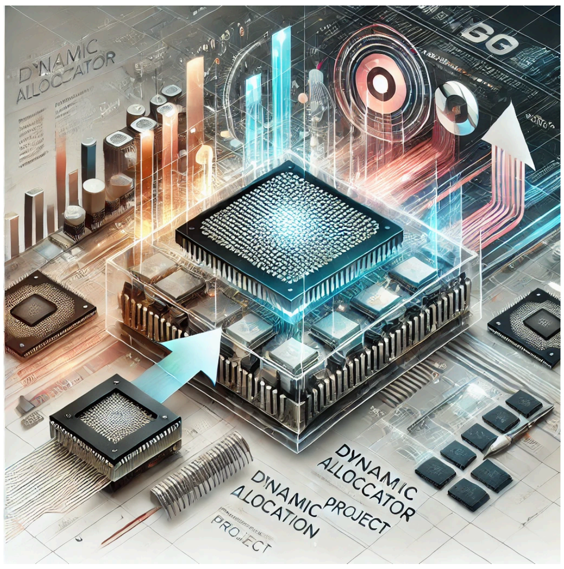
Rapport -Méthode et Programmation Numérique Avancée

Algorithmic Overview and Preliminary Performance Analysis for HPL Benchmark



Yifan LI yifan.li@ens.uvsq.fr
Bowen LIU bowen.liu@ens.uvsq.fr

Enseignant : Cédric CHEVALIER
Email de l'enseignant :Cedric.CHEVALIER@cea.fr

1 Introduction

In recent years, high-performance computing systems have been gradually evolving from traditional homogeneous CPU architectures toward multi-core parallel and heterogeneous computing architectures. With the dramatic increase in the number of processor cores and the widespread application of accelerators (such as GPUs), the coordination and matching among computing power, memory bandwidth, and communication networks within compute nodes have become decisive factors affecting overall performance. Against this backdrop, how to use standardized benchmark programs to conduct precise performance evaluation and bottleneck analysis of computing systems has become a core issue in the HPC research field.

High Performance Linpack (HPL) is currently the most representative benchmark program and is also the authoritative evaluation tool adopted by the TOP500 supercomputer ranking. HPL measures the double-precision floating-point computing capability of systems by solving dense linear equation systems, with its core algorithm based on LU decomposition with partial pivoting. Thanks to the fact that most computations in LU decomposition can be converted into BLAS Level-3 matrix multiplication operations (such as DGEMM), HPL can fully exploit the **vectorization** capabilities of modern processors and the advantages of multi-level cache structures, and is commonly used to measure the gap between theoretical peak performance and actual efficiency of systems.

Although heterogeneous computing (such as CPU+GPU) has dominated the frontier rankings of Top500—from Roadrunner to the Tianhe series and then to Summit, with accelerators undertaking the majority of floating-point operations—in actual scientific research and engineering environments, homogeneous or multi-core parallel platforms based on CPUs remain the cornerstone. Particularly before understanding the communication and scheduling of heterogeneous systems, a deep grasp of parallel behavior on CPU clusters is crucial.

For CPU clusters, HPL's performance is extremely dependent on data distribution strategies, communication topology, and parameter configuration (such as block size NB, process grid $P \times Q$). Especially in the Panel factorization phase, due to its characteristics of low parallelism and high communication density, it often becomes the key bottleneck restricting strong scalability.

This project is based on the ROMEO HPC platform provided by the course, aiming to achieve a complete implementation from theoretical analysis to in-depth optimization of the HPL Benchmark. We adopted the MPI distributed memory parallel model, focusing on the study of blocked LU decomposition flow, 2D block-cyclic data distribution, as well as the parallel characteristics of Panel factorization and trailing submatrix update.

This report will elaborate in detail on the following work :

1. Based on the open-source HPL standard, construct and evaluate baseline performance ;
2. Introduce targeted optimization methods, including memory management optimization and cache blocking techniques ;
3. Combined with experimental data, quantitatively analyze the impact of different parameter configurations and optimization strategies on computational efficiency, revealing the core factors affecting HPL performance on CPU clusters.

