieve better overall performance. To adaptively select the appropriate scheduling scheme for a given

Table 2. SpQRF with Scheduling Method of the Datasets Randomly Selected

| SpMat | $m$ | $nnz$ | $den$ | $N_T$ | $t_b$ | $t_s$ | $pg$ |
|---|---|---|---|---|---|---|---|
| thermal | 3.4 | 66.5 | 0.55% | 22 | 34.01 | 42.31 | −19.6% |
| rajat06 | 10.9 | 46.9 | 0.04% | 197 | 19.59 | 22.41 | −12.6% |
| rdist2 | 3.2 | 56.8 | 0.56% | 186 | 16.44 | 16.72 | −1.7% |
| rdb968 | 1.0 | 5.6 | 0.60% | 27 | 3.48 | 3.43 | 1.5% |
| rdist3a | 2.4 | 61.9 | 10.76% | 94 | 18.16 | 15.78 | 15.1% |
| Wordnet3 | 82.6 | 132.9 | 1.95% | 15 | 5364.94 | 5012.68 | 7.1% |

$m$ is the size of each sparse matrix ($m/10^3$), and $nnz$ is the number of non-zero elements ($nnz/10^3$). $den$ represents the density ratio of a sparse matrix. $tb$ represents the execute time (millisecond, ms) with batch scheduling. $t_s$ represents the execute time (millisecond, ms) with stream scheduling. $pg$ represents the performance gap of the comparison scheduling methods, $pg = tb/ts - 1$, where $pg > 0$ represents the stream scheduling bring better performance.

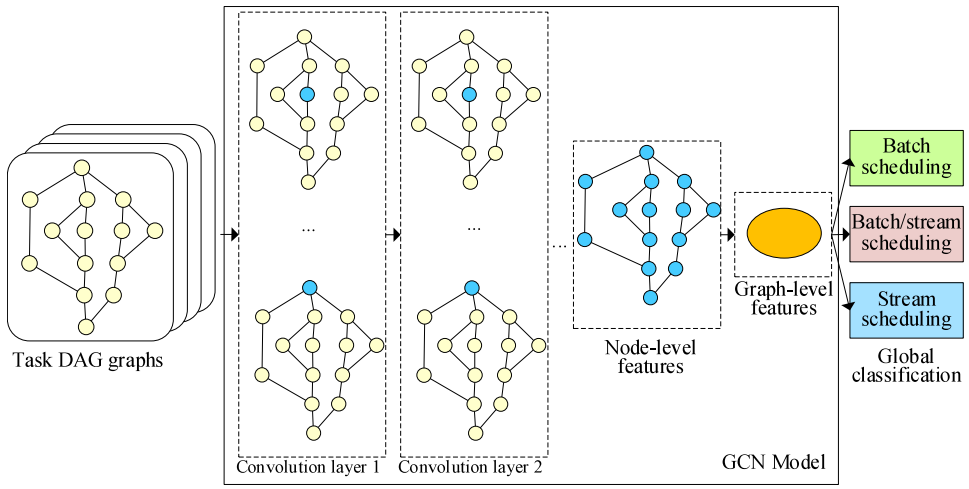



Fig. 3. Task scheduling scheme classification based on GCN model.

application according to the characteristics of the computing tasks, we analyze the corresponding DAG of the execution process on task parallelism, which can be obtained from explicit or implicit methods. For example, for a specific matrix decomposition function, the algorithm will form an elimination tree to represent the DAG for task parallelism [15]. Other statically executed code can be obtained through the compiler.

Due to the complexity of dynamic task parallelism, we hope to obtain a selection of an appropriate scheduling mode with the help of machine learning. Considering the feature of the DAG graph dataset and taking into account the tradeoff between effect and performance, we choose the Graph Convolutional Neural network (GCN) [25] as a machine learning classifier to intelligently predict the target scheduling options for each input DAG. The workflow of the GCN-based adaptive batch-stream scheduling is shown in Figure 3.

Let $G = (V, E)$ be a DAG graph of a given application, where $V$ is the set of nodes (i.e., computing tasks) and $E$ is the set of edges (i.e., logical or data dependencies between tasks). In addition, $X = G_1, G_2, \ldots, G_N$ represents a set of training samples, and $Y = y_1, y_2, \ldots, y_N$ are the corresponding classification labels, namely, the scheduling schemes of the input DAG graphs. Label $y_i$ is the

appropriate scheduling scheme for $G_i$, $y_i \in R^S = (B, S, BS)$, where $B$ is batch scheduling, $S$ is stream scheduling, and $BS$ represents that $G_i$ can be executed using batch scheduling or stream scheduling to achieve similar scheduling performance. Therefore, we can define the problem of task-scheduling scheme selection as a three-category classification problem.

To effectively implement the GCN model, it is crucial to assign suitable features to both vertices and edges. Vertices are assigned four distinct features: in-degree, out-degree, type-degree and type. Each vertex symbolizes a task, which has logical or data dependencies with other vertices in the task dependency graph. The in-degree feature represents the number of parent or predecessor tasks on which the task relies. The out-degree feature represents the number of child or successor tasks generated by the task. The type-degree feature represents the number of types of child or successor tasks generated by the task. The type feature refers to the type of tasks that are generated by the same parent and perform the same functional operation. Commonly, tasks of a same type are generated in batches by a specific "for" loop within the source code. These tasks are characterized by their concurrent spawning at short intervals and uniform execution times. Edges are directed, representing the sequential relationship in the execution process of the tasks. Edges are assigned one feature: the weight, which represents the likelihood of a successor's execution. The execution of a successor task may be conditional, contingent upon the fulfillment of specific criteria related to the actual performing of the predecessor task. Consequently, the weight feature is designed to represent the likelihood of executing the successor task. Given the challenge in ascertaining the precise probability of task execution prior to program execution, this weight feature is binary: a value of 1 indicates conditional execution, while a value of 0 denotes definite execution.

When the first deployment of ABSS module on a chiplet-based multi-chip system, the GCN classifier is trained via utilizing benchmark testing results as its training dataset. The training process of the GCN classifier is completed before the ABSS module starts optimizing the target programs, and the predicting process is carried out at runtime. To train the GCN classifier, we generate a large number of random task DAG graphs. For each graph, we respectively use batch and stream processing to execute it, choose a better performance scheduling scheme, and set it as the classification label of the graph. All of the graphs and the corresponding labels are used as the training dataset of GCN. Then, given a task parallelism programming application, the DAG graph $G_i$ is input to the GCN model. The GCN model can effectively collect the parallelism and dependencies of nodes and pass them to the neighborhood, and finally predict the task scheduling scheme. If the output $y_i$ of GCN is $BS$ or $S$, we will perform stream scheduling on $G_i$, otherwise, we will use batch scheduling. Based on the GCN-based classifier, we can adaptively select batch or stream scheduling schemes for different task programming applications to improve running performance.

### 3.3 Chiplet-based Core-cluster Binding

The behavior of threads in the system is ultimately executed by the OS. The user level cannot directly control the specific behavior of the thread, but we can still indirectly interfere with its behavior by setting the affinity of the thread to the computing core. The core-affinity strategy (i.e., thread binding to the core) is usually used in multi-core architecture. As opposed to no binding threads, the thread binding strategy can directly bind threads to computing cores, which is a common optimization for multi-core architectures. Binding threads to cores can significantly reduce the number of thread migrations among CPUs and reduce the number of cache invalidations, but it also completely limits the OS's ability to schedule threads. When threads bounded to a core are blocked, the core may have no threads available, resulting in idleness.

