SUSTech Technique Report

APAC HPC-AI 2025 Student Competition

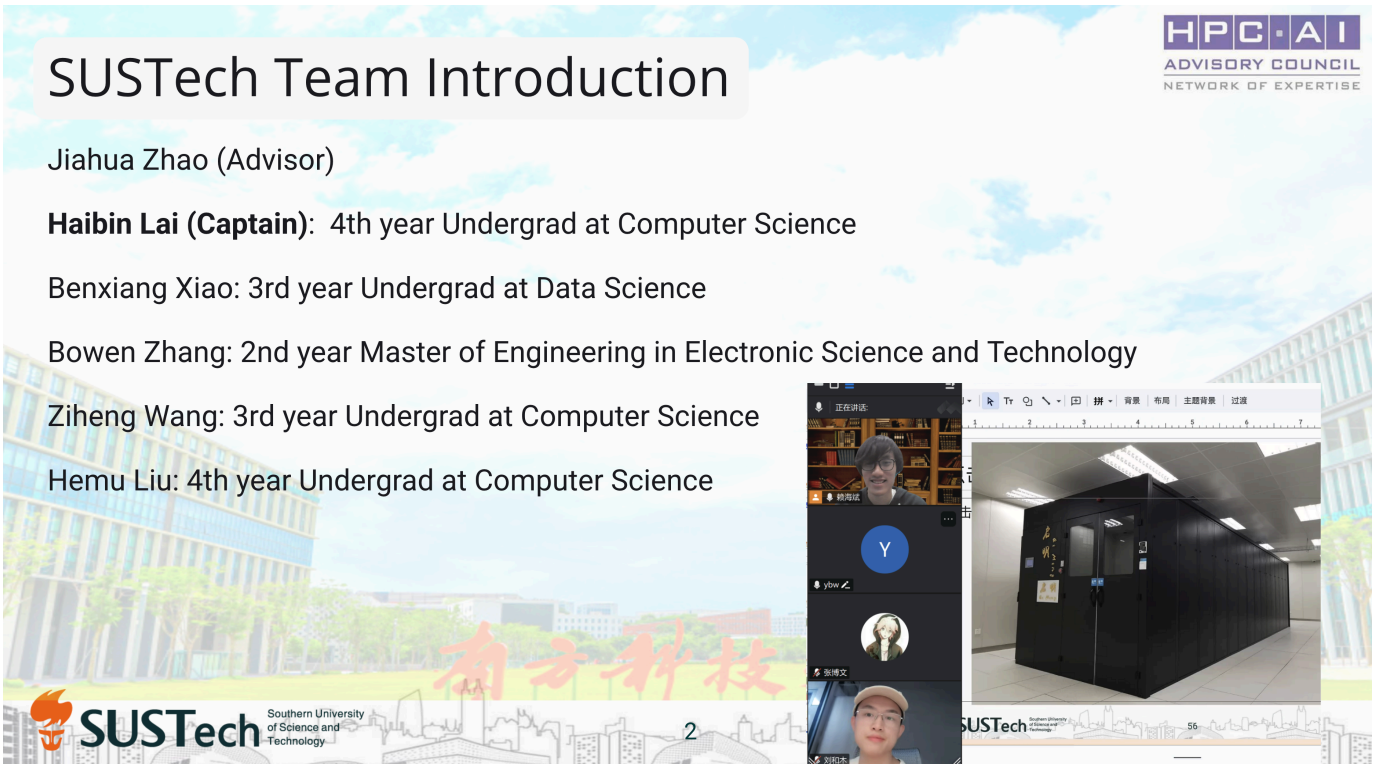
Team:
SUSTech Scout Regiment

Captain:
Haibin Lai

Email:
laihb2022@mail.sustech.edu.cn

Introduction

The **SUSTech Scout Regiment** is a passionate and multidisciplinary team from the **Southern University of Science and Technology (SUSTech)**. Our members come from different academic years and research backgrounds, united by a shared enthusiasm for **high-performance computing** and **scientific software optimization**. We aim to tackle real-world supercomputing challenges while learning from hands-on system-level experiments and performance tuning.



In this year's competition, our team utilized **two distinct high-performance computing environments** to evaluate and optimize our workloads.

For the **CPU track**, we conducted experiments on the **Qiming 2.0 cluster** at SUSTech, which is equipped with **Intel® Xeon® Gold 6138 CPUs** and a **Lustre parallel file system**. This environment closely resembles the **Gadi** and **NSCC** reference platforms provided by the competition organizers. We executed

NWChem workloads on up to **four nodes (160 CPU cores)** to evaluate scalability, communication efficiency, and memory utilization, while applying multi-level optimization techniques to enhance performance.