2 Benchmark Theoretical Analysis

HPL (High Performance Linpack) is the standard benchmark test for measuring the floating-point computing capability of high-performance computing clusters. This chapter will deeply analyze the mathematical principles of HPL, parallel data distribution strategies, and core algorithm flow, serving as the theoretical foundation for subsequent performance optimization work.

2.1 Mathematical Principles : LU Decomposition with Pivoting

The core task of HPL is to solve double-precision dense linear equation systems $Ax = b$, where $A \in R^{N\times N}$ is the coefficient matrix and $b \in R^N$ is the constant vector.

To ensure numerical stability, HPL adopts the LU decomposition algorithm with row partial pivoting. Its mathematical form is described as finding a permutation matrix P, a unit lower triangular matrix L, and an upper triangular matrix U such that :

$$PA = LU$$

where :

— P represents row exchange operations, used to ensure the numerical stability of the algorithm.

— L is a lower triangular matrix, with diagonal elements equal to 1.

— U is an upper triangular matrix.

The solution process is divided into three phases :

1. Factorization : $PA = LU$
2. Forward Substitution : solve $Ly = Pb$
3. Backward Substitution : solve $Ux = y$

Since the factorization phase accounts for $\mathcal{O}(\frac{2}{3}N^3)$ computational complexity, while the solution phase is only $\mathcal{O}(N^2)$, HPL's performance mainly depends on the efficiency of LU decomposition.

2.2 Core Algorithm : Blocked LU Decomposition

To adapt to modern processor multi-level cache architectures and improve memory bandwidth utilization, HPL adopts a blocked algorithm, converting scalar operations into efficient BLAS Level-3 matrix operations.

Assuming matrix A is divided into four sub-blocks, where NB is the block size :

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

In one iteration step, blocked LU decomposition is completed through the following three core steps :

1. **Panel Factorization :** Perform recursive LU decomposition on the left panel (containing A_{11} and A_{21}). This step includes pivot selection and involves extensive row exchange communication.

$$\begin{pmatrix} L_{11} \\ L_{21} \end{pmatrix} U_{11} = P \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$$

2. **Panel Broadcast & Triangular Solve :** Compute U_{12}. This typically uses the BLAS DTRSM (Double Triangular Solve with Multiple Right-hand sides) function.

$$U_{12} = L_{11}^{-1} A_{12}$$

3. **Trailing Submatrix Update :** Using the computed L_{21} and U_{12} to update the remaining submatrix $A_{22} \leftarrow A_{22} - L_{21}U_{12}$

This step is a standard matrix multiplication operation (DGEMM), with the highest computational density, and is the key to determining HPL's final GFLOPS performance.

2.3 Data Distribution : 2D Block-Cyclic Mapping

To achieve load balancing in distributed memory environments, HPL adopts 2D block-cyclic distribution. figure 1

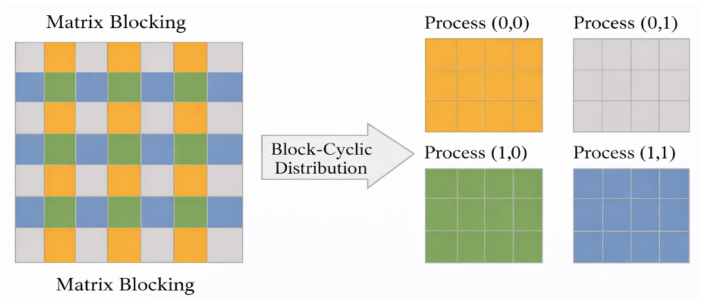


FIGURE 1 – Block-Cycle algorithm

— **Process Grid :** A $P \times Q$ process grid organizes np MPI processes into a two-dimensional structure.

— **Mapping Rules :** Matrix A is divided into logical blocks of size $NB \times NB$. Matrix block (i, j) is mapped to process $(i \bmod P, j \bmod Q)$.