To avoid the above challenge, we propose a chiplet-based core-cluster binding mechanism that can establish an affinity between CCDs' core-cluster and threads. Threads can run as long as possible in the CCD cores without migrating to other CCD cores, which can reduce the impact of the

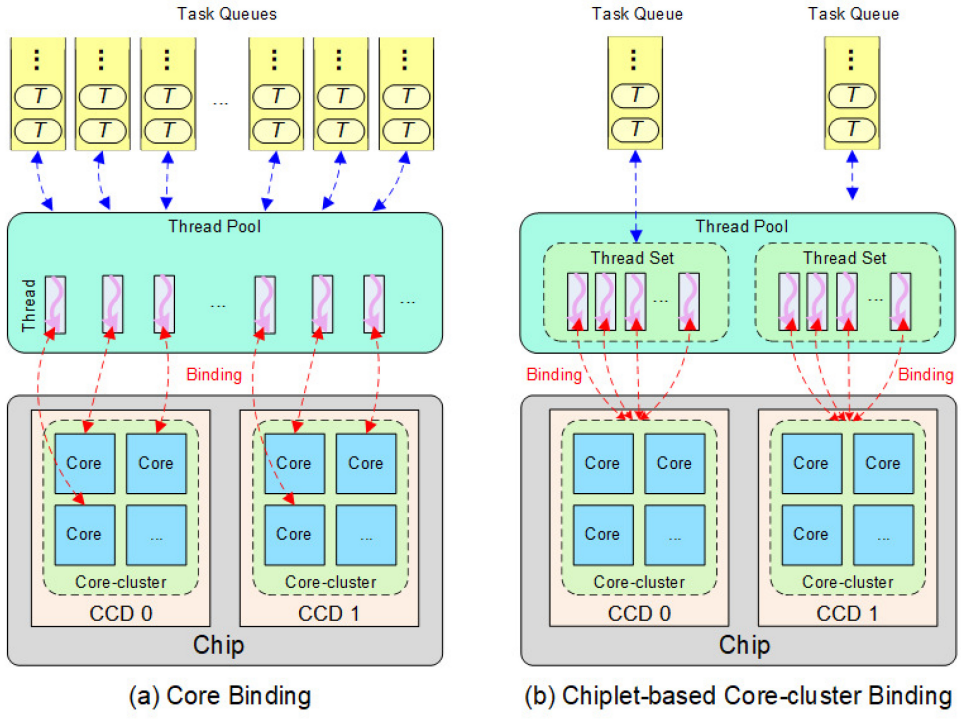



Fig. 4. Examples of core binding and chiplet-based core-cluster binding mechanisms.

NUMA relationship. Examples of core binding and chiplet-based core-cluster binding mechanisms are illustrated in Figure 4.

Assume that there are $k$ CCDs, and each CCD has $m$ cores. The task execution time of the $j$-th core on the $i$-th CCD is $T_{ij}$, where $i = 1, 2, \ldots, k; j = 1, 2, \ldots, m$. $T_{max}$ represents the longest task execution time among all cores. $T_{cmax-i}$ represents the longest task execution time among all cores in the $i$-th CCDs. $U_{\text{Core}}$ and $U_{\text{CCL}}$ represent the CPU utilization rate by using the core binding mechanism and chiplet-based core-cluster binding mechanism, respectively. For the core binding mechanism, the CPU utilization rate of the chiplet-based multi-chip system is:

$$U_{\text{Core}} = \frac{1}{km} \sum_{i=1}^{k} \sum_{j=1}^{m} \frac{T_{ij}}{T_{max}} = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{mT_{max}} \sum_{j=1}^{m} T_{ij}. \tag{2}$$

For the chiplet-based core-cluster binding mechanism, the CPU utilization rate of the chiplet-based multi-chip system is:

$$U_{\text{CCL}} = \frac{1}{k} \sum_{i=1}^{k} \frac{\sum_{j=1}^{m} T_{ij}}{mT_{cmax-i}} = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{mT_{cmax-i}} \sum_{j=1}^{m} T_{ij}. \tag{3}$$

It can be seen that $U_{\text{CCL}}$ is not less than $U_{\text{Core}}$, since $T_{max} = max\{T_{cmax-i} | i = 1, 2, \ldots, k\}$. Therefore, we use the chiplet-based core-cluster binding mechanism to bind threads to core-cluster of the CCD instead of individual cores. Unless all threads on the CCD are blocked, the cores on the CCD will not be idle. In addition, threads are controlled in a CCD, avoiding the migration of thread context between CCDs. In this way, it is possible to ensure the local access of the data used

when the tasks carried by the threads is interrupted and returned to the execution. Moreover, this flexible mechanism also gives the OS a certain amount of leeway. It also laid the foundation for us to further efficiently use computing core resources.

### 3.4 Chiplet-based Nearest Task Stealing

We propose the chiplet-based nearest task stealing method to automatically and efficiently balance the workload for the system thereby improving resource utilization and optimizing program performance. The chiplet-based nearest task stealing method is only effective after fixing the relative position relationship among threads, CCDs and memory, which relies on the chiplet-based core-cluster binding mechanism.

Initially, several preparations need to be done before the chiplet-based nearest task stealing method will work effectively. Firstly, once the chiplet-based core-cluster binding mechanism is working, threads are naturally grouped according to CCDs, and the relationship among threads can be represented by the NUMA distance [29]. A task queue is established for the group of threads of each CCD, and shared by all threads within the group. Secondly, a searching space is established according to the NUMA distance table, which stipulates the searching order for each thread to search task queues. Threads are required to prioritize the local CCD's task queue and extend to the most distant one if necessary.

During the time the chiplet-based nearest task stealing method is working, tasks are not only being passively stolen by other available threads, but also being actively pushed to other available threads. We discuss three situations as follows. (1) When a thread's executed task generates a new task and the thread's affiliated CCD has at least one idle computing core, the thread pushes this new task directly to the thread that can immediately execute it on the idle core. In this case, the new task does not need to go through the task queue. (2) When a thread's executed task generates a new task and the thread's affiliated CCD has no idle computing core, the thread then pushes the new task to the local task queue. (3) When a thread's executed task completes its task and does not spawn any extra task, the thread then firstly searches for an available task based on the established search order, starting from near to far. Once an available task is identified, the thread fetches and executes it. In the running state of the ABSS module, each thread continually repeats these processes. An example of the nearest task stealing is shown in Figure 5. There are two idle cores on $CCD_2$, and the task queue $Q_2$ of $CCD_2$ is empty. Since $CCD_3$ is the nearest neighbor of $CCD_2$ with the minimum NUMA distance $d_{2,3} = 16$, so the first idle core steals a task from $Q_3$. At this time, there are no more tasks available in $Q_3$. Therefore, the second idle core on $CCD_2$ can only find the second nearest neighbor $CCD_1$ ($d_{1,2} = 25$) and steals an available task from $Q_1$.