CPU Cluster Introduction

To take the hardware as close as the Gadi and NSCC platform, we conducted our NWChem experiments on up to 4 nodes (160 CPU cores) of the Qiming 2.0 cluster.

Component	Qiming 2.0 Description
System Name	Qiming 2.0 Supercomputing Cluster
Compute Nodes	409 blade nodes providing high-density parallel computing capability
Storage System	Lustre Parallel file system
Interconnect Network	Infiniband high-speed network (100 Gbps) with low latency

Component	Node Specification
CPU Model	Intel® Xeon® Gold 6138 @ 2.00GHz
Architecture	2 × 20 cores (40 cores total), 2 NUMA nodes
Threads per Core	1
Base / Max Frequency	2.00 GHz / 3.70 GHz

For the **GPU track**, our **DeepSeek-R1** experiments were performed on the **organizers' H200 reference cluster**. We are deeply grateful for the opportunity to access this advanced infrastructure, which allowed our software-level optimizations and fine-grained kernel tuning to fully demonstrate their performance potential.

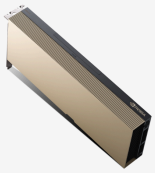
GPU Cluster Introduction

Our performance on Deepseek-R1 2 nodes 16 GPUs workload is achieved on the organizers' H200 reference cluster. We're thankful for the access to this infrastructure, which allows our software optimizations to truly shine.

