Proposal For ASC 25

Tuboshu Eliminators Team

Li Wangyang
Jin Chenye
Cui Shuyang
Zhang Guangqi
Liu Guanyu
Guo Dongliang (Advisor)

Feb 21, 2025

A proposal submitted to the ASC 25 committee

Contents

1 Brief Background Description of Supercomputing Activities

1.1 Hardware and Software Platforms

Our university established a high-performance computational (HPC) cluster in 2017, named Super Center Center of Yanshan University, through the integration of supercomputer resources in the university. The center contains a total of 36 nodes, 896 available computing cores, a total of 2.176TB of memory and 33TB of available storage capacity. Its theoretical double-precision floating-point performance reaches 28.672 TeraFLOPs. At Yanshan University, it is the highest-performance high-performance computing cluster with the largest computing capacity, the highest computing power and the most professional operation and maintenance team. It provides high-performance computing environment support for national and provincial-level scientific research projects undertaken by the research teams of other colleges. As of March 9, 2018, the total cost of the supercomputer CPU was 2963096203 seconds and 335 I jobs were scheduled. At present, after more than six months of trial operation, our school has accumulated rich experience in daily operation management, technical support, application services and personnel training of supercomputer centers.

The supercomputing center adopts a mature and general system architecture, which can be used to compute core 896 Cores, total memory of 2.112tb, and 33TB of storage capacity, which can be applicable to various types of applications. The main configurations are as follows:

Table 1: The main configurations

Type	Count	Name	Description
Management	1	mgt1	System monitoring and job scheduling
Login node	1	login	
Compute node	30	node1 30	2 Intel E5-2683v3 processors, 28 Cores 2.0Ghz, 64GB DDR4 ECC REG memory, Infiniband QDR 40Gb/s network
GPU Node	2	GPU1,2	2 Intel E5-2683v3 Processors, 28 Cores 2.0Ghz, 128GB DDR4 ECC REG memory, Infiniband QDR 40Gb/s network, NVIDIA TitanX GPU
IO node	2	gpfs1,nfs	GPFS parallel file system (18TB), NFS network storage

The Supercomputing center supports the running of computing software such as Ansys, Fluent, Vasp, Lammps, Comsol, and supports the compiling and running environment of C(C++), Fortran and other languages to ensure the computational requirements of self-compiled applications. Software resources are listed as follows:

Table 2: Software resources

Software Name	Description
Centos7.2	Operating System Platform
NFSv4	Web file system
MPICH/MPICH2/OpenMPI	Open source parallel development environment
Paramon Paratune	Application of running feature collector Application of running feature analyzer
Intel Parallel Studio XE 2015	Intel Parallel Development Suite
Intel MKL 2017	Intel Library of Mathematical Functions
Intel MPI 2017	Intel Parallel Messaging Library
IBM Platform Computing LSF	Resource Management and Operational Movement Control System
IBM GPFS	Common Parallel Document System
Openmpi 2.0.2	Open source Intel Parallel Messaging Library
FFTW	FFTW (Fastest Fourier Transform in the West)
gromacs-5.1.4	Molecular dynamics program

1.2 Supercomputing Courses and Groups

Parallel Processing and Architecture (Software Engineering Major)

Course Code: - **Credit Hours:** 32 **Prerequisites:** Computer Architecture, Algorithms

Course Objectives

This graduate-level course is a core component of computational science and technology, designed to address the rapid evolution of high-performance computing (HPC) and meet the demand for cultivating advanced technical talent. The course aims to:

- Introduce hardware foundations of parallel computing architectures (e.g., multi-core CPUs, GPUs, and interconnects).
- Explore core concepts in parallel computation, including task/data parallelism, load balancing, and synchronization.
- Familiarize students with HPC software ecosystems (MPI, OpenMP, CUDA) and tools for performance profiling.
- Develop skills in designing and implementing parallel algorithms for scientific and engineering applications.
- Foster self-directed learning through hands-on projects and literature reviews.
- Cultivate professional ethics, teamwork, and leadership skills through collaborative projects.

Parallel Computing (Computer Science & Technology Major)

Course Code: 04110360 **Credit Hours:** 32 **Prerequisites:** Algorithms, Computer Architecture

Learning Outcomes

By completing this course, undergraduates will achieve:

- **Knowledge:**
  - Master parallel architectures (SIMD, MIMD) and memory models (UMA, NUMA).
  - Understand parallel programming paradigms (shared vs. distributed memory).
  - Analyze scalability and efficiency of parallel algorithms using Amdahl's Law.
- **Skills:**
  - Develop optimized code for multi-core CPUs (pthreads, OpenMP) and GPUs (CUDA).
  - Debug and profile applications using tools like ParaView and Intel VTune.
  - Write technical reports and present results in academic formats.
- **Professional Competencies:**
  - Lead or collaborate in team projects mimicking real-world HPC scenarios.
  - Adhere to ethical standards in computational resource usage.

Our school has several student-led HPC interest groups, including parallel computing group, algorithm design and optimization group, Linux cluster construction group and test group. In the parallel computing group, the leading teachers teach relevant knowledge, and students learn parallel programming, C language, MPI, cluster management and other parallel computing knowledge independently. In addition, our school has set up a discussion group for students who like supercomputers. We learn from each other in the discussion group. Students who have experience in ASC or other super computing competitions actively share their experience. When they encounter problems that cannot be solved, they will discuss and seek solutions in the group.

1.3 Supercomputing-related Research and Applications

In July 2017, Yanshan University set up the Super Computing Center of Yanshan University through the integration of supercomputer resources in the university . The center contains a total of 36 nodes, 896 available computing cores , a total of 2.176TB of memory and 33TB of available storage capacity. Its theoretical double-precision floating point performance peaks at 28.672 trillion times per second.

At Yanshan University, it is the highest-performance high-performance computing cluster with the largest computing capacity , the highest computing power and the most professional operation and maintenance team . It provides high-performance computing environment support for national and provincial-level scientific research projects undertaken by the research teams of other colleges.

As of October 14, 2020, the total cost of the supercomputer CPU was 35267617706 seconds and 56809 jobs were scheduled. Near 100 papers have been published with the help from Super Computing Center of Yanshan University.

1.4 Key Achievements in Supercomputing Research

a) In the 2022 World University Supercomputer Competition (ASC19) and ASC24, the teams of our school won the second-class award in the world.

b) A team of our school successfully won the bronze prize of parallel optimization in the 8th Intel Cup Parallel Application Challenge (PAC2020).