By adopting the chiplet-based nearest task stealing method, the total NUMA node migration distance of all tasks generated by a program throughout its entire execution process in the system is very small. Assume there are three task migration scenarios: The event $A$ represents that tasks only migrate among cores within the same CCD, corresponding to the mentioned situation (1) or (2). The event $B$ represents that tasks only migrate among CCDs within the same socket, and the event $C$ represents that stealing tasks migrate across sockets, corresponding to the mentioned situation (3). The NUMA distance between cores within the same CCD is represented as $d_1$. The NUMA distance between CCDs within the same socket is represented as $d_2$. The NUMA distance between sockets is represented as $d_3$, and $d_1 < d_2 < d_3$. The expectation of NUMA distance of tasks migration $E_d$ can be calculated as:

$$E_d = d_1 P(A) + d_2 P(B|\overline{A}) + d_3 P(C|\overline{A \cup B}). \tag{4}$$

Because of the chiplet-based nearest task stealing method, tasks and its parent task are usually executed in the same CCD. In HPC programs, the parent task usually needs to wait for the results

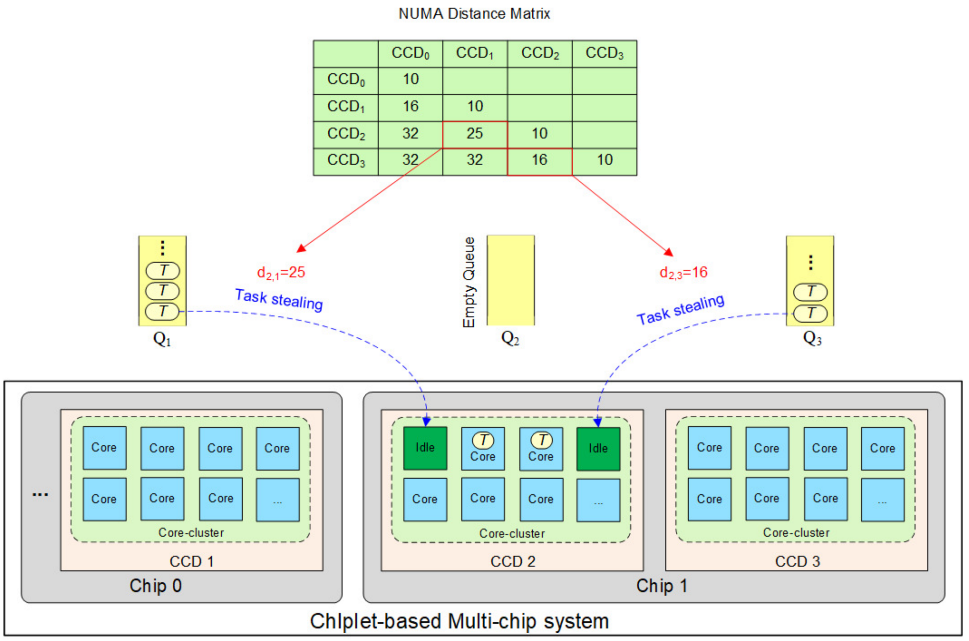



Fig. 5. The example of the chiplet-based nearest task stealing method.

returned by its spawned child tasks thereby it will release the computing resource during the waiting. At this situation, the idle computing resources can immediately execute another new task in the same CCD, resulting in a greater probability of occurrence of the event *A*. Therefore, the expectation value of the total migration distance is small. It indicates that the system can achieve an effective workload balance and meanwhile cost low transmission overhead.

### 3.5 Implementation

We implement the proposed ABSS module on a real-world chiplet-based multi-chip system of the HiSilicon Kunpeng 920 server. The module includes our contributions above and a POSIX-based implementation that supports scalable parallel programming. It is designed to support the parallelism of dynamic tasks on the chiplet-based multi-chip system and does not require a special language or compiler. Universal programming makes the ABSS module flexible and efficient. The example code of the SparseQR program from the SuiteSparse with using ABSS module APIs is shown in Figure 6. In the *main* function, the *ABSS_init* API and the *ABSS_finalize* API is needed for controlling the ABSS moudule starting and ending. In the *SparseQR* function, the *ABSS_selector* API utilizes the trained GCN classifier to select a scheduling scheme and stores the result as as a global variable for the *qr_factorize* function. The selection is according to the *task_graph* parameter obtained from the *qr_analyze* function. The *ABSS_addtask* API adds the *qr_factorize* function as a task to the ABSS module. The *ABSS_barrier* API ensures that all subtasks spawned by the *qr_factorize* task have been completed at this program point.

*3.5.1 Module Initialization.* Before the module works, it needs to be initialized by using the *ABSS_init*() interface. Multiple preparations will be executed for parallelism serving, including:

(1) Creating a thread pool to reduce the overhead of thread creation and destruction at runtime, which is a conventional technique for all parallel systems;

```c
int main(int argc, char* argv[])
{
    ......

    //Initializing the ABSS Module
    ABSS_init(SYSTEM_INFO);

    ......

    //Sparse QR factorization
    int err = SparseQR(...);

    ......

    //Finalizing the ABSS Module
    ABSS_finalize(SHUTDOWN_MODE);

    ......
}
```

```c
int SparseQR(...)
{
    ......

    //Obtaining the task dependency graph of Sparse QR factorization
    ABSS_DAG task_graph = qr_analyze(...);

    //Using ABSS GCN classifier to select a schduling scheme
    ABSS_selector(&qr_factorize, task_graph);

    //Adding the qr_factorize function as a task to ABSS module.
    ABSS_addtask(&qr_factorize, qr_factorize_params, barrier_tag);

    //Parallel tasks are synchronized here
    ABSS_barrier(barrier_tag);

    ......
}
```

Fig. 6. The example code with ABSS module APIs from the viewpoint of users. The left portion of the code displays the *main* function, which calls the *SparseQR* function depicted on the right.

**ALGORITHM 3.1:** *ABSS_StreamAddTask*() Interface

**Require:**
$t_i$: The current computing task;
$Q$: The task queues;
$T$: The thread pool;
**Ensure:**
$\hbar$: Execution status.
1: Get current CCD rank: $r \leftarrow getCCDRank()$;
2: Set variable and initialize: $idx \leftarrow r$;
3: Set a variable and initialize: $a = 0$
4: **repeat**
5: **for** $i$ in $T$ **do**
6: **if** $T[i]$ is idle **then**
7: $a \leftarrow i$;
8: **end if**
9: **end for**
10: **if** $a \geq 0$ **then**
11: Lock $T[a]$;
12: $T[a] \leftarrow t_i$;
13: Unlock $T[a]$;
14: Wake up $T[a]$;
15: **else**
16: $idx \leftarrow getNearestCCDRank(r)$;
17: **end if**
18: **until** $idx \geq 0$
19: Put the task into local task queue: $Q[idx] \leftarrow t_i$;
20: **return** $\hbar$.

(2) Grouping threads evenly according to the number of CCDs with the NUMA support of the Linux kernel, and these threads are bounded to CCDs instead of individual cores;
(3) Allocating memory space to each task queue in the local NUMA node, and assigning it to the corresponding CCD for task scheduling and chiplet-based nearest task stealing;
(4) Generating a search space table based on the NUMA distance matrix, and duplicating the table for each NUMA node;

(5) Creating an array to store and update markers of synchronization for the task parallelism.