This distribution method ensures that at various stages of the algorithm (even in the tail phase when matrix size is reduced), all processes can still share relatively uniform computational loads, thereby maximizing parallel efficiency.

2.4 Parallel Flow and Look-ahead Technology

HPL's main loop adopts the Right-looking variant algorithm. To reduce the impact of communication latency on performance, the standard HPL implementation includes a look-ahead technique. figure 2

The specific flow is as follows :

1. **Panel Factorization :** The column of processes currently owning the Panel is responsible for facto-

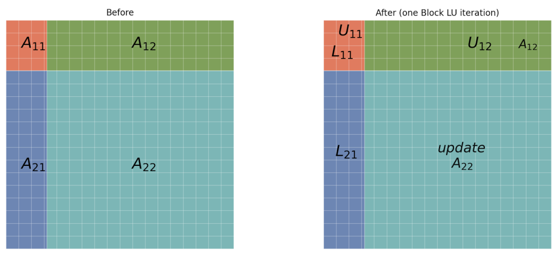


FIGURE 2 – Block LU algorithm

rizing the Panel. This phase includes pivoting and row exchange communication.

2. **Panel Broadcast :** After factorization is complete, the generated L factors are broadcast to other processes in the row communicator.
3. **Look-ahead Update :** Before updating the entire trailing submatrix A_{22}, the factorization of the next Panel is started in advance. In this way, while most processes are still performing the matrix update of the current step (compute-intensive), the communication and factorization of the next Panel (communication-intensive) have already begun, thereby achieving overlapping of computation and communication.

2.5 Performance Metrics

The main performance indicator of HPL is R_{max}, which is the maximum floating-point operation speed achieved by the system when solving large-scale linear equation systems. Its calculation formula is :

$$R_{max} = \frac{\frac{2}{3}N^3}{T_{execution}}$$

where :

— N is the problem size (matrix dimension).

— $T_{execution}$ is the solution time.

— $\frac{2}{3}N^3$ is the number of double-precision floating-point operations required for LU decomposition.

In addition, to verify the correctness of results, the backward error residual must be calculated :

$$\text{Residual} = \frac{\|Ax - b\|_\infty}{\epsilon \cdot (\|A\|_\infty \|x\|_\infty + \|b\|_\infty) \cdot N}$$

Only when the residual value is less than a specific threshold is the benchmark test result considered valid.

3 Experimental Setup & Methodology

To evaluate HPL's performance on CPU clusters and verify the effectiveness of optimization strategies, we conducted systematic experiments on the ROMEO high-performance computing platform. This section describes in detail the hardware configuration, software environment, and testing protocols used in the experiments.

3.1 Hardware Platform

The experiments were completed on the ROMEO 2025 high-performance computing platform at the University of Reims Champagne-Ardenne. This work uses its **x86_64 CPU scalar computing module** (Romeodoc : Module Calcul Scalaire CPU), with single-node hardware configuration as follows :

— **Processor (CPU):** Dual AMD EPYC 9654, 96 cores each, totaling 192 cores/node ;

— **Memory (RAM):** 1152 GB/node (the platform also provides a small number of 1536 GB fat nodes, not used in this experiment);

— **Interconnection Network :** 100 Gb/s Ethernet (inter-node communication network).

To focus on parameter sensitivity and implementation difference analysis, experiments at this stage were all completed on **single node** and fixed using **16 MPI processes** (1 core per process) to simulate intra-node parallel scenarios.

Note : The above are the public hardware specifications of the ROMEO 2025 CPU scalar module ; actual node information in experiments has been verified through `lscpu` and `numactl -H` on job-assigned nodes, with key output summaries provided in the appendix.