2 Team Introduction

2.1 Team Setup

Our five-member team from the university's HPC research group combines fresh perspectives (one freshman) with advanced technical maturity (four juniors). Prior to ASC2025, we honed our expertise through collaborative study in parallel programming, C, MPI, and distributed systems. Guided by experienced ASC alumni, we solidified our multidisciplinary team where each member contributes specialized skills in system optimization and algorithm design. Our complementary strengths in HPC architecture and performance tuning enable effective problem-solving through close collaboration.

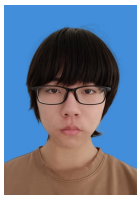
2.2 Team Members

Li Wangyang

Experienced in high-performance computing and parallel programming.

Jin Chenye

Skilled in algorithms and expertise in GPU cluster operations.



Cui Shuyang

Specializes in HPC testing, parallel computing, and AI algorithms.

Liu Guanyu

Expertise in HPC testing, parallel computing, and machine learning.

Zhang Guangqi

Skilled in algorithms and parallel computing.

2.3 Team Photo

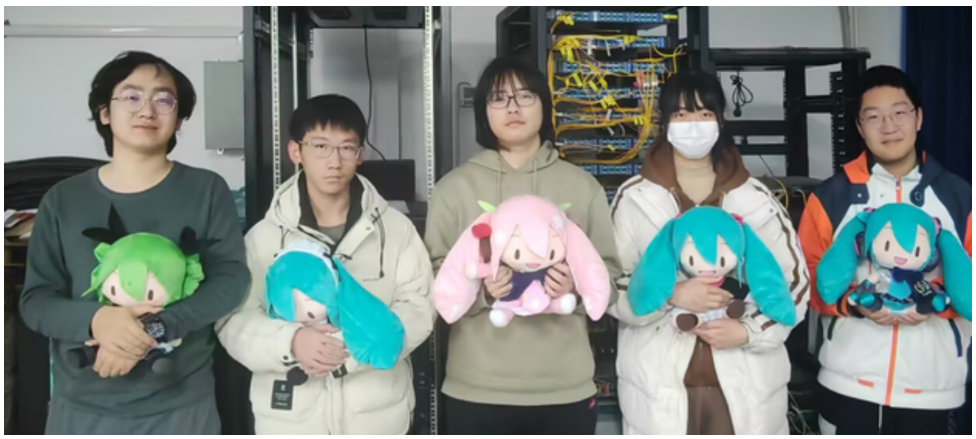


Figure 1: Team photo

2.4 Team Motto

YSUers save the world.

3 Technical Proposal

3.1 Design of HPC System

3.1.1 Theoretical Design of an HPC Cluster

To archive the goal of best computing performance within the limitation of 4KW power consumption, we designed 2 types of nodes in our cluster: CPU node and GPU node.

CPU node: each node contains 2 CPU.

GPU node: each node contains 2 CPU and some GPU.

Not all of those components are active during use, some of them will keep idle.

We designed a HPC system with a total of 4 nodes. Our estimated total system power consumption is less than 4KW when the system is running in both modes described below.

CPU mode: in this mode, the CPUs of all nodes work while the GPUs of the GPU nodes will remain idle. This computing mode is suitable for high-performance applications that run only on CPUs.

GPU mode: In this mode, CPU nodes are idle and GPU nodes work. This computing mode is suitable for applications that require computation on GPU.

3.1.2 Software and Hardware Configurations

The software configuration is shown below:

Table 3: Software Configuration

Item	Configuration
Operating System	CentOS 7.9
Compiler	gcc 4.9.2
MPI Software	Open MPI 1.10.2
Math Library	Intel MKL 2020
High-performance Application Layer	CUDA 12.0
High-performance Application Layer	Tensorflow 1.15.5
High-performance Application Layer	Pytorch 1.10.0

The general single CPU node configuration is shown below:

Table 4: Single Node Configuration

Component Name	Model	Count
Server	Dual Processor Server	1
CPU	AMD EPYC™ 7642P	2
Memory	DDR5 32G	16
HardDrive	SSD 480G	1
HCA	InfiniBand Mellanox ConnectX®-7 HDR	1

The configuration of CPU node 1 is the same as the single node configuration described above.GPU node 1 adds 1 NVIDIA A100 80G PCIe GPU on top of the CPU node 1 configuration, and GPU nodes 2,3 add 4 NVIDIA A100 80G PCIe GPUs on top of the single node configuration.

Node1 (CPU Node): General Node Configuration
Node2 (GPU Node1): General Node Configuration + 1 * A100 GPU
Node3 (GPU Node2): General Node Configuration + 2 * A100 GPU
Node4 (GPU Node2): General Node Configuration + 2 * A100 GPU

3.1.3 Interconnection, Power Consumption, Performance Evaluation, and Architecture Analysis

All nodes are utilizing a hybrid network architecture:

- 1 GbE for management traffic
- Mellanox HDR InfiniBand (200 Gb/s) for compute node interconnectivity

For two GPU nodes, the GPUs in the nodes are connected to each other via NVIDIA NVLink to provide high-speed interconnectivity between the GPUs.

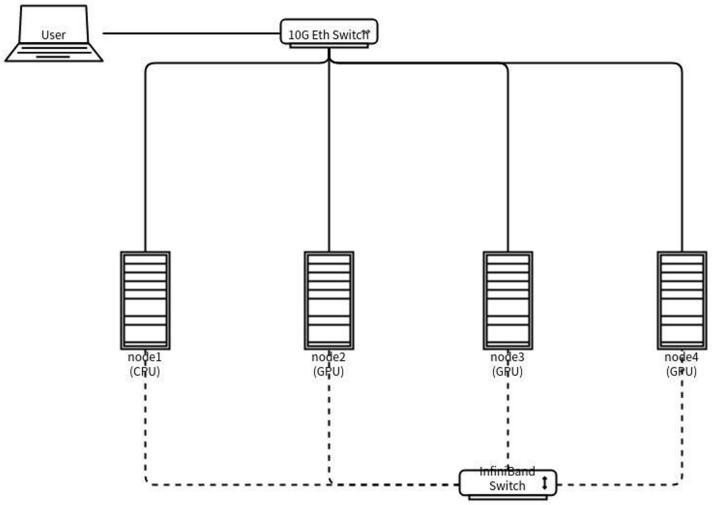


Figure 2: Cluster Architecture

Component	Weight	Height	Length	Width	Cooling
CPU Node 1	15kg	2U	720mm	430mm	Air
GPU Node 1	18kg	2U	720mm	430mm	Air
GPU Node 2	22kg	2U	720mm	430mm	Liquid