*3.5.2 ABSS Module Interfaces and the Thread Action.* In our ABSS module, we develop several important interfaces to support task parallel programming:

(1) *ABSS_AddTask*() interface: a task is added and waiting to be scheduled as batch/stream way according to the judgment of the GCN classifier.
(2) *ABSS_BatchAddTask*() interface: a task is added to the local task queue, waiting for other tasks to arrive at the same level of the call tree.
(3) *ABSS_StreamAddTask*() interface: a task is added to the local task queue, waiting to be stolen by other threads or pushed to the nearest thread for immediate execution. The detailed processes is described in Algorithm 3.1.
(4) *ABSS_Barrier*() interface: an explicit blocking operation where execution must wait until all tasks of the current function branch to that point have returned.
(5) *ABSS_ThreadAction*() function: an implicit operation for thread controlling including all thread actions used in our ABSS module at runtime. The detailed processes is described in Algorithm 3.2.

**ALGORITHM 3.2** *ABSS_ThreadAction*() Function

**Require:**
$d$: Running state;
$Q_s$: Task queues in stream mode;
$Q_b$: Task queues in batch mode;
$A_{\text{avb}}$: The global array of the available threads number value on CCDs;
$A_{\text{syn}}$: The global array of synchronization;
**Ensure:**
$\hbar$: Execution status.

Get current CCD rank: $r \leftarrow$ *getCCDRank*();
Allocate a thread block on current CCD: $b \leftarrow$ *malcOnCCD*($r$);
Initialize the thread block : *initTB*($b$);
Set thread state in $b$ is idle: ($b$→state) ← IDLE;
Set a local variable: $s$ is idle: $s \leftarrow$ IDLE;
**repeat**
  Atomically update the available threads number value: $A_{\text{avb}}[r]$++;
  ($b$→state) ← $s$;
  **repeat**
    Sleep and wait to be awakened;
  **until** ($s \leftarrow$ ($b$→state)) is IDLE && $d$ is KEEP-RUNNING
  Atomiclly update the available threads number value: $A_{\text{avb}}[r]$- -;
  **if** $d$ is not KEEP-RUNNING **then**
    **break**;
  **end if**
  **if** $s$ is EXECUTION **then**
    Execute current task: *do*($b$→$t$);
    Atomiclly update the value of synchronization: $A_{\text{syn}}[(b \rightarrow p)]$- -;
    Update state: $s \leftarrow$ STEALING;
  **end if**
  **if** $s$ is STEALING **then**
    **if** *length*($Q_s[r]$) > 0 **then**
      Lock $Q_s[r]$;
      Get the task: ($b$→$t$) ← *pop*($Q_s[r]$);

```
25:            Unlock $Q_s[r]$;
26:            Execute current task: $do(b \rightarrow t)$;
27:            Atomiclly update the value of synchronization: $A_{\text{syn}}[(b \rightarrow p)]$- -;
28:        else
29:            $i \leftarrow getNearestCCDRank(r)$
30:            if $i \geq 0$ then
31:                Lock $Q_s[i]$;
32:                Get the task: $(b \rightarrow t) \leftarrow pop(Q_s[i])$
33:                Unlock $Q_s[i]$;
34:                Execute current task: $do\ (b \rightarrow t)$;
35:                Atomiclly update the value of synchronization: $A_{\text{syn}}[(b \rightarrow p)]$- -;
36:            end if
37:        end if
38:        Update state: $s \leftarrow$ IDLE;
39:    end if
40:    if $s$ is BATCHING then
41:        Execute current queue: $do(Q_b[b \rightarrow q])$;
42:        Atomiclly update the value of synchronization: $A_{\text{syn}}[(b \rightarrow q)]$- -;
43:        Update state: $s \leftarrow$ IDLE;
44:    end if
45: until True
46: return $\hbar$;
```

#### 3.5.3 *Explanation of Algorithm 3.2.*

(1) Algorithm 3.2 line 2: *malcOnCCD*($r$) interface is based on NUMA library, which allocates a space on the specified NUMA node.
(2) Algorithm 3.2 lines 9-11: This loop is a regular implementation for a thread pool, which can make the thread wake up here and prevent the thread from being awakened accidentally.
(3) Algorithm 3.2 lines 15-18: this code block makes the thread immediately execute its task when the thread state is set to EXECUTION. When the task completes, the thread state would immediately be set to STEALING.
(4) Algorithm 3.2 lines 19-34: this code block makes the thread to steal task from the nearest CCD. the *getNearestCCDRank*($r$) interface (line 27 at Algorithm 3.2 and line 14 at Algorithm 3.1) will search each CCD task queue according to the distance from near to far, and return the rank number of the first non empty queue. When the task completes, the thread state would immediately be set to IDLE. This represents the thread would steal the nearest task only once.
(5) Algorithm 3.2 lines 35-38: This code block makes the thread immediately execute the corresponding task in the batch queue, when the task accomplishes, the thread state would immediately be set to IDLE.

## 4 EXPERIMENTS

### 4.1 Experimental Setting

We implement the proposed ABSS module on the real-world chiplet-based multi-chip system of the HiSilicon Kunpeng 920 server [31], and conduct comparison experiments to evaluate the performance of the proposed model. The system settings of such chiplet-based multi-chip system are described in Table 3. The NUMA distance is obtained by the *"numactl -H"* command on Linux [22].

Table 3. System Settings of HiSilicon Kunpeng 920 Server

| Items | Values |
|---|---|
| Processor | Kunpeng 920 ARM v8.2 |
| Server | 2 Sockets; 4 CCDs (2 CCDs/Socket); 128 Cores (32 Cores/CCD) |
| Clock cycle | 2.6 GHz |
| Cache | 64KB L1 I Cache; 64KB L1 D Cache; 512KB L2 Cache; 32MB L3 Cache |
| NUMA | 4 Nodes; 256GB (64GB/Node) |
| Theoretical throughput | 1331.2Gflops |
| HPL benchmark | 2.2407 Gflops |

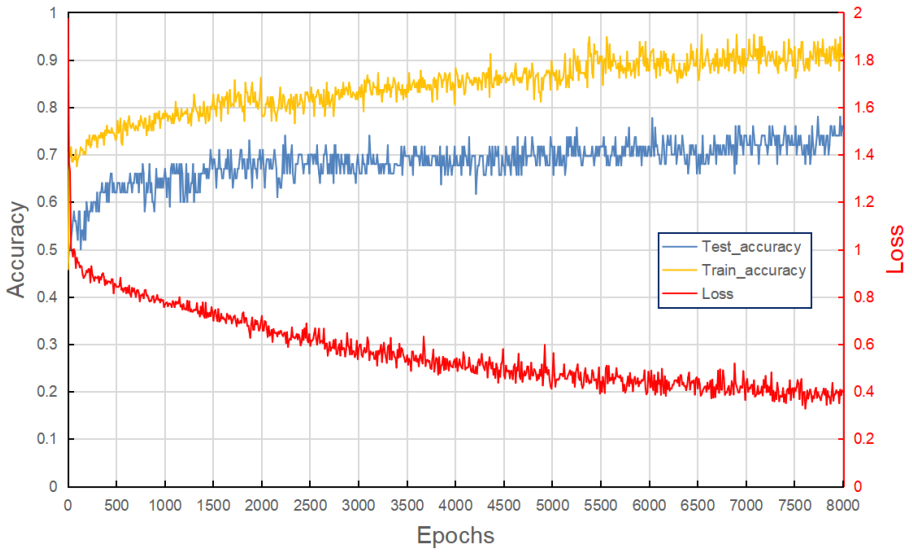



Fig. 7. The result of GCN classifier.