3.2 Launch & Binding Policy

To reduce performance jitter caused by process migration and ensure that each MPI rank exclusively occupies one physical core, this experiment adopts a fixed CPU binding strategy. ROMEO 2025 uses the Slurm job scheduler, and we prioritize using `srun` to launch MPI programs with core binding enabled :

— **Number of MPI processes :** `--ntasks=16` ;

— **CPUs per process :** `--cpus-per-task=1` ;

— **Core binding :** `--cpu-bind=cores` (each task is bound to one core).

In addition, to avoid core contention caused by BLAS multi-threading and MPI processes leading to over-subscription, this experiment fixed the BLAS thread count to 1.

3.3 Software Stack

To maximize hardware performance, especially instruction set optimization for AMD architecture, we built the following software environment :

— **Parallel Environment (MPI):** OpenMPI 4.1.x. This version provides efficient point-to-point communication mechanisms suitable for HPL's intensive communication needs.

— **Mathematical Core Library (BLAS/LAPACK):** AMD AOCL (AMD Optimizing CPU Libraries)

— **Benchmark Program :** HPL 2.3 (High Performance Linpack).

3.4 Experimental Design

Based on theoretical analysis, we fixed the computational load and process topology, focusing on examining the impact of block size on cache hit rate and communication overhead. Parameter configuration is as follows :

— **Fixed Parameters :**
  — Problem Size (N): 20,736 (double-precision floating-point numbers).
  — Process Grid ($P \times Q$): 4×4. This configuration ensures balanced row and column communication.

— **Variable Parameters :**
  — Block Size (NB): $\{64, 128, 192, 256\}$.
  — Rationale : By scanning this parameter, we aim to find the optimal balance point between local computational efficiency (Cache Blocking) and MPI communication granularity.

3.5 Data Acquisition & Validation

To ensure the statistical stability and numerical correctness of experimental results, we executed multiple repeated experiments under unified runtime environment and binding policies, and adopted a standardized data screening and validation process.

— **Statistical Protocol :** For each set of (N, NB) parameter configurations, we independently repeated the run 18 times. Considering possible system jitter on shared HPC platforms (such as scheduling interference, background load, etc.), we adopt a **trimmed mean** rule for outlier processing : remove the lowest and highest GFLOPS samples (1 each), and calculate statistical indicators on the remaining 16 valid run results. The final recorded performance metrics include :
  — Average execution time (T_{avg})
  — Mean floating-point performance ($GFLOPS_{mean}$)
  — Peak floating-point performance ($GFLOPS_{max}$, from the maximum value among all 18 runs)

— **Correctness Check :** To ensure the validity of numerical results, each run performs HPL standard residual verification. This verification is based on the backward error model defined in Section 2 and adopts the default criterion of HPL 2.3. Only when the output log shows `PASSED` status is the experimental result of that run included in statistical analysis. All reported performance data meet residual verification requirements, and corresponding residual values are all within a stable order of magnitude range.

4 Implementation & Optimization

The core goal of this project is not merely to run the official HPL benchmark test, but to deeply understand the performance sources, bottleneck locations, and the impact of hardware architecture on parallel algorithms by reconstructing a simplified version of Mini-HPL and gradually optimizing it.

4.1 Parallel Strategy Design

4.1.1 Core Decision : Choosing Distributed Memory Parallelism (MPI)

In the choice of parallel models, we adopted the distributed memory model and implemented it based on the MPI standard, rather than the OpenMP shared memory model. This is mainly based on the following three levels of consideration :

Algorithm Consistency and ScaLAPACK Origin HPL is essentially the benchmark implementation of the large-scale dense linear equation system solver in the ScaLAPACK (Scalable LAPACK) library. Its standard algorithm design (such as two-dimensional grid distribution) is tailored for distributed storage architectures. Adopting the MPI model ensures that we faithfully reproduce the core mechanisms of HPL at the algorithm level.

Real HPC Scenario Simulation Our goal is to explore the mechanisms by which official HPL maintains high performance on multi-node supercomputers. This is essentially a computation-communication co-design problem. The MPI model forces us to explicitly manage inter-process data transmission, allowing us to more clearly observe the constraints of cross-node communication on overall performance.