Cluster

Configuration	Specification
GPUs per Node	8 * H200
Num of Nodes	4

H200 GPU

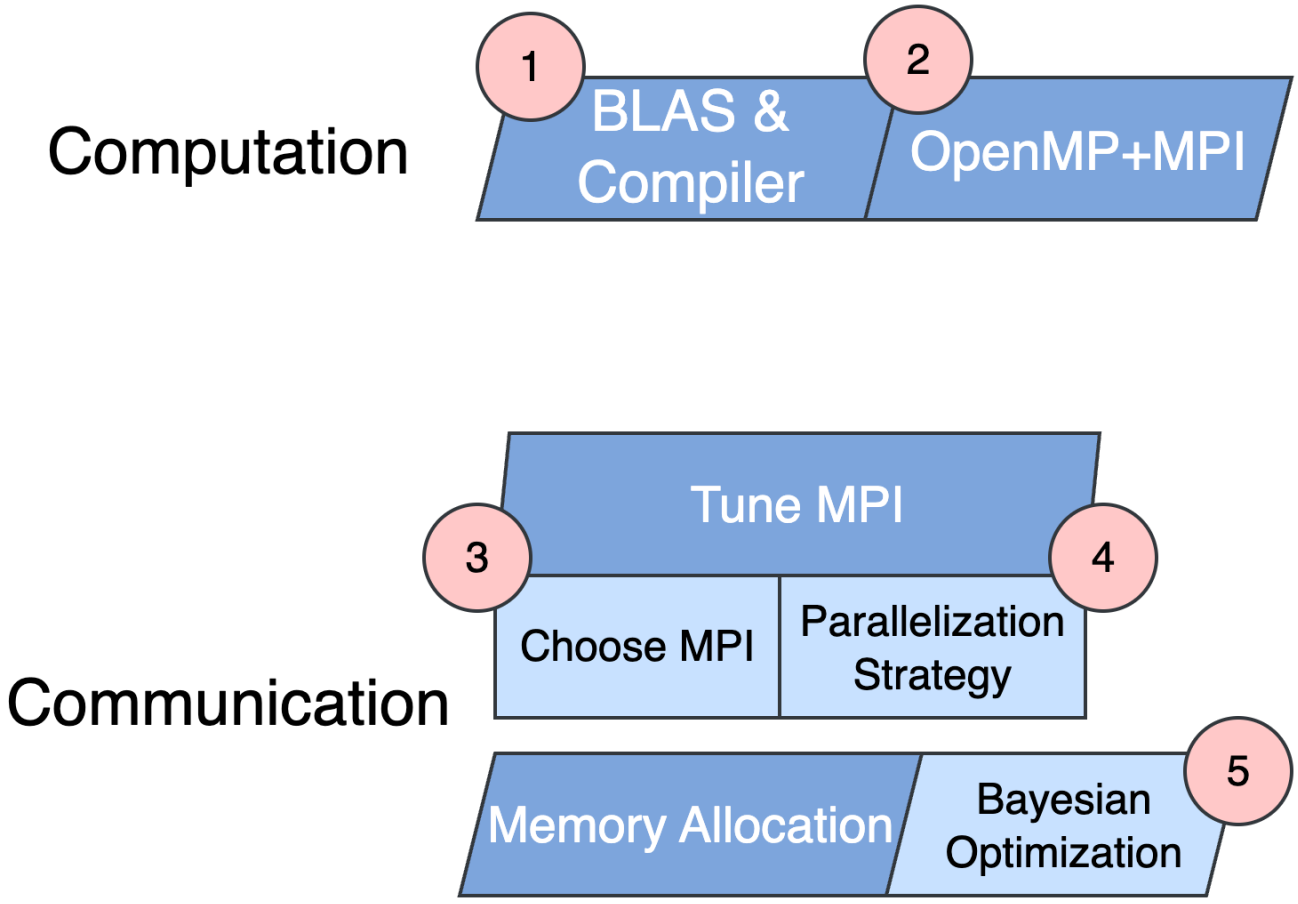


Configuration	Specification
GPU BF16 Tensor Core	1979 TFLOPS
Memory	141GB

NWChem

Overview

NWChem is a scalable, open-source computational chemistry package designed for large-scale simulations on supercomputers. It supports a wide range of scientific models, including **quantum chemistry** and **molecular dynamics**.
Among its core components, the **Self-Consistent Field (SCF)** and **Density Functional Theory (DFT)** modules are particularly representative.
SCF workloads iteratively solve the electronic wavefunction, while DFT workloads extend this with **grid-based numerical integrations**.
Both workloads are **communication-intensive**, relying heavily on **MPI** and **Global Arrays (GA)** to perform distributed computation across nodes.

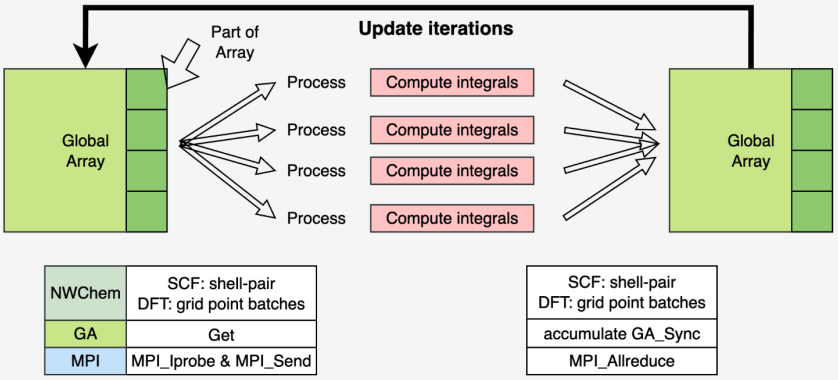


Workload Description

In NWChem, data is organized in **Global Arrays**, which provide a shared-memory abstraction on top of MPI.
During each SCF or DFT iteration, processes **fetch sub-blocks of distributed arrays**, perform local computations, and then **write the results back** to the global data structure.
This design simplifies programming but introduces synchronization and communication overhead.

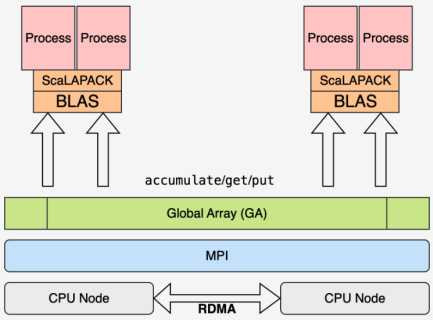
NWChem workload analysis

Both SCF and DFT are communication-intensive and rely heavily on **MPI** and **Global Arrays** for distributed computation.