### 4.2 Experiments on GCN-based Classifier

All experiments in this section are conducted on Ubuntu 16.04.6 LTS and TITAN RTX-24G. The GCN model is implemented in Python 3.7, PyTorch 1.4.0, and PyTorch-geometric 1.6.0. To simulate the execution mode of dynamic tasks in HPC applications, we randomly generate 1,000 DAG graphs using BLAS Level 1 routine as the task load kernel. The maximum number of tasks in each DAG graph is 2,048. The task load changes hierarchically with task spawn, and the calculation time of workload conforms to a normal distribution. The experimental results of the GCN-based classifier are shown in Figure 7.

The training accuracy of the GCN model is shown in the yellow curve in Figure 7, the test accuracy is the blue curve and the loss is the red curve. When the epochs reach 6,500, the loss gradually converges to 0.4, the training accuracy of the model is about to 90%, and the test accuracy of the model is about to 72%. The experimental results show that the GCN model can accurately predict the appropriate scheduling across different programs and realize adaptive batch-stream scheduling.

### 4.3 Optimization Results of Chiplet-based Core-cluster Binding Mechanism

*4.3.1 Benchmark.* We migrate the **General Matrix Multiplication (GEMM)** routine from ATLAS [28] to our ABSS module, and use this routine as our kernel routine of DAG. As a basic routine

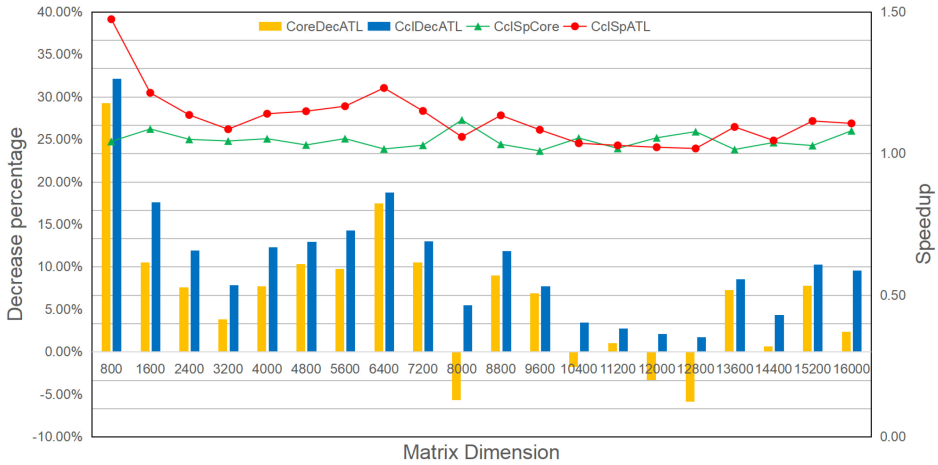



Fig. 8. Histogram of the performance comparison results among different core binding mechanisms.

of BLAS level 3, GEMM has been widely used in various HPC applications. The parallel load of the GEMM routine is more balanced. It can reduce the fluctuations in test performance caused by unbalanced workloads. We believe the GEMM routine can reflect the difference in thread affinity strategy to the greatest extent. We measure the performance of GEMM with our ABSS module parallelization and the performance of the prototype ATLAS GEMM.

We generate 20 random square matrices with a dimension from 8,00 to 16,000 (interval of 800) and use them to test each routine for analysis. Since the block size of ATLAS GEMM is $80 \times 80$, we choose these dimensions that are multiples of 80 to match the block size and reduce the impact of the data cache hit rate. We test each routine 20 times and then calculate the average execution time in order to reduce the error caused by fluctuations in system performance. The prototype of ATLAS GEMM is without any thread binding strategy, and we note it as *AtlGEMM*. The modified GEMM with using our ABSS module parallelization adopts two thread binding mechanisms: core binding mechanism noted as *CoreGEMM* and chiplet-based core-cluster binding mechanism noted as *CclGEMM*.

*4.3.2 Performance Evaluation on the GEMM Routine.* Figure 8 shows the performance comparison results among *AtlGEMM*, *CclGEMM*, and *CoreGEMM*. In this figure,

(1) The *CoreDecATL* represents the percentage of time decrease as *CoreGEMM* to *AtlGEMM*.
(2) The *CclDecATL* represents the percentage of time decrease as *CclGEMM* relative to *AtlGEMM*.
(3) The *CclSpCore* represents the speedup (Sp) of *CclGEMM* relative to *CoreGEMM*,
(4) The *CclSpATL* represents the speedup (Sp) of *CclGEMM* relative to *AtlGEMM*.

Our ABSS module has noticeable optimization effects on the prototype ATLAS GEMM routine, and the average optimization effect is greater than 8.00%. All the *CclGEMM* performances are better than *coreGEMM* or *AtlGEMM*, while the *coreGEMM* performances are worse than *AtlGEMM* in few cases. It shows that under the chiplet-based multi-chip system, the core binding mechanism is not necessarily better than no binding strategy and worse than chiplet-based core-cluster binding mechanism.

We analyze these GEMM routines from page faults and CPU migration count perspectives. In Figure 9,

(1) The *CclDecPF* represents the decrease percentage of the page fault counts for matrices running in *CclGEMM* relative to *AtlGEMM*.

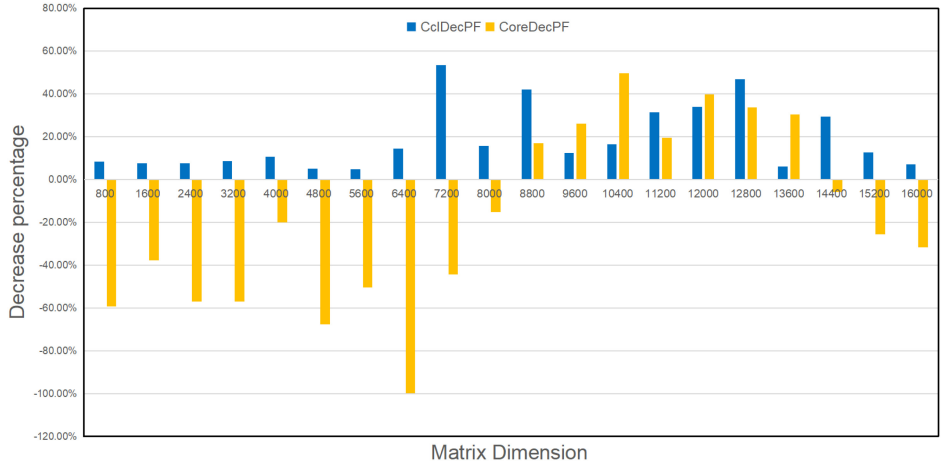



Fig. 9. Histogram of the impact of different binding mechanisms on page faults.