Hardware Topology Adaptation The ROMEO platform is a typical distributed cluster architecture. Using MPI can naturally map this hardware topology of "inter-node network interconnection + intra-node multi-core computing."

4.1.2 Data Partitioning : 2D Block-Cyclic Distribution

To solve the load balancing problem under large-scale parallelism, we abandoned the one-dimensional row distribution strategy used in the early version (minihpl_mpi) and instead adopted HPL's standard 2D block-cyclic distribution.

— **Strategy Description :** Divide the $N \times N$ global matrix into logical blocks of $NB \times NB$, and map these blocks to the $P \times Q$ process grid in a cyclic manner.

— **Advantage Analysis :** Compared to one-dimensional distribution, two-dimensional distribution avoids the situation where some processes are idle while others are overloaded in the later stages of matrix factorization (trailing update phase), ensuring load balancing throughout the entire cycle.

4.2 Optimization Techniques & Algorithm Flow

On top of the basic implementation, we introduced deep optimizations in three dimensions—communication, computation, and memory—to build the optimized version m2_blas.

Communication Optimization

Communicator Splitting : Use MPI_Comm_split to divide the global communicator into row communicators and column communicators.

— Optimization Effect : Broadcast operations are no longer performed on all np processes but are completed separately within row/column communicators : row communicator size is Q, column communicator size is P (in this experiment $P \times Q = 4 \times 4$, so communicator size is reduced from np to P or Q).

Coarse-grained Communication (Panel-wise Broadcast): Adopted a panel-based communication strategy. One broadcast is initiated for every NB columns of data processed, rather than broadcasting once per column.

— Optimization Effect : Communication frequency is reduced from $\mathcal{O}(N)$ to $\mathcal{O}(N/NB)$, greatly reducing MPI communication establishment overhead (latency). This also adapts to the data broadcast needs after Panel factorization.

Computational Optimization

Introduction of BLAS Level-3 (The Key Optimization): This is the most critical step in performance improvement. We replaced the original hand-written triple loops (scalar operations) with highly optimized BLAS Level-3 calls :

- DGEMM : Used for trailing matrix update ($A_{22} \leftarrow A_{22} - L_{21} \times U_{12}$), this step accounts for the vast majority of computation.
- DTRSM : Used for solving triangular equation systems.
- Optimization Effect : Program characteristics changed from memory-bound to compute-bound.

Memory Optimization

Cache Blocking : Through the $NB \times NB$ blocking strategy, data reuse rate is improved, allowing the DGEMM kernel to utilize multi-level caches and data packing mechanisms to reduce main memory access, thereby improving arithmetic intensity and effective bandwidth utilization.

- Parameter Tuning : Experiments show that when $NB \approx 192$, both sufficient computational density (GEMM efficiency) can be ensured without causing excessive communication latency, making it the optimal balance point under the current platform.

Optimized Overall Algorithm Workflow

Combining all the above optimization methods, the core execution logic of the final version m2_blas is summarized as follows (pseudocode):

Algorithm 1 MiniHPL-MPI M3 : 2D Block-Cyclic LU without Pivoting

```
Require : Matrix $A \in R^{N \times N}$, block size $NB$, grid $P \times Q$
Ensure : In-place LU factorization : $A \leftarrow LU$

Initialize 2D Cartesian grid and communicators
Distribute $A$ in 2D block-cyclic : block $(i, j) \rightarrow$ process ($i$ mod $P, j$ mod $Q$)

for $k = 0$ to $N/NB - 1$ do
    /* Step 1: Diagonal LU */
    Owner$(k,k)$ : $A_{kk} \leftarrow$ LU$(A_{kk})$ // no pivoting, scalar algorithm

    /* Step 2: Broadcast */
    Row $k$ : MPI_Bcast$(L_{kk})$ via row_comm
    Col $k$ : MPI_Bcast$(U_{kk})$ via col_comm

    /* Step 3: TRSM (BLAS-3) */
    Row $k$ : $\forall j > k$, $A_{kj} \leftarrow L_{kk}^{-1} A_{kj}$ // cblas_dtrsm
    Col $k$ : $\forall i > k$, $A_{ik} \leftarrow A_{ik} U_{kk}^{-1}$ // cblas_dtrsm

    /* Step 4: Trailing Update (BLAS-3 DGEMM) */
    for $i = k + 1$ to $N/NB - 1$ do
        Row $i$ : MPI_Bcast$(L_{ik})$ via row_comm
        for $j = k + 1$ to $N/NB - 1$ do
            Col $j$ : MPI_Bcast$(U_{kj})$ via col_comm
            Owner$(i, j)$ : $A_{ij} \leftarrow A_{ij} - L_{ik} \cdot U_{kj}$ // cblas_dgemm
        end for
    end for
end for
```