NWChem	SCF: shell-pair DFT: grid point batches
GA	Get
MPI	MPI_Iprobe & MPI_Send

SCF: shell-pair DFT: grid point batches
accumulate GA_Sync
MPI_Allreduce



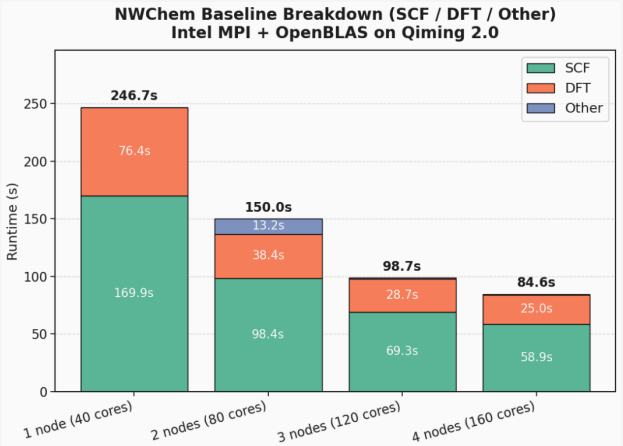
For our experiments, we ran the **baseline NWChem** build using **Intel MPI** and **OpenBLAS**.
We evaluated performance based on **wall-clock time**, ensuring that all outputs were scientifically correct.
However, we observed **limited scalability beyond four nodes**, indicating potential bottlenecks in communication and computation.

Baseline Running

Evaluation Metrics: Wall Time (lower is better)

Qiming 2.0 doesn't have precompiled NWChem, we compiled with Intel-MPI 2021 with latest OpenBLAS.

SCF Domains the computation.
Both workloads scales when using 2 nodes from 1 node. When using 3 nodes and 4 nodes, they didn't scale very well.




```
Total DFT energy =     -917.738245100032
One electron energy =  -3287.425825887609
Coulomb energy =        1466.406697276750
Exchange-Corr. energy =  -112.234989196318
Nuclear repulsion energy = 1015.515872707146
```

```
Total SCF energy =     -912.818341194699
One-electron energy =  -3287.137154917513
Two-electron energy =   1358.802941015668
Nuclear repulsion energy = 1015.515872707146
```