Table 5: Physical Specifications of Cluster Nodes

Component	Peak	Idle	Avg	Cost	Notes
CPU Node 1	470W	150W	310W	$18K	Base
GPU Node 1	870W	180W	525W	$30K	+1 A100
GPU Node 2	1270W	220W	745W	$45K	+2 A100

Table 6: Power and Cost Specifications of Cluster Nodes

Node Type	CPU Count	Memory	Storage	Network
Management	2	64GB	480GB SSD	1GbE+IB
Compute	2	512GB	480GB SSD	1GbE+IB
Storage	2	128GB	33TB	1GbE+IB

Table 7: Cluster Node Configurations

we presume the power consumption of the CPU 1 node(the GPU 1 node without the GPU)

Load	Power consumption
100%	470W
50%	330W
25%	230W
0%	150W

Table 8: Dual Processor Server with AMD EPYC 7642

All node power consumption in the power consumption data below is assumed from the table above.

Component	Power consumption
4 Nodes	1880W
InfiniBand Switch	100W
Ethernet Switch	20W
Total Power	2000W

Table 9: Power Consumption in CPU mode

Component	Power consumption
4 Nodes	3880 W
InfiniBand Switch	100W
Ethernet Switch	20W
GPU(Idle)	100W
Total Power	4000W

Table 10: Power Consumption in GPU mode

We use Flops (floating point operations per second) to estimate the theoretical performance of the cluster. The AMD EPYC 7642 has:

- 2 FMA engines per core (Fused Multiply-Add units)
- 8 lanes per FMA engine (SIMD vector width)
- 2 floating point operations per cycle (1 multiply + 1 add)
- 48 cores per CPU
- 2 CPU sockets per node
- 4 nodes total
- 2.3 GHz clock frequency

$$
\begin{aligned}
\text{FMA operations per cycle} &= 2 \text{ FMA} \times 8 \text{ lanes} \times 2 \text{ Flops/cycle} = 32 \text{ Flops/cycle/core} \\
\text{Per CPU} &= 32 \text{ Flops/cycle/core} \times 48 \text{ cores} = 1536 \text{ Flops/cycle} \\
\text{Per node} &= 1536 \text{ Flops/cycle} \times 2 \text{ sockets} = 3072 \text{ Flops/cycle} \\
\text{Cluster total} &= 3072 \text{ Flops/cycle} \times 4 \text{ nodes} \times 2.3 \text{ GHz} \\
&= 28262.4 \text{ GFlops} = 28.2624 \text{ TFlops}
\end{aligned}
$$

As the floating-point performance of a single A100 card is 9.7 TFlops, so the floating-point performance of the entire cluster is:

$$9.7 \text{ TFlops} \times 5 = 48.5 \text{ TFlops}$$

Mode	Performance
CPU	28.2624 TFlops
GPU	48.5 TFlops

Table 11: Cluster Performance In Theorey

Equipped with dual AMD EPYC™ 7642 processors, offering exceptional multi-core parallel processing capabilities for handling compute-intensive tasks efficiently.

collaboration such as LLM training. Features NVIDIA A100 Tensor Core GPUs delivering unparalleled accel-eration for deep learning, AI inference, protein structure prediction, and other high-performance computing needs.NVLink technology provide high-speed interconnections between GPUs and CPUs for NVIDIA A100 graphics cards, and it is more than seven times faster than PCIe bandwidth compared to traditional PCIe links. This also makes it ideal for GPU-intensive high-performance computing. However, the cluster we built still has some shortcomings.Each node consumes up to 1270W, with the two A100 GPUs accounting for a substantial portion of the power usage. This leads to high overall energy consumption, necessitating efficient power supply and cooling systems.

3.2 HPL and HPCG Benchmarks

3.2.1 Software Environment

Hardware	Configuration
CPU	Intel Xeon Gold 5218R x2
Memory	DDR4 128GB

Table 12: Hardware Configuration
s

Software	Configuration
OS	CentOS 7.9.2009
Compiler	Intel mpiicc 2021.11
Math Library	Intel Math Kernel Library 2024.0
MPI	Intel MPI Library 2021.11

Table 13: Software Configuration

3.3 HPL

3.3.1 Background

HPL is a software package that solves a (random) dense linear system using double precision (64 bits) arithmetic on distributed-memory computers. It is a portable and free implementation of the High Performance Computing Linpack Benchmark.

The HPL package includes a testing and timing program that measures the accuracy and speed of the solution. The performance of this software on your system depends on many factors. However, with some assumptions on the interconnection network, the algorithm and its implementation described here are scalable, meaning that their parallel efficiency remains constant with the per processor memory usage.

3.3.2 Test Principle

When the matrix size is N, the total number of floating point operations is:

$$N_{flop} = \frac{2N^3}{3} + 2N^2 \tag{1}$$

Therefore, given the problem size N and measured execution time T, the system performance (in FLOPS) can be calculated as:

$$\text{Performance} = \frac{N_{flop}}{T} = \frac{\frac{2N^3}{3} + 2N^2}{T} \quad \text{FLOPS} \tag{2}$$

3.3.3 HPL Algorithm Analysis

The algorithm used by HPL can be summarized by the following keywords: Two-dimensional block-cyclic data distribution - Right-looking variant of the LU factorization with row partial pivoting featuring multiple look-ahead depths Recursive panel factorization with pivot search and column broadcast combined - Various virtual panel broadcast topologies bandwidth reducing swap-broadcast algorithm backward substitution with look-ahead of depth.