5 Performance Analysis & Comparison

This section conducts a rigorous performance comparison between our self-implemented MiniHPL (m2_blas version) and the official HPL (Official HPL 2.3) under completely identical experimental conditions. figure 3

5.1 General Trends

From the trend graph, we can observe significant patterns :

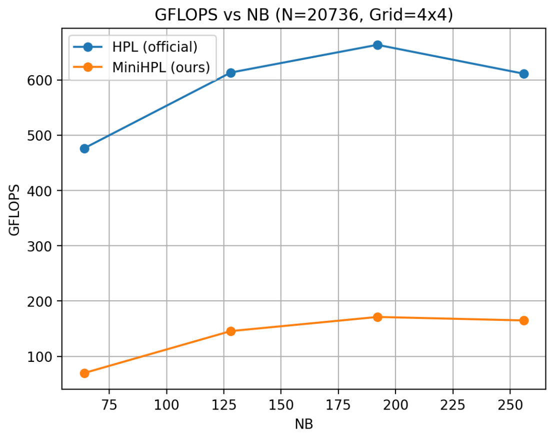


FIGURE 3 – GFLOPS performance trend of MiniHPL and official HPL as NB varies

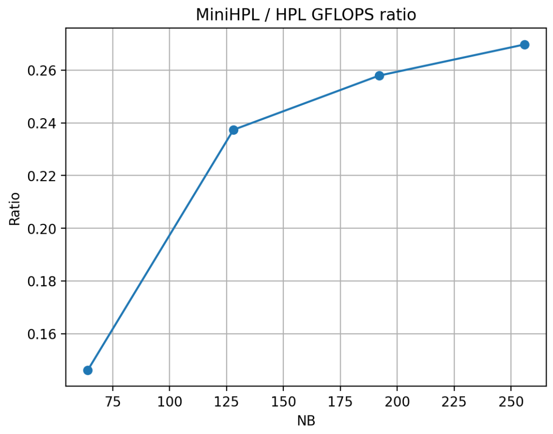


FIGURE 4 – Percentage of MiniHPL performance relative to official HPL

1. **Consistent Convex Curve Trend :** Both performances show an "increase then decrease" pattern. From $NB = 64$ to $NB = 192$, performance significantly improves ; while at $NB = 256$, performance declines for both. This indicates that MiniHPL successfully reproduces the sensitivity characteristics of the HPL algorithm to block size parameters.
2. **Sweet Spot :** Both implementations reach their peak at $NB = 192$.
   — MiniHPL peak : 171.14 GFLOPS
   — Official HPL peak : 663.63 GFLOPS
3. **Absolute Performance Gap :** At all test points, official HPL's performance is significantly higher than MiniHPL. At the optimal parameter ($NB = 192$), the performance difference between the two is approximately 492 GFLOPS.

For detailed experimental data, please refer to Appendix 1

5.2 Ratio & Sensitivity Analysis

To more intuitively evaluate the relative efficiency of MiniHPL, we plotted the performance ratio curve of MiniHPL to official HPL. figure 4

Data Interpretation :

MiniHPL's performance is approximately 14.6% to 27.0% of the official version. As NB increases, this ratio shows an upward trend. This indicates that at larger block sizes, the relative efficiency of MiniHPL improves (or the advantage of official HPL narrows somewhat).