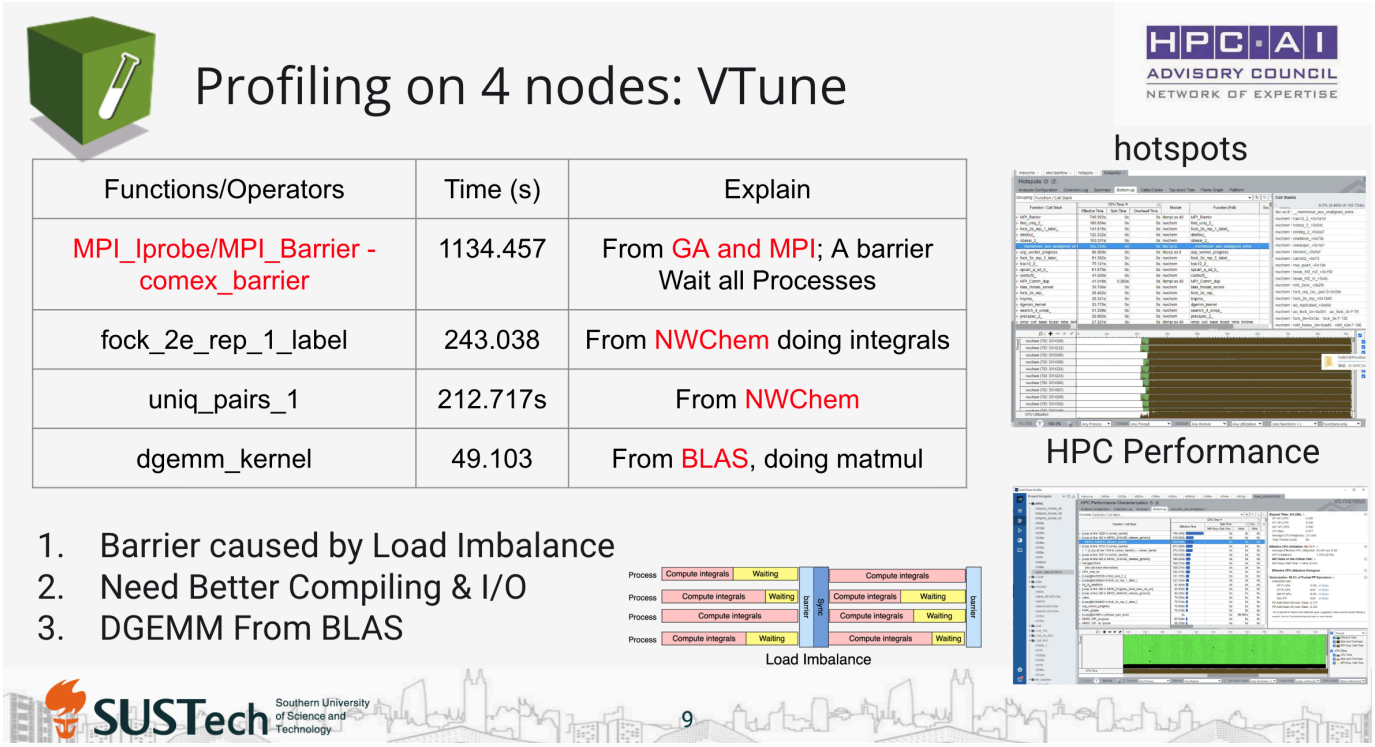

Performance Analysis

We used **Intel VTune Profiler** to analyze runtime behavior and identify bottlenecks.
The results revealed that approximately **28% of total execution time** was spent in **communication**,

primarily due to **MPI_Barrier** and **MPI_Iprobe** operations.
This suggests **load imbalance** across processes, where some ranks finish computation earlier and wait for synchronization.

The second major bottleneck lies in **computation efficiency**.
Many of NWChem's core operators are **memory-bound**, meaning their performance is constrained by memory bandwidth rather than compute throughput.
As a result, optimization opportunities mainly come from **compiler-level tuning**, **loop transformations**, and **I/O improvements** rather than algorithmic restructuring.

Finally, we identified significant time spent in **BLAS routines**, especially **DGEMM (Double-precision General Matrix Multiplication)**.
These are computationally heavy but can be improved through **optimized BLAS libraries**, **multi-threading**, or **NUMA-aware scheduling**.



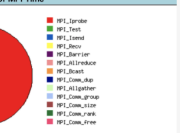
MPI Profiling Insights

Using **IPM (Integrated Performance Monitoring)**, we further examined the MPI communication pattern.
The profiling results show that a large proportion of processes frequently invoke **MPI_Iprobe**, indicating they are **polling for incoming data** instead of performing useful computation.
This behavior confirms our earlier observation: the system suffers from **communication waiting and load imbalance**, which severely limits parallel efficiency.

Profiling on 4 nodes: IPM Analysis

HPC-AI ADVISORY COUNCIL
NETWORK OF EXPERTISE

Computation			Communication
Event	Count	Pop	% of MPI Time



HPM Counter Statistics

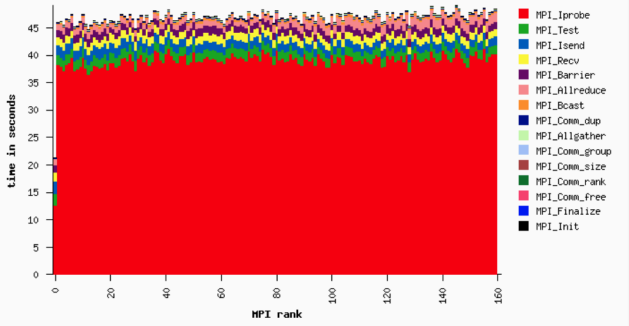
Event	Ntasks	Avg	Min(rank)	Max(rank)

Communication Event Statistics (100.00% detail, ~6.0507e-03 error)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Iprobe	0	1158048316	6247.954	0.000e+00	2.500e-01	83.51	32.02
MPI_Test	0	981680880	268.799	0.000e+00	1.769e-01	3.59	1.38
MPI_Isend	40	876161170	228.365	0.000e+00	2.379e-01	3.05	1.17
MPI_Barrier	0	1097120	207.313	2.861e-06	1.770e-01	2.77	1.06
MPI_Recv	40	876161170	207.093	0.000e+00	2.005e-01	2.77	1.06
MPI_Allreduce	5120	365120	55.820	1.597e-05	1.461e-01	0.75	0.29
MPI_Allreduce	8	246080	34.990	5.007e-06	1.195e-02	0.47	0.18

83% MPI is waiting!