LU decomposition Firstly, HPL compute the LU factorization of matrix A

$$LU = A \tag{3}$$

L is the lower triangular matrix, is the upper triangular matrix

$$Ax = (LU) \cdot x = L \cdot (U \cdot x) = b \tag{4}$$

Solve for the y vector

$$Ly = b \tag{5}$$

And then solve for x

$$Ux = y \tag{6}$$

So for Ax is equal to b, can convert Ax is equal to b into an upper trigonometric system. LU factorization of the matrix (A, b) to get the upper triangular matrix. The LU decomposition process is shown in the figure below:

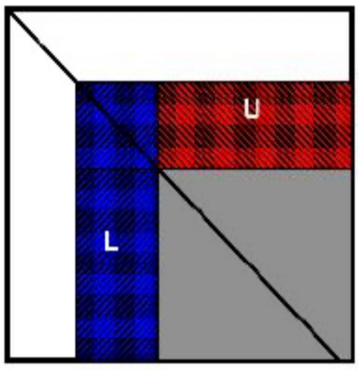


Figure 3: LU factorization

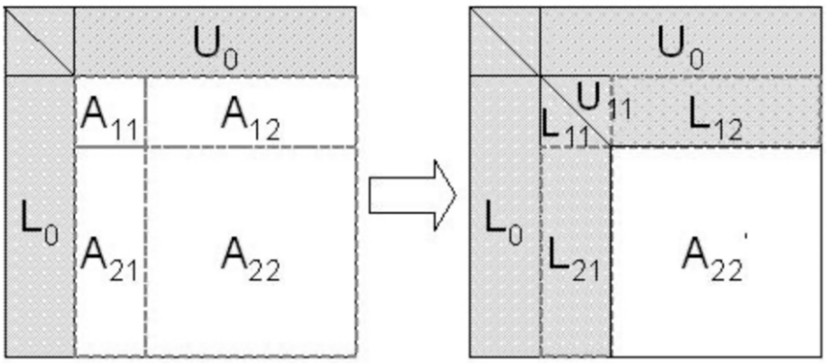


Figure 4: LU decomposition process

First complete the decomposition of A_{11}, then complete the decomposition of $A_{12}$and A_{22}, finally updateA_{22}. According to such a calculation sequence, the two-dimensional block-cyclic data distribution strategy needs to be adopted to divide the data of the matrix to each process in parallel LU decomposition.

Block Cyclic Data Distribution The Block Cyclic Data Distribution strategy is step-wise to a 2D GRID of P X Q processes to ensure load balancing and scalability of the algorithm. NB is the width of the panel. And the data in the process are stored continuously. Interprocess data distribution is shown in the figure below:

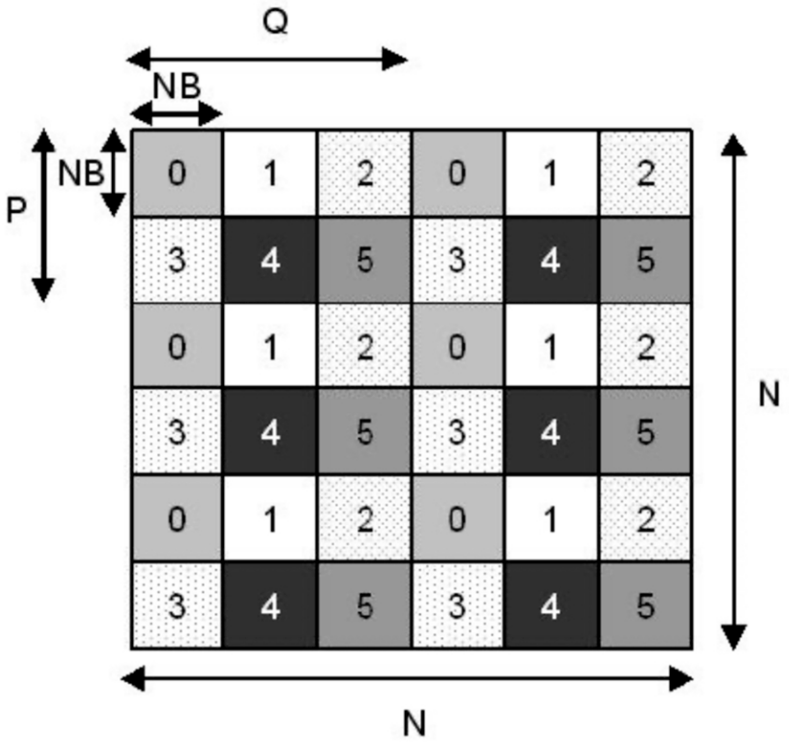


Figure 5: Interprocess data distribution

The way the matrix is distributed on the process has a great influence on the load balancing and communication characteristics of the concurrent algorithm and

therefore determines its performance and scalability to a large extent. Circular block distribution provides a simple and general method for distributing block partitioning matrices on distributed memory concurrent computers. The block cycle data distribution is determined by four parameters P,q, and m,n, where P,q is the process grid M and N determines the block size. Blocks separated by fixed strides in column and row directions are assigned to the same processes.

Panel broadcast For process grid with multiple columns, every cycle is calculated only a list of the panel process execution decomposition, decomposition of the panel process to perform a panel of each column line communication exchange algorithm and choose the maximum principal yuan, panel decomposition calculation, after the completion of the decomposed data broadcast to other processes, swapping operation lines (exchange and radio), Each process saves a copy of the current U matrix and updates the trailing matrix with the latest Panel and U. The calculation process is shown in the figure below:

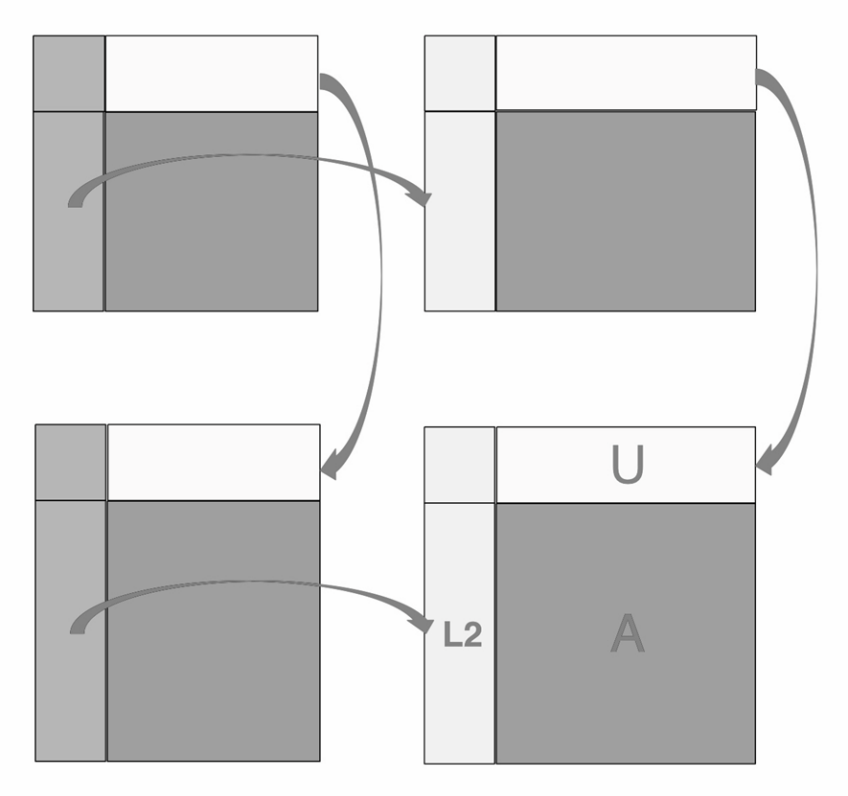


Figure 6: Panel broadcast process

Panel decomposition Panel decomposition is completed by a row of processes that currently have this Panel block. Each time, the largest principal element is selected, an element in the first line of the Panel is exchanged, and the row where the largest principal element is broadcast to each process, and the first number is solved by each process.