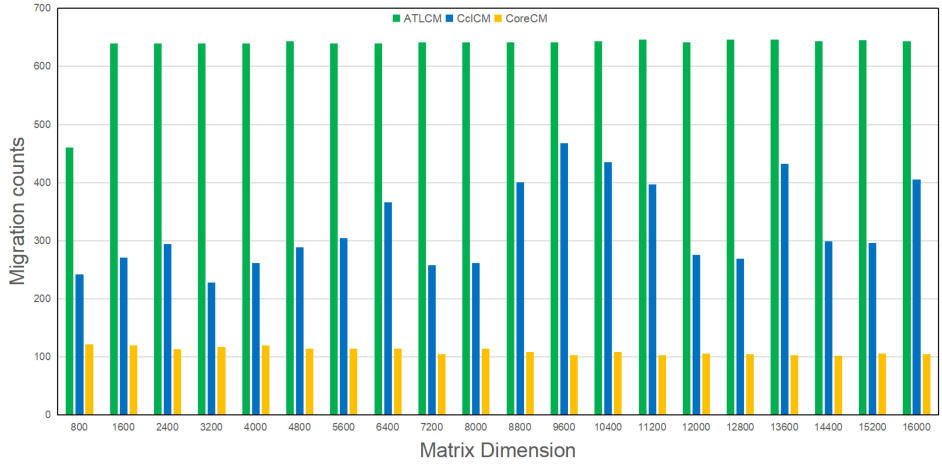



Fig. 10. Histogram of the impact of different binding mechanisms on CPU migration counts.

(2) The *CoreDecPF* represents the decrease percentage of the page fault counts for matrices running in *CoreGEMM* relative to *AtlGEMM*.

In all cases, the *CclGEMM* with chiplet-based core-cluster binding mechanism results in a decrease in page faults on matrices, while in several cases, the *coreGEMM* with core binding mechanism causes an increase in page faults, as shown in Figure 9.

Figure 10 shows the impact of different binding mechanisms on the CPU migration count, that is, the thread migration count across CPU cores. The *ATLCM*, *CoreCM* and *CclCM* in Figure 10 respectively represent that CPU migration counts for matrices running in *AtlGEMM*, *CoreGEMM* and *CclGEMM*. The thread-bound in the CCD-based core-cluster mechanism is freer than the thread-bound in the core, and will not cause excessive CPU migration.

### 4.4 Optimization Results of ABSS Module

*4.4.1 Benchmark.* Firstly, we use the DAG diagram that we generated in Section 4.2 to simulate the dynamic tasks of HPC applications on chiplet-based multi-chip system. We evaluate our ABSS

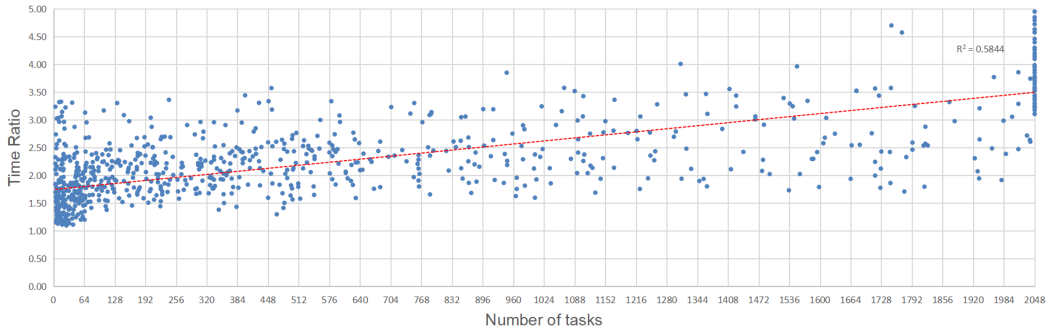



Fig. 11. The scatter plots of time ratio (TBB/ABSS) by the number of tasks, where $y > 1$ represents that the ABSS module can bring better performance and the red dotted line indicates the trend of performance changing with the number of tasks increasing.

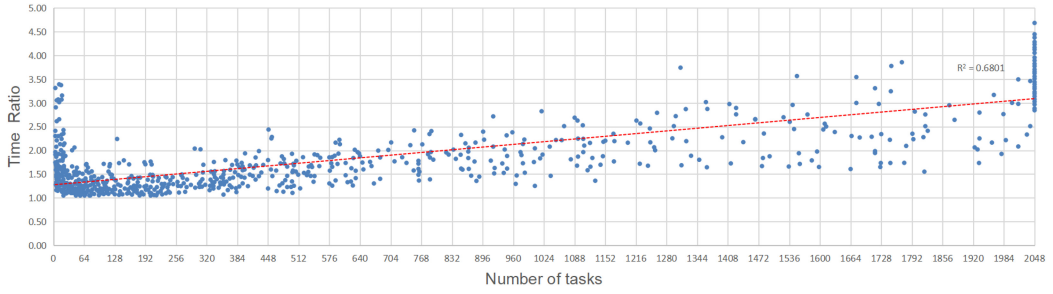



Fig. 12. The scatter plots of time ratio (OpenMP/ABSS) by the number of tasks, where $y > 1$ represents that the ABSS module can bring better performance and the red dotted line indicates the trend of performance changing with the number of tasks increasing.

module, TBB, and OpenMP for parallelism on the 1,000 DAG diagrams, respectively. Then, we compare ABSS with TBB and OpenMP, and report the performance comparison by the number of tasks.

Secondly, to truly reflect the advantages of the proposed ABSS module in dynamic task performance on the chiplet-based multi-chip HPC system, we use the widely-used benchmarks, such as the SpQRF [2] routine, SpLQF routine, SpLUF routine and SpCHF routine from the **Linear Algebra Package (LAPACK)**. These routines in SuiteSparse (version:5.7.2) use an elimination tree to organize tasks, which is a typical parallelism issue of dynamic tasks. These routines perform parallel execution by using Intel TBB for dynamic task parallelism. We replace Intel TBB with our ABSS module to realize the parallelism of these routines.

*4.4.2 Performance Evaluation on Random DAG.* Figure 11 and Figure 12 show the performance comparison by the number of tasks, where the blue dots present the time ratio (TBB/ABSS and OpenMP/ABSS). We use the least square method to realize linear regression. The $R^2$ of the fitted regression line in the figures is known as the determination coefficient, which represents the statistical measure of the distance between the data and the fitted regression line. In general, if $R^2$ is greater than 0.5, the fitting regression line would indicate the trend of the $y$ value changing with the increase of the $x$ value. All blue dots in both Figure 11 and Figure 12 are above the $y = 1$ line, which indicates our ABSS module has better performance than TBB and OpenMP. In addition, the red fitted regression lines in both Figure 11 and Figure 12 indicate that as the number of

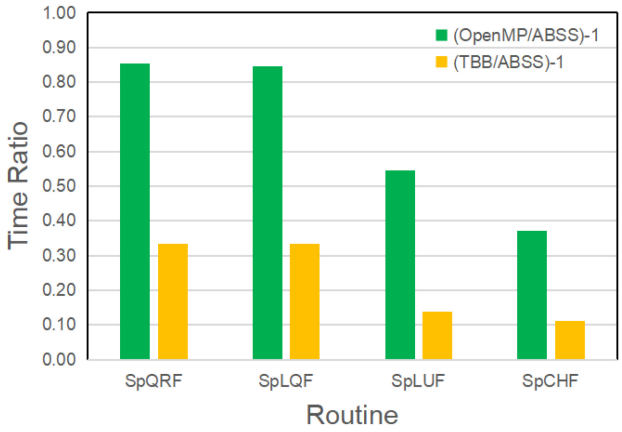



Fig. 13. Histogram of spare matrix factorization routines including SpQRF, SpLQF, SpLUF, and SpCHF. The green areas represent the average time ratio ((OpenMP/ABSS)-1), and the yellow areas represent the average time ratio ((TBB/ABSS)-1).