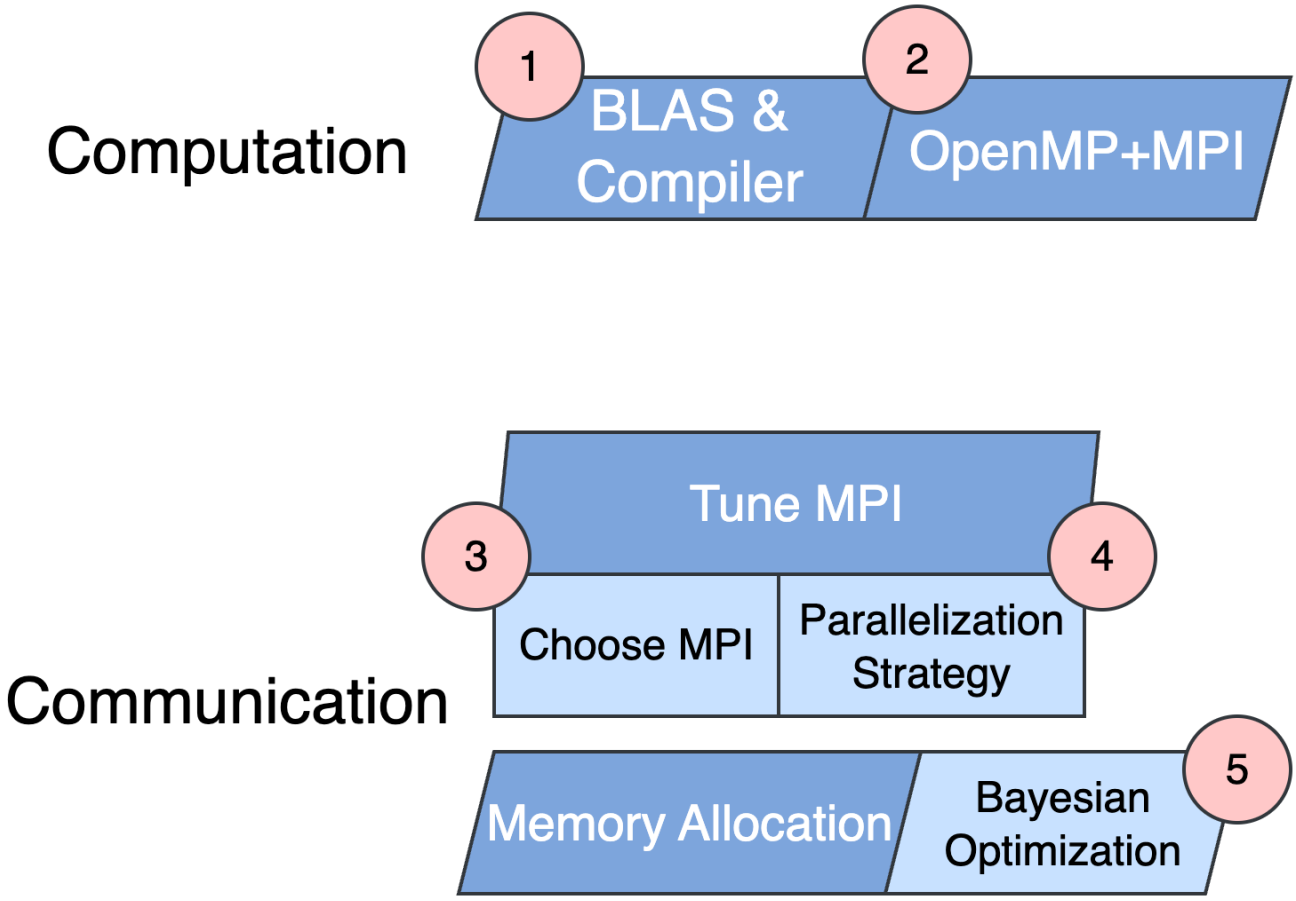
MPI_Iprobe (red) Domains (For nonblocking test), MPI_Test(Green)
MPI_Recv(Yellow) MPI_Send(Blue)