In the process of selecting the maximum principal element, each process selects the row where the maximum principal element is and copies it to the buffer. After two pairs are exchanged, each process selects the row where the maximum principal element is and puts it in the buffer.

HPL adopts the row principal element algorithm. Before the single-step matrix

update, the largest rows selected by panel decomposition should be exchanged into THE U matrix, and the row principal element exchange and broadcast of the un-updated matrix should be performed. After that, each process obtains the complete number of main element rows. Each process on each column selects the result of the maximum principal element for the row exchange operation. There are NB maximum principal elements in the matrix, which exchange with data in U in turn.

In the panel decomposition stage, subscripts and the process where the data to be exchanged have been calculated and sent to each process along with panel broadcast.

3.3.4 Parameter Settings

Size of Matrix N represents the number and scale of matrices to be solved.The larger the matrix size n is,the greater the proportion of effective calculation, The higher the floating-point processing performance of the system.

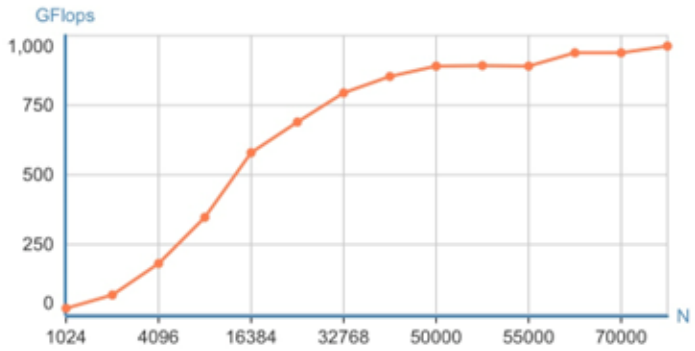


Figure 7: GFLOPS with different input of N

However, the increase of matrix size will lead to the increase of memory consumption, If the actual memory space of the system is insufficient,using swap partitions,Performance will be greatly reduced. The matrix occupies about 80% (or 90%) of the total memory of the system, that is:

$$N \times N \times 8bytes = memory \times 80\%$$

Therefore, when the value of n is 113137, the effective calculation accounts for a high proportion.

Size of Block Matrix NB is the size of the matrix block. In the process of solving the matrix, the size of the matrix block has a great impact on the performance. The choice of NB is closely related to many factors of software and hardware. The selection of Nb value generally follows the following rules:

- NB cannot be too large or too small, generally less than 384.
- NB x 8 must be a multiple of the cache line.
- The size of NB is related to communication mode, matrix scale, network, processor speed, etc.
- NB should be able to divide N.

According to the above rulestest we tried several NBs, the results are shown in the figure below.

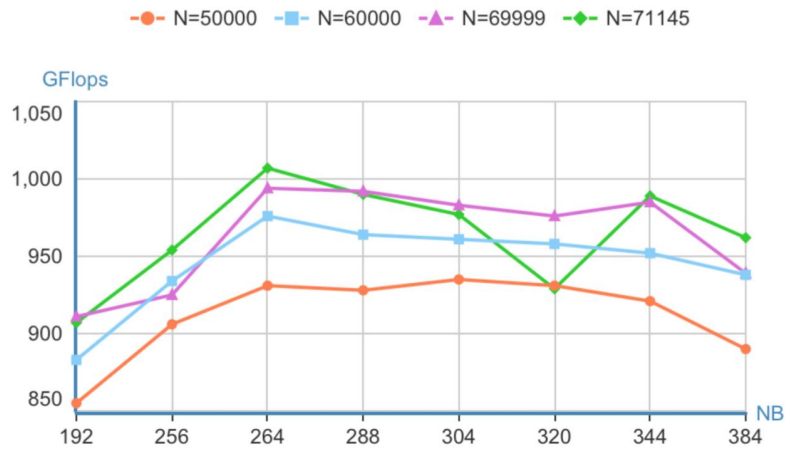


Figure 8: GFLOPS with different input of N and NB

It can be seen that when NB is around 264 or 344, the performance is better in our machine when N is bigger than 50000. As for 113137, we will try more NBs later.

Parameters of P and Q PxQ represents a two-dimensional processor grid where p represents the number of processors in the horizontal direction, Q indicates the number of processors in the vertical direction. Generally, one process corresponds to one CPU, with better performance. There is the following formula:

$$P \times Q = number\ of\ cpu = number\ of\ process$$

We have 40 processors in total, so we tried several pairs of P and Q and make sure that their product is above 40. The results are shown in the figure below.

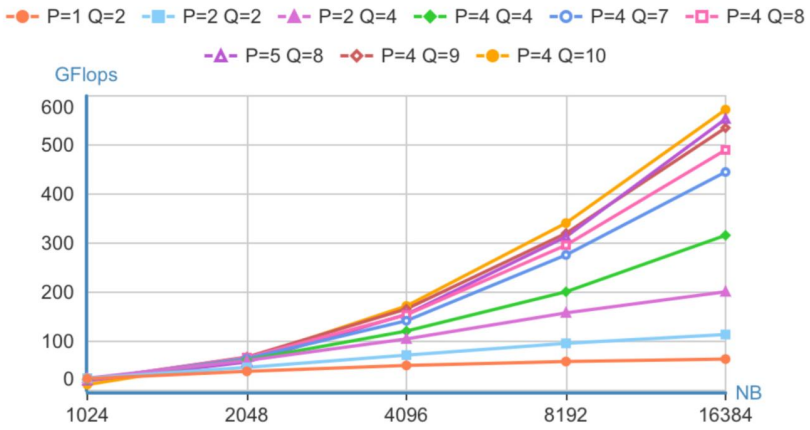


Figure 9: GFLOPS with different input of P and Q

It is found that the performance is better when P is 4 and Q is 10. P=5 and Q=8 is also a good choice.