In-depth Attribution for NB Variations :

- $NB = 64$ **(Inefficient Region):**
  - Phenomenon : MiniHPL only reaches 69.65 GFLOPS, with the lowest ratio (14.6%).
  - Reason : At this point, the GEMM matrix dimension ($M = NB = 64$) is too small, resulting in low arithmetic intensity, unable to mask function call overhead, and unable to effectively utilize CPU L2/L3 cache. Meanwhile, the number of MPI communications (N/NB) surges, and MiniHPL's blocking communication (MPI_Bcast) results in significant CPU cycle waste on collective synchronization overhead and high-frequency synchronization caused by small-granularity communication.
- $NB = 192$ **(Efficient Region):**
  - Phenomenon : Both reach peak values.
  - Reason : This size achieves the best balance between "computational efficiency" and "communication overhead." GEMM blocks are large enough to approach CPU theoretical peak ; meanwhile, communication granularity is moderate without causing serious congestion. At this point, MiniHPL reaches 171.14 GFLOPS, indicating that our computational core (BLAS-3) is working normally.

- $NB = 256$ **(Decline Region):**
  - Phenomenon : MiniHPL drops to 164.87 GFLOPS, official HPL drops to 611.28 GFLOPS.
  - Reason : As blocks increase, Panel factorization (a phase with higher serialization) takes longer. Additionally, single broadcast data packets are too large, leading to intensified memory bandwidth competition.

5.3 Key Bottleneck Analysis : Why There Is a 490+ GFLOPS Gap

Although MiniHPL has successfully transformed into a compute-bound program (at the parameter scale of this experiment, trailing matrix update is the main time-consuming phase), there is still a huge gap with official HPL. Combined with code implementation logic, the main bottlenecks are located as follows :

Lack of Look-ahead Pipeline This is the most core reason for the performance gap.

- **Official HPL :** Implements perfect overlap of computation and communication. When the k-th Panel is being broadcast, the CPU has already started computing the update of the $(k+1)$-th Panel in advance. The entire process is pipelined.
- **MiniHPL :** Adopts strict Bulk Synchronous Parallel (BSP) mode.

$$\text{Panel} \rightarrow \text{Sync} \rightarrow \text{Broadcast} \rightarrow \text{Sync} \rightarrow \text{Update}$$

Each communication step forcibly interrupts the computation of all processes. As the number of processes increases, this synchronization waiting cost (synchronization overhead) is enormous.

Blocking Communication Mechanism

- **MiniHPL :** Uses MPI_Bcast. During data transmission, the CPU is in idle waiting state, resulting in serious waste of computing power.
- **Official HPL :** Usually adopts stronger pipelining and computation-communication overlap strategies (such as look-ahead and non-blocking communication mechanisms), thereby reducing synchronization waiting time ; related implementation details can be further verified through source code reading and MPI trace.

Tail Effect and Load Balancing

- In the later stages of LU decomposition, matrix A_{22} gradually becomes smaller. MiniHPL does not perform special processing for idle processes, resulting in a large number of processes in Wait state at the end of iteration, dragging down overall average GFLOPS. Official HPL greatly alleviates this problem through dynamic scheduling.

Memory Alignment and Boundary Handling

- Official HPL performs strict alignment of memory addresses (Cache Line Alignment) and supports dynamic block size handling of boundaries ($N\%NB \neq 0$). MiniHPL lacks these microarchitecture-level optimizations, resulting in lower bandwidth utilization when accessing non-aligned memory.