Processes in each nodes are waiting data

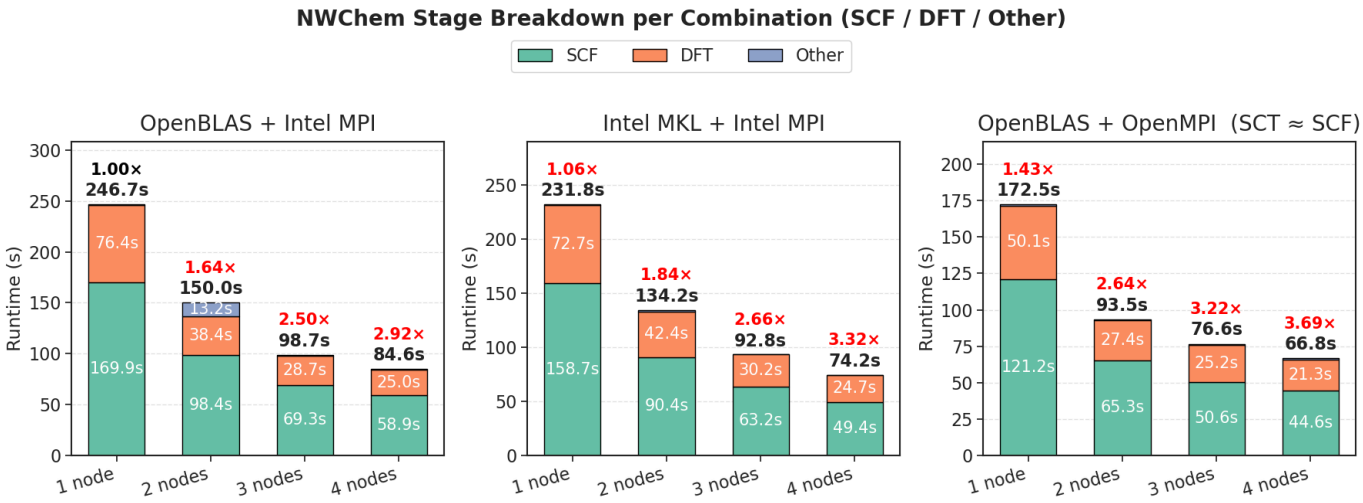
Optimization Strategies

Based on the performance bottlenecks identified through **VTune** and **IPM** profiling, we proposed and implemented **five key optimizations** targeting both computation and communication inefficiencies.