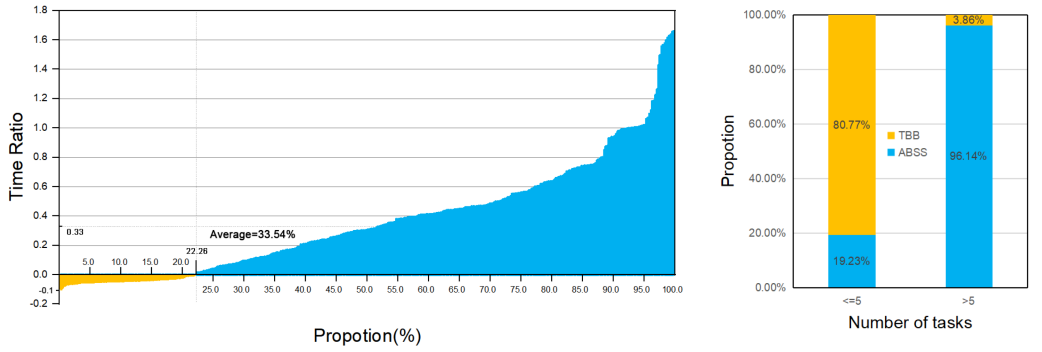
tasks increases, ABSS brings a wider range of performance improvements than the comparison methods.

*4.4.3 Performance Evaluation on Sparse Matrix Factorization Routines.* We randomly select the square matrices from the UF dataset [13] and evaluate the execution time of the sparse matrix factorization routines to compare the effectiveness of our ABSS module and the TBB method. Figure 13 shows that using our ABSS, compared with OpenMP, the average performance of the four routines is improved by 85.3%, 84.5%, 54.6%, and 37.1%, respectively, and compared with TBB, the average performance is improved by 33.5%, 33.3%, 13.9%, and 11.2%, respectively.

Additionally, we also focus on SpQRF for further experiments to analyze the optimization scope of our ABSS module. There are 408 square matrices in the UF dataset, and the maximum number of parallel tasks generated at runtime is 224 and the minimum number is 2. We rank the results of these 408 tasks according to the optimization range, as shown in Figure 14(a). It can be seen from the yellow zone in Figure 14(a) that 22.26% of the tasks that use TBB achieve better performance, but the performance gap is less than 10%. The blue area shows that 77.74% of the tasks that use our ABSS module achieve better performance. The average optimization range of our ABSS module is 33.54%, and the peak is 165.8%. In Figure 14(b), we further analyze the number of tasks that affect the optimization scope of ABSS. The results show that when the number of tasks is less than 5, only 3.86% of the tasks can achieve better performance by using ABSS. However, when the number of tasks is greater than 5, more than 80% of the tasks achieve better performance by using ABSS.

## 5 RELATED WORK

Various studies have been continuously exploring the NUMA architecture to reduce the different effects caused by the NUMA phenomenon [7, 18, 23]. For example, the ForestGOMP solution in [7] groups threads that share data in the memory hierarchy. In [18], Popov and Jimborean explored thread mapping, data mapping, and parallelism across stages in different applications and systems. In [23], Terboven et al. provided hints to the OpenMP compiler and runtime to guide the assignment of tasks to threads, which improves data placement and memory locality on NUMA. Although the above research has contributed to the distributed processor of the NUMA architecture, it doesn't consider the chiplet-based multi-chip system. In the parallelism solutions to dynamic




(a) (b)

Fig. 14. (a) Histogram of time ratio ((TBB/ABSS)-1) by the datasets proportion, where $y > 0$ represents that the ABSS module can bring better performance. (b) Histograms of task proportion by the number of tasks.

tasks, the complex interaction between these factors makes it difficult to determine the optimal task-to-core assignment. In [10], Clet-Ortega et al. evaluated different task scheduling strategies for OpenMP on large multicore NUMA systems and designed a configurable task scheduler. This approach is well-suited for scenarios where users possess a clear understanding of the optimal strategy for each specific application, enabling them to pre-configure the task scheduler statically prior to execution. However, in the context of applications characterized by dynamic task parallelism, the emergence of parallel tasks during runtime can vary based on the actual executing situation, making it difficult for users to pre-determine an optimal scheduling strategy. To address this challenge, our work provides the ABSS approach for optimizing dynamic task parallelism.

For the schedulability of DAG tasks executed by a thread pool, various studies have been conducted on multiprocessors [8, 21]. In [8], Casini et al. focused on the issues when using thread pools to execute parallel tasks, such as the performance degradation of deadlock and blocking synchronization mechanism. Schmid and Mottok presented a response time analysis of DAGs, which did not block the subtask scheduler, nor did it have the possibility to limit the number of threads per task [21]. To achieve workload balancing between cores or core clusters, Wang et al. implemented a work-stealing runtime to achieve dynamic task parallelism on heterogeneous cache-coherent systems [26]. In [24], Tzannes et al. obtained the adaptive task granularity of parallel loops, and it is necessary to check whether the task queue is empty before executing the iteration. In [33], Zhang et al. presented an automated HPC batch job scheduler based on reinforcement learning, which is used to solve the adaptability problem of batch job schedulers. In [20], Schmid et al. presented a proof-of-concept parallelism framework, which realizes static memory allocation in a work-stealing environment on the embedded multicore architecture.

The above studies have made outstanding contributions to optimization or implementation from different perspectives. Different from the existing work, we propose a chiplet-based nearest task stealing method to automatically and efficiently balance the workload of dynamic tasks and avoid frequent context switching of threads. We also focus on how to combine the characteristics of chiplet-based multi-chip systems to generate parallel execution solutions for dynamic task parallelism.

## 6 DISCUSSION ON SCALABILITY

Most Large-scale HPC programs are characterized by dynamic task parallelism. A prime example is CitcomCu [32], a widely-used numerical simulation software in the field of geodynamics, which

includes numerous tasks for simulating the evolution of geological area. These tasks exhibit both data and temporal dependencies. Similarly, in the realm of machine learning, large-scale applications utilize computation graphs for training and inference, which essentially function as task DAGs. Consequently, the optimization of such applications, featuring dynamic task parallelism in multi-node cluster computing systems, is also the optimization target of the ABSS module. The ABSS module is capable of scaling from chiplet-based multi-chip systems to multi-node cluster computing systems. This adaptability is particularly relevant in the context of complex network topologies within multi-node cluster computing systems. In such systems, nodes with higher network bandwidth can be grouped as a communication cluster, analogous to the CCD concept mentioned in the paper. This grouping results in reduced communication delays within the same cluster, while inter-cluster communications are subject to higher delays. In ABSS module, both inter-cluster and intra-cluster communication delays can be still quantified via using a distance matrix. The distance matrix tends to be larger in size, reflecting the complexity of the network topology. By employing the chiplet-based nearest task stealing method, which utilizes the distance matrix to establish the searching space, the ABSS module can still effectively reduce the communication overhead across the entire system.