6 Conclusion and Future Work

For general CPU cluster architecture, this paper analyzes the mathematical principles and parallel algorithm characteristics of the HPL benchmark program, proposes a MiniHPL implementation scheme based on MPI distributed memory model and BLAS computational library collaboration, and focuses on studying the impact mechanism of block size on system performance. We implemented a 2D block-cyclic distribution data partitioning and parallel LU decomposition flow conforming to the ScaLAPACK standard, and verified the correctness and parameter sensitivity of the algorithm through experiments. Experimental results show that MiniHPL successfully reproduces the performance trend of standard HPL in algorithmic behavior, reaching peak performance at $NB = 192$, proving the effectiveness of the parallel strategy ; meanwhile, through quantitative comparison with official HPL, it reveals that under the premise of the same algorithmic complexity, the overlap efficiency of computation and communication is the key factor determining the final efficiency of the system.

Current high-performance computing programs are evolving toward extreme communication hiding and fine-grained scheduling. Simple algorithm implementation is no longer sufficient to achieve the theoretical peak performance of hardware. Based on the analysis of this project, future work (or directions for subsequent optimization) mainly focuses on eliminating synchronization blocking bottlenecks, conducting research on look-ahead pipeline technology, non-blocking communication (MPI_Ibcast) transformation, and CPU affinity scheduling to achieve deep overlap of computation and communication. Meanwhile, with the increase in the number of cores in compute nodes, to reduce intra-node communication overhead, hybrid parallel models (MPI + OpenMP) and strong/weak scalability evaluation on large-scale clusters are also important research directions for further improving HPL's large-scale parallel efficiency.

Références

[1] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark : Past, present and future. *Concurrency and Computation : Practice and Experience*, 15(9):803–820, 2003.

[2] Netlib. HPL – a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Software and documentation page, 2000. Accessed : 2026-02-15.

[3] TOP500. The linpack benchmark. Project page, 2026. Accessed : 2026-02-15.

[4] TOP500. TOP500 description. Methodology page, 2026. Accessed : 2026-02-15.

[5] L. Susan Blackford, Jaeyoung Choi, Andrew Cleary, Edward D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Kendall Stanley, David Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[7] Edward Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 3 edition, 1999.

[8] MPI Forum. MPI : A message-passing interface standard, version 4.1. Standard specification, 2023. Accessed : 2026-02-15.

[9] SchedMD. Slurm workload manager : Resource binding. Documentation page, 2026. Accessed : 2026-02-15.

[10] Open MPI. mpirun(1) – open mpi manual. Manual page, 2026. Accessed : 2026-02-15.

[11] Advanced Micro Devices, Inc. Amd optimizing cpu libraries (AOCL). Product page, 2026. Accessed : 2026-02-15.

[12] Advanced Micro Devices, Inc. Amd optimizing cpu libraries (AOCL) user guide. User guide (PDF), 2026. Accessed : 2026-02-15.

[13] ROMEO HPC (URCA). Description matérielle des ressources ROMEO. Documentation page, 2026. Accessed : 2026-02-15.

Annexe A

Données brutes des expériences HPL

Cette annexe contient les résultats expérimentaux complets obtenus lors de l'exécution de HPL et MiniHPL. Les données correspondent aux mesures directement utilisées pour produire les figures du rapport.

NB	N	ranks	mini$_{t}ime_{s}$	mini$_{g}flops$	hpl$_{t}ime_{s}$	hpl$_{g}flops$	Grid	gflops$_{r}atio_{m}ini_{o}ver_{h}pl$	time$_{r}atio_{m}ini_{o}ver_{h}pl$	gflops$_{g}ap_{h}pl_{m}inus_{m}ini$
64	20736	16	85.347498	69.645474	12.48	476.35	4x4	0.14620651621706726	6.838741826923076	406.70452600000004
128	20736	16	40.828758	145.5853	9.69	613.41	4x4	0.23733766974780326	4.213494117647059	467.8247
192	20736	16	34.731611	171.142851	8.96	663.63	4x4	0.25788896071606165	3.876295870535714	492.487149
256	20736	16	36.052852	164.870922	9.73	611.28	4x4	0.2697142422457794	3.705329085303186	446.40907799999997