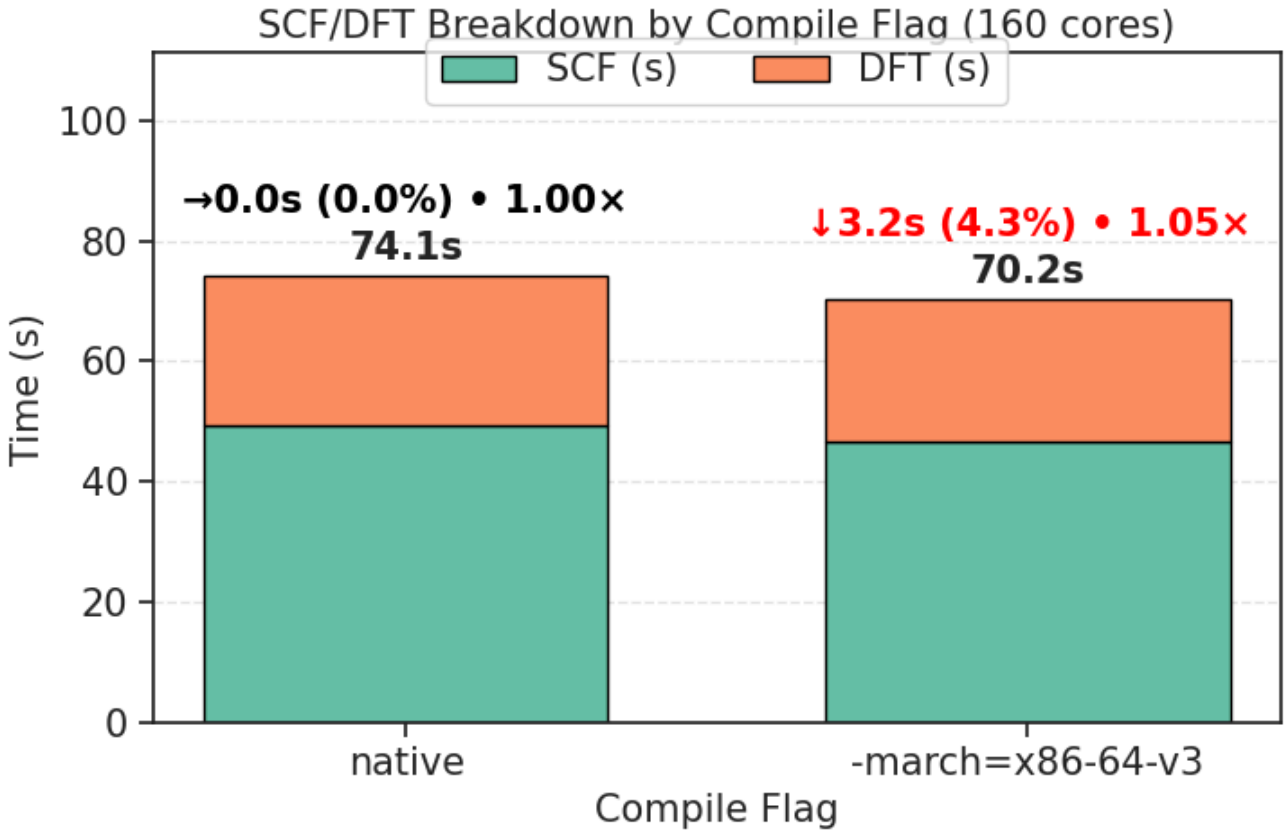


1. Compiler and BLAS Optimization

To improve computational efficiency, we experimented with multiple compiler and BLAS combinations. We compared **Intel OneAPI (ICC + MKL)** and **GCC + OpenBLAS**, observing that MKL generally provides better memory alignment and vectorization performance for dense matrix operations such as **DGEMM**.



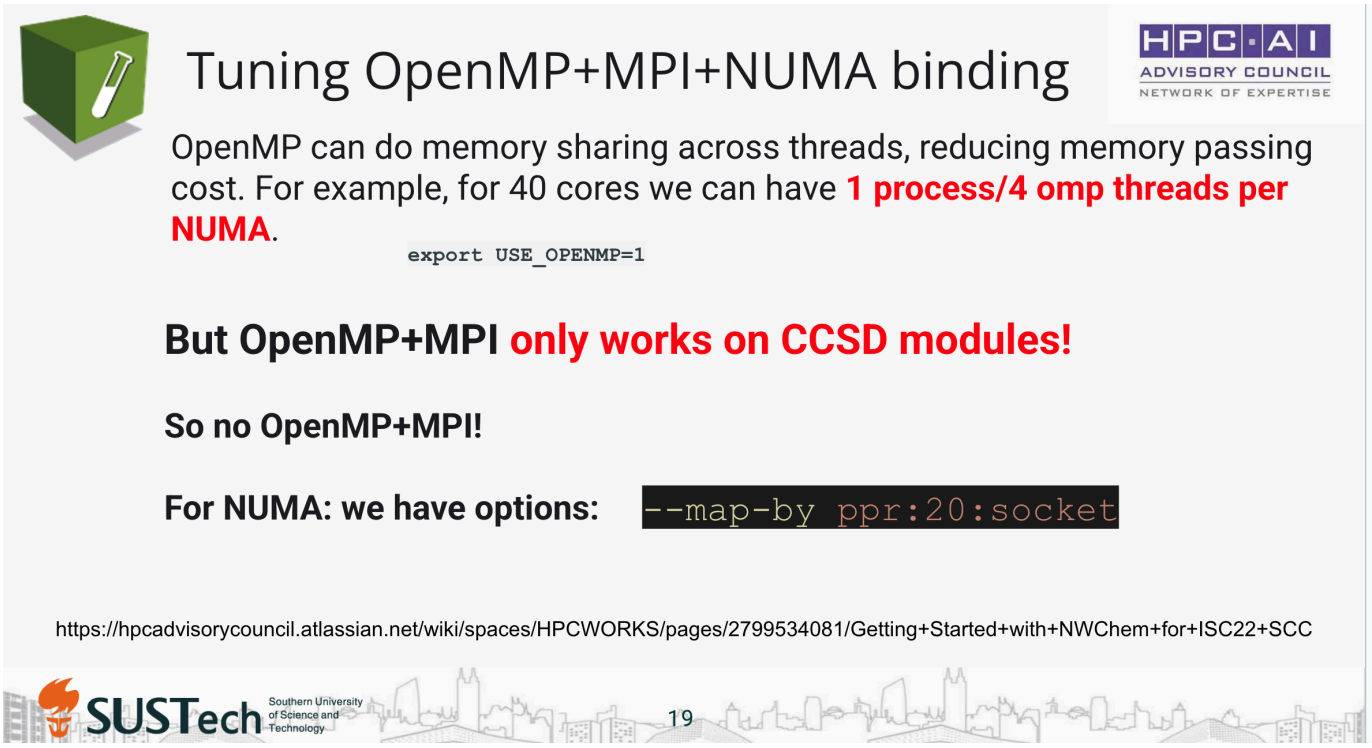
By enabling advanced compiler flags (e.g., `-O3`, and `-fopenmp`), we improved the efficiency of memory-bound operators within SCF and DFT kernels.



2. NUMA + OpenMP + MPI Hybrid Parallelism

To better exploit on-node parallelism, we introduced **hybrid OpenMP + MPI parallelization**. Each node runs fewer MPI ranks, with OpenMP threads handling intra-node computations.