In such system, the ABSS module can still adopt the adaptive batch-stream scheduling method, training a GCN classifier to select the most suitable scheduling scheme. The batch scheduling strategy aims to minimize the costs associated with task allocation and status querying. However, in multi-node environments, this strategy may lead to load imbalances, thereby increasing the cost of global synchronization, a challenge that is more pronounced than in the chiplet-based multi-chip systems. Conversely, the stream scheduling strategy can effectively reduce the global synchronization overhead within the system. However, it necessitates frequent process state synchronization, potentially increasing the costs of task execution and result collection, and leading to reduced network bandwidth utilization. Both batch and stream scheduling strategies exhibit distinct advantages and disadvantages in the context of multi-node cluster computing. The ABSS module can construct a larger GCN model to automatically and adaptively select the most suitable scheduling strategy for programs thereby enhancing the efficiency and effectiveness of the scheduling process in such complex computing environments under multi-node cluster systems.

## 7 CONCLUSION

In this work, we considered the dynamic task parallelism in the emerging chiplet-based multi-chip system, and proposed an adaptive batch-stream scheduling (ABSS) module with three optimization approaches to promote performance improvement. The GCN-based scheduling classifier method was firstly used to automatically select the best batch-stream scheduling solution for a given application. Then, the chiplet-based core-cluster binding mechanism was designed to establish affinity between CCDs and threads, so as to improve the utilization of computing cores. We further proposed the chiplet-based nearest task stealing method for automatically and efficiently balancing the workload for chiplet-based multi-chip systems. We implemented the proposed ABSS module on the HiSilicon Kunpeng-920 chiplet-based multi-chip system. The experimental results show that ABSS achieves higher performance acceleration than the comparison parallelism solutions.

## REFERENCES

[1] U. A. Acar, A. Charguéraud, and M. Rainey. 2013. Scheduling parallel programs by work stealing with private deques. In *PPoPP'13*. 219–228.

[2] P. R. Amestoy, I. S. Duff, and C. Puglisi. 1996. Multifrontal QR factorization in a multiprocessor environment. *Numer. Linear Algebr. Appl.* 3, 4 (1996), 275–300.

[3] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2008. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst.* 20, 3 (2008), 404–418.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. 2008. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems* 20, 3 (2008), 404–418.

[5] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2020. Groute: Asynchronous multi-GPU programming model with applications to large-scale graph processing. *ACM Transactions on Parallel Computing (TOPC)* 7, 3 (2020), 1–27.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN N.* 30, 8 (1995), 207–216.

[7] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Nam. 2010. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *Int. J. Parallel Program.* 38, 5 (2010), 418–439.

[8] Daniel Casini, Alessandro Biondi, and Giorgio Buttazzo. 2019. Analyzing parallel real-time tasks implemented with thread pools. In *Proceedings of the 56th Annual Design Automation Conference 2019*. 1–6.

[9] Milind Chabbi, Abdelhalim Amer, and Xu Liu. 2020. Efficient abortable-locking protocol for multi-level NUMA systems: Design and correctness. *ACM Transactions on Parallel Computing (TOPC)* 7, 3 (2020), 1–32.

[10] Jérôme Clet-Ortega, Patrick Carribault, and Marc Pérache. 2014. Evaluation of OpenMP task scheduling algorithms for large NUMA architectures. In *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings 20*. Springer, 596–607.

[11] Huawei Technologies CO. 2023. Kunpeng Math Library. (2023). http://www.hikunpeng.com/developer/boostkit/library/math

[12] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. *ACM SIGPLAN N.* 48, 4 (2013), 381–394.

[13] Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1–25.

[14] C. E. Leiserson. 2010. The Cilk++ concurrency platform. *J. Supercomput.* 51, 3 (2010), 244–257.

[15] Shengle Lin, Wangdong Yang, Haotian Wang, Qinyun Tsai, and Kenli Li. 2021. STM-multifrontal QR: Streaming task mapping multifrontal QR factorization empowered by GCN. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[16] T. Lovett and R. Clapp. 1996. STiNG: A CC-NUMA computer system for the commercial marketplace. In *ISCA'96*. 308–317.

[17] R. Maddox and R. J. Safranek. 2009. Introduction to Intel QuickPath Interconnect. *High Performance Multi-Core Processor Fabric* (2009).

[18] M. Popov and A. Jimborean. 2019. Efficient thread/page/parallelism autotuning for NUMA systems. In *ISC'19*. 342–353.

[19] J. Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc.

[20] Michael Schmid, Florian Fritz, and Jürgen Mottok. 2022. Fine-grained parallelism framework with predictable work-stealing for real-time multiprocessor systems. *Journal of Systems Architecture* 124 (2022), 102393.

[21] Michael Schmid and Jürgen Mottok. 2021. Response time analysis of parallel real-time DAG tasks scheduled by thread pools. In *29th International Conference on Real-Time Networks and Systems*. 173–183.

[22] Harsha Vardhan Simhadri, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Aapo Kyrola. 2016. Experimental analysis of space-bounded schedulers. *ACM Transactions on Parallel Computing (TOPC)* 3, 1 (2016), 1–27.

[23] Christian Terboven, Jonas Hahnfeld, Xavier Teruel, Sergi Mateo, Alejandro Duran, Michael Klemm, Stephen L. Olivier, and Bronis R. de Supinski. 2016. Approaches for task affinity in OpenMP. In *IWOMP'16*. Springer, 102–115.

[24] Alexandros Tzannes, George C. Caragea, Rajeev Barua, and Uzi Vishkin. 2010. Lazy binary-splitting: A run-time adaptive work-stealing scheduler. *ACM SIGPLAN Not.* 45, 5 (2010), 179–190.

[25] Haotian Wang, Wangdong Yang, Renqiu Ouyang, Rong Hu, Kenli Li, and Keqin Li. 2023. A heterogeneous parallel computing approach optimizing SpTTM on CPU-GPU via GCN. *ACM Transactions on Parallel Computing* 10, 2 (2023), 1–23.

[26] MoyangWang, Tuan Ta, Lin Cheng, and Christopher Batten. 2020. Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems. In *ISCA'20*. IEEE, 173–186.

[27] Tianqi Wang, Fan Feng, Shaolin Xiang, Qi Li, and Jing Xia. 2022. Application defined on-chip networks for heterogeneous chiplets: An implementation perspective. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 1198–1210.

[28] R. Whaley and J. Dontarra. 1998. Automatically tuned linear algebra software. *IEEE* (1998).

[29] S. Williams, L. Ionkov, and M. Lang. 2017. NUMA distance for heterogeneous memory. In *MCHPC'17*. 30–34.
[30] Yibo Wu, Liang Wang, Xiaohang Wang, Jie Han, Jianfeng Zhu, Honglan Jiang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2022. Upward packet popup for deadlock freedom in modular chiplet-based systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 986–1000.
[31] Jing Xia, Chuanning Cheng, Xiping Zhou, Yuxing Hu, and Peter Chun. 2021. Kunpeng 920: The first 7nm chiplet-based 64-core ARM SoC for cloud services. *IEEE Micro* PP, 99 (2021), 1–1.
[32] Jin Yang, Wangdong Yang, Ruixuan Qi, Qinyun Tsai, Shengle Lin, Fengkun Dong, Kenli Li, and Keqin Li. 2023. Parallel algorithm design and optimization of geodynamic numerical simulation application on the Tianhe new-generation high-performance computer. *The Journal of Supercomputing* (2023), 1–32.
[33] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: An automated HPC batch job scheduler using reinforcement learning. In *SC'20*. IEEE, 1–15.

Received 25 July 2023; revised 28 November 2023; accepted 23 January 2024