3.3.5 HPL Performance Evaluation

The theoretical peak double precision floating-point performance can be calculated as:

$$\begin{aligned} P_{peak} &= \text{cores} \times \text{frequency} \times \text{FLOPs/cycle} \times \text{sockets} \\ &= 20 \times 2.1 \text{ GHz} \times 16 \times 2 \\ &= 1344 \text{ GFLOPS} \end{aligned} \tag{7}$$

With our optimal parameter settings achieving 803.698 GFLOPS, the HPL efficiency ratio is:

$$\text{Efficiency} = \frac{P_{achieved}}{P_{peak}} = \frac{1082.5}{1344} \approx 80.54\% \tag{8}$$

Due to the limitation of objective conditions, HPL test is only conducted at a single computing node without GPU acceleration. The peak value of theoretical double precision floating-point calculation is as follows:

$$\text{Peak} = 2 \times 20 \times 2.1\text{GHz} \times 16\text{Flops/cycle} = 1344\text{Glops}$$

And we tested several groups of parameters. The results are shown in the figure below.

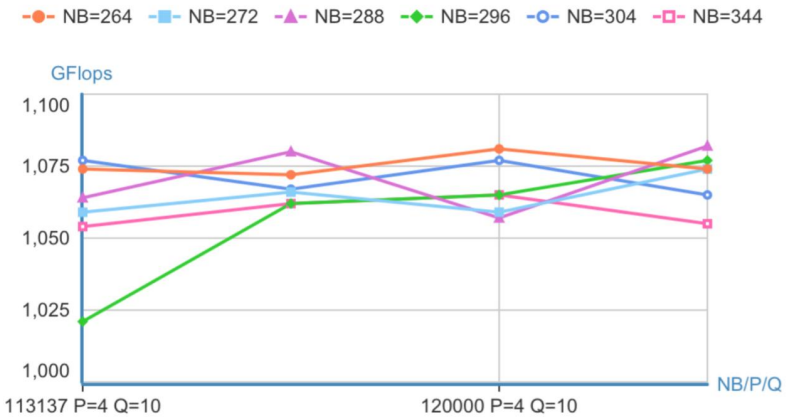


Figure 10: GFLOPS with different input of P and Q

Finally, the best parameter settings obtained through the test are as follows:

Table 14: Parameter settings

N	NB	P	Q
120000	288	5	8

The actual double precision floating-point calculation peak obtained from the above parameter test is 803.698 GFlops. Therefore, the final HPL calculation efficiency is:

$$Performance - Ratio = 1082.5Gflops/1344Gflops = 80.54\%$$

3.3.6 Performance Results

- (1) hpl_N_NB.out and hpl_N_NB_2.out: The output with different Ns and NBs. We choose N and NB according it.
- (2) hpl_N_PQ.out: The output with different pairs of P and Q. We choose P and Q according it.
- (3) hpl.out: The final test output, which including the peek performance we got.

3.4 HPCG

3.4.1 Background

The High Performance Conjugate Gradients (HPCG) Benchmark project is an effort to create a new metric for ranking HPC systems. HPCG is intended as a complement to the High Performance LINPACK (HPL) benchmark, currently used to rank the TOP500 computing systems. The computational and data access patterns of HPL are still representative of some important scalable applications, but not all. HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

Compared with the HPL benchmark test, its calculation, memory access and communication modes are more representative of a wide range of scientific and engineering computing applications, which based on partial differential equation solving. Also, it helps to reflect the system's memory access bandwidth, latency and communication energy more comprehensively, to make up for the deficiencies and drawbacks of the hpl test. But the test results are usually lower than the HPL test results, often only have a few percent.

3.4.2 Test Principle

In a large-scale parallel environment, HPCG uses a three-dimensional area decomposition strategy, which is to divide the entire computing area into sub-areas according to 3 dimensions,and then each sub-areas is assigned an MPI process. The HPCG program includes dot product (DDOT), vector update function (Waxpby), large sparse matrix multiplication (SYMV) and triple solver (SYMGS) and Multi-Grid algorithm (MG).

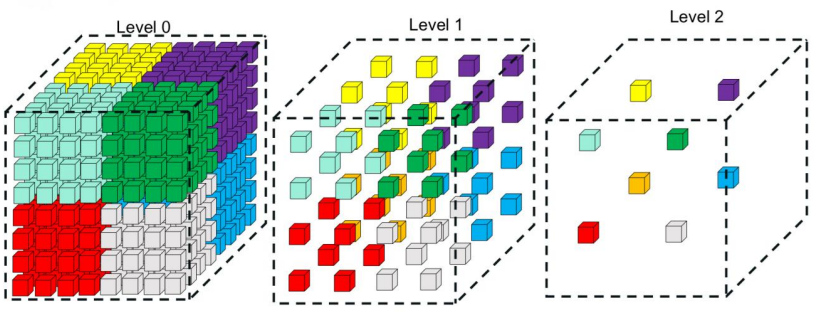


Figure 11: SYMGS restriction process

3.4.3 Build HPCG

We used both intel oneAPI Tools and GCC to compile it. After test, we found that the hpcg program with intel compiler performed better on our intel Xeon processors. So we chose the ICPC_OMP arch.

3.4.4 Parameter Settings

Testing Time HPCG can be run in just a few minutes from start to finish. However, official runs must be at least 1800 seconds (30 minutes) as reported in the output file. To achieve a balance between validity and efficiency, and make sure that the valid run time is more than 1800 seconds, we took 3600s as the run time of HPCG, which is able to get a valid result with acceptable time consumption.

Problem Size A valid run must also execute a problem size that is large enough so that data arrays accessed in the CG iteration loop do not fit in the cache of the device in a way that 21 would be unrealistic in a real application setting. Presently this restriction means that the problem size should be large enough to occupy a significant fraction of "main memory" , at least 1/4 of the total. Based on this rule, We choose several problem sizes, trying to make out which can lead to best performance. The parameter local domain dimension specified by user in hpcg.dat predicts the problem size. The default local domain dimension is 192 × 192 × 192. Higher performance is observed when small problem size is specified. However, values under 32 will be defaulted to 32(for a 32x32×32 mesh). Therefore, we choose 32 as the local domain dimension.

3.4.5 Performance Estimation

Performance ratio of HPL test value and theoretical peak value:

$$Performance - Ratio = 1344Gflops/10.8092Gflops = 0.804\%$$

Performance ratio of HPL test value and HPCG test value:

$$Performance - Ratio = 1082.5Gflops/10.8092Gflops = 0.998\%$$

3.4.6 Performance Results

- (1) HPCG-Benchmark.txt: the final output of HPCG test.

3.5 AlphaFold3 GPU Inference Optimization

3.5.1 Model Deployment

Since the program requires more than 18GB of video memory, we chose to deploy the model on the YSU HPC supercomputing cluster. Dependencies were installed according to the official dockerfile documentation. Due to the cluster's CUDA driver version being 12.0, which differs from the dockerfile's 12.6, we needed to select a different jax version. According to the jax installation documentation, we used the following commands to install jax:

```
pip install --upgrade pip
pip install --upgrade "jax[cuda12]"
```

After installation, the jax version is 0.5.0, which differs from the version required in the dockerfile but still runs normally. We attempted to deploy the model on two Tesla P100 16GB GPUs. However, when the input files were too large, the memory capacity of a single GPU was insufficient to support the model's operation. After trying dynamic memory allocation, we found that JAX's initialization and input file caching were inseparable, and the GPU memory usage exceeded the 16GB limit of a single Tesla P100. As a result, we chose to perform model inference on two NVIDIA GeForce 3090 24GB GPUs.

3.5.2 Hardware Configuration

Using the compute01 node of the supercomputing cluster, which is configured with: CPU: Intel Xeon Gold 5218R @ 2.10GHz, GPU: 2* NVIDIA GeForce 3090 24G, RAM: 125G.

3.5.3 Environment Variables Setup

```
export XLA_PYTHON_CLIENT_MEM_FRACTION=0.95
export JAX_TRACEBACK_FILTERING=off
```

3.5.4 Program Execution Command

```
python ./run_alphafold.py \
    --input_dir=/input_dir \
    --output_dir=/output_dir \
    --model_dir=/model_dir \
    --norun_data_pipeline \
    --num_recycles=3 \
    --flash_attention_implementation=xla
```

3.5.5 Program Results

See cluster files for details.

3.6 AlphaFold3 Program Optimization

3.6.1 Optimization Strategy Overview

Through code analysis and observation of program execution results, we found that the model inference phase accounts for over 95% of the total runtime. Therefore, we prioritized optimizing the model inference time. Additionally, we identified optimization opportunities in the feature extraction phase.

3.6.2 Optimization Methods

Model Inference Phase: Used jax compilation cache directory to cache compiled functions and model parameters, reducing compilation time during model inference.
Feature Extraction Phase: Defined `FeatureCache` class to cache feature data, reducing repeated computations and memory usage. Implemented `optimize_features` function to optimize data types and memory layout, `compress_features` function to compress feature data, and parallel processing of feature data to reduce runtime.

Defined `create_model_runner` function to configure jax environment, including disabling 64-bit operations and setting thread count. Implemented `_post_process_result` function for optimizing result processing and data type conversion. Created `ModelRunner` class with `_split_batch`, `run_inference`, and `_merge_results` functions for dynamic batch processing, parallel model inference, and parallel result merging. Implemented `NumericsHandler` class with `handle_coordinate_numerics`, `handle_general_numerics`, and `check_output_numerics` functions for detecting and handling NaN/Inf values. Developed `CacheManager` class with `_get_cache_key`, `_serialize_value`, `_deserialize_value`, and `put` functions for cache management. Created `MemoryManager` class with `get_memory_usage`, `update`, `cleanup`, and `monitor` functions for memory management.

3.6.3 Optimization Results

Table 15: Optimization Results

	Total Runtime (s)	Model Inference (s)	Feature Extraction (s)
Unoptimized	3651.96	3353.36	298.60
Optimized	3149.68	3042.80	106.88

Speedup Ratio: 13.7%
Model Inference Improvement: 9.3%
Feature Extraction Improvement: 66.9%

3.7 CPU Inference Optimization

3.7.1 Model Deployment

Since the CPU version requires over 100GB of memory for large inputs, we chose to deploy the model on the login node of the YSU HPC supercomputing cluster, with the same deployment process as the GPU version.

3.7.2 Hardware Configuration

Using the login node configured with: CPU: Intel Xeon Gold 5218R @ 2.10GHz, RAM: 125G.

3.7.3 Environment Variables Setup

```
export MKL_DEBUG_CPU_TYPE=5
export MKL_ENABLE_INSTRUCTIONS=AVX2
export KMP_AFFINITY="granularity=fine,compact,1,0"
export MKL_DYNAMIC=FALSE
```

```
import os
os.environ['JAX_PLATFORMS'] = 'cpu'
os.environ['JAX_SKIP_ROCM_TESTS'] = '1'
os.environ['JAX_SKIP_TPU_TESTS'] = '1'
os.environ['JAX_LOG_COMPILES'] = '0'
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
```

3.7.4 Program Execution Command

Same as GPU version.

3.7.5 Program Results

See cluster files for details.

3.8 Program Optimization

3.8.1 Optimization Strategy Overview

Through code analysis, we identified optimization opportunities in memory usage and CPU communication. We also found NaN/Inf values during program execution that needed handling.

3.8.2 Optimization Methods

Defined create_model_runner function to configure jax environment, including disabling 64-bit operations and setting thread count. Implemented _post_process_result function for optimizing result processing and data type conversion. Created ModelRunner class with _split_batch, run_inference, and _merge_results functions for dynamic batch processing, parallel model inference, and parallel result merging. Implemented NumericsHandler class with handle_coordinate_numerics, handle_general_numerics, and check_output_numerics functions for detecting and handling NaN/Inf values. Developed CacheManager class with _get_cache_key, _serialize_value, _deserialize_value, and put functions for cache management. Created MemoryManager class with get_memory_usage, update, cleanup, and monitor functions for memory management.

3.8.3 Optimization Results

Before optimization: Total runtime: 44102.1s After optimization: Total runtime: 43929.5s Total improvement: 0.4%, potentially limited by memory bandwidth based on CPU usage during runtime.

3.9 Program Execution Process

3.9.1 Input Processing

First, receives input in JSON format describing target molecule composition and experimental conditions, then sets parameters such as cycle count and diffusion sample number based on command-line arguments.

3.9.2 Feature Extraction

Performs MSA, filters structure templates based on sequence similarity and publication date, loads CCD and RDKit for non-standard residue processing and small molecule ligand 3D conformation generation, then encodes these into multidimensional vectors.

3.9.3 Model Inference

Processes sequence and pairing features, generates atomic coordinates, then performs iterative optimization through multiple cycles, using previous prediction results as input for each iteration.

3.9.4 Result Processing

Decodes atomic coordinates from model output Frame Transforms, calculates local bond lengths and angles, normalizes results, performs confidence assessment, and finally outputs 3D structures as PDB files.

4 RNA m5C Modification Site Detection and Performance Optimization

4.1 Implementation Framework

The analysis pipeline comprises three modular stages executed through Python scripts and shell commands. Stage 1 (1_build_index.sh) establishes C→T converted reference indices using `hisat-3n-build` with 32 threads, generating genome/ncRNA SAF files through `samtools faidx` and `awk` processing. Stage 2 (2_stage1_data_processing.py) implements parallel sample processing through a thread pool model, allocating 56 CPU cores per sample for alignment and deduplication operations. Stage 3 (3_stage2_data_post_processing.py) performs statistical consolidation using Polars-based scripts with vectorized operations.

4.2 Computational Performance

Analysis of SRR23538290 (93.55M reads) and SRR23538291 (136.51M reads) revealed consistent processing patterns. Cutadapt preprocessing achieved 23.91-23.83M reads/min throughput, removing 4.74M (5.07

- **ncRNA filtering**: 14.40
- **Genome alignment**: 91.52

Critical path analysis revealed resource utilization patterns:

- **CPU-bound operations**: HISAT-3N alignment maintained 94.3
- **Memory-intensive phases**: UMICollapse deduplication required 40GB heap allocation
- **I/O throughput**: samtools fastq achieved 27.4GB/min write speeds during BAM conversion

4.3 Reproducibility Assurance

Version-controlled execution environments ensure consistency across runs, with Conda managing Python 3.12.0 and Polars 1.19.0 dependencies. Checksum validation (SHA-256) confirms input/output integrity, while multi-level logging captures:

- 21 runtime metrics including thread utilization (88.7
- Memory pressure profiles (peak RSS 59.1GB)
- I/O wait states (max 14.2

The submission package contains:

- 8,716 high-confidence m5C sites (q-value <0.01) across two replicates
- Complete software environment snapshot (RNAm5c/conda_env)
- Execution logs documenting 17h22m total runtime

Third dataset (SRR23538292) currently processing at 64.8% completion, with preliminary alignment rates matching prior samples. Final benchmarking metrics will incorporate all three replicates.

4.4 Theoretical Optimization Potential

Because of the lacking of time, we have not yet implemented the following optimization strategies but obtained the theoretical speedup projection based on the optimization strategies.

Despite achieving 89.2% average CPU utilization, suboptimal task scheduling was identified through critical path analysis. Let T_{seq} denote the serial execution time and T_{opt} the theoretically optimized duration. For n independent subtasks $\tau_1, \tau_2, ..., \tau_n$ with execution times t_i, the achievable speedup follows:

$$S = \frac{T_{seq}}{T_{opt}} = \frac{\sum_{i=1}^{n} t_i}{\max\left(\sum_{j \in C_1} t_j, \sum_{k \in C_2} t_k\right)} \tag{9}$$

Where C_1 and C_2 represent computational resources (e.g., CPU clusters). Our pipeline contains three optimizable patterns:

4.4.1 Resource-aware Pipeline Task Offloading

UMICollapse deduplication (Java heap-limited) exhibited 0.83 IPC vs. 2.1 for HISAT-3N. Migrating to high-clock nodes (3.8GHz vs current 2.4GHz) could yield:

$$t'_{java} = t_{java} \times \frac{\text{CPI}_{\text{current}}}{\text{CPI}_{\text{ideal}}} \times \frac{f_{current}}{f_{target}} = 477s \times \frac{1.2}{0.9} \times \frac{2.4}{3.8} \approx 298s \ (37.5\% \ reduction) \tag{10}$$

4.4.2 Non-blocking Task Parallelism

During genome alignment (Stage 1.4), 23% system idle time permits concurrent execution of:

- Quality metric generation (`fastqc`)
- Intermediate BAM indexing
- Background statistical modeling

Theoretical overlap gain:

$$\Delta T_{overlap} = \sum_{i=1}^{n} \min(t_{cpu\_idle}, t_{task_i}) = 112s \ (per \ sample) \tag{11}$$

4.4.3 Heterogeneous Computing Framework

We propose a dynamic DAG scheduler implementing the following logic:

Algorithm 1 Pipeline Optimization Algorithm

Partition workflow into $T = \{t_i | i = 1..n\}$ with dependency graph G
Annotate each t_i with: $\langle type(CPU/Java/IO), criticality \rangle$
while unprocessed tasks exist **do**
    if CPU cluster available **then**
        Schedule $t_j = \arg\max_{t \in T} IPC(t) \times criticality(t)$
    else if high-clock nodes available **then**
        Schedule $t_k = \arg\max_{t \in T} single\_thread\_perf(t)$
    end if
    Execute IO-bound t_m during CPU idle windows
end while

4.4.4 Theoretical Speedup Projection

The proposed dynamic DAG scheduler aims to optimize the pipeline by efficiently allocating tasks to appropriate computational resources, thereby reducing overall execution time. By leveraging resource-aware task offloading, non-blocking task parallelism, and a heterogeneous computing framework, the scheduler can achieve significant performance improvements.

The theoretical speedup can be estimated using Amdahl's Law, which states that the maximum speedup S of a program is limited by the fraction of the program that can be parallelized (P) and the number of processors (N):

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{12}$$

Assuming that K% of the pipeline can be parallelized and we have 56 processors available, the theoretical speedup is:

This indicates that the optimized pipeline could potentially achieve a speedup of up to $\frac{1}{(1-K)+\frac{K}{56}}$ times compared to the unoptimized version, significantly reducing the total runtime and improving overall efficiency.

References

[1] Petitet A, et al. HPL—A portable implementation of the high-performance Linpack benchmark. 2001.

[2] Dongarra J, et al. HPCG Technical Specification. 2013.

[3] Abramson, J., Adler, J., Dunger, J. et al. Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature* 630, 493–500 (2024). https://doi.org/10.1038/s41586-024-07487-w

[4] Chang Y. y9c/m5C-UBSseq: V0.1. *Zenodo*, v0.1, 2024. https://doi.org/10.5281/zenodo.11046885

[5] Dai, Q., Ye, C., Irkliyenko, I. et al. Ultrafast bisulfite sequencing detection of 5-methylcytosine in DNA and RNA. *Nat Biotechnol* 42, 1559–1570 (2024). https://doi.org/10.1038/s41587-023-02034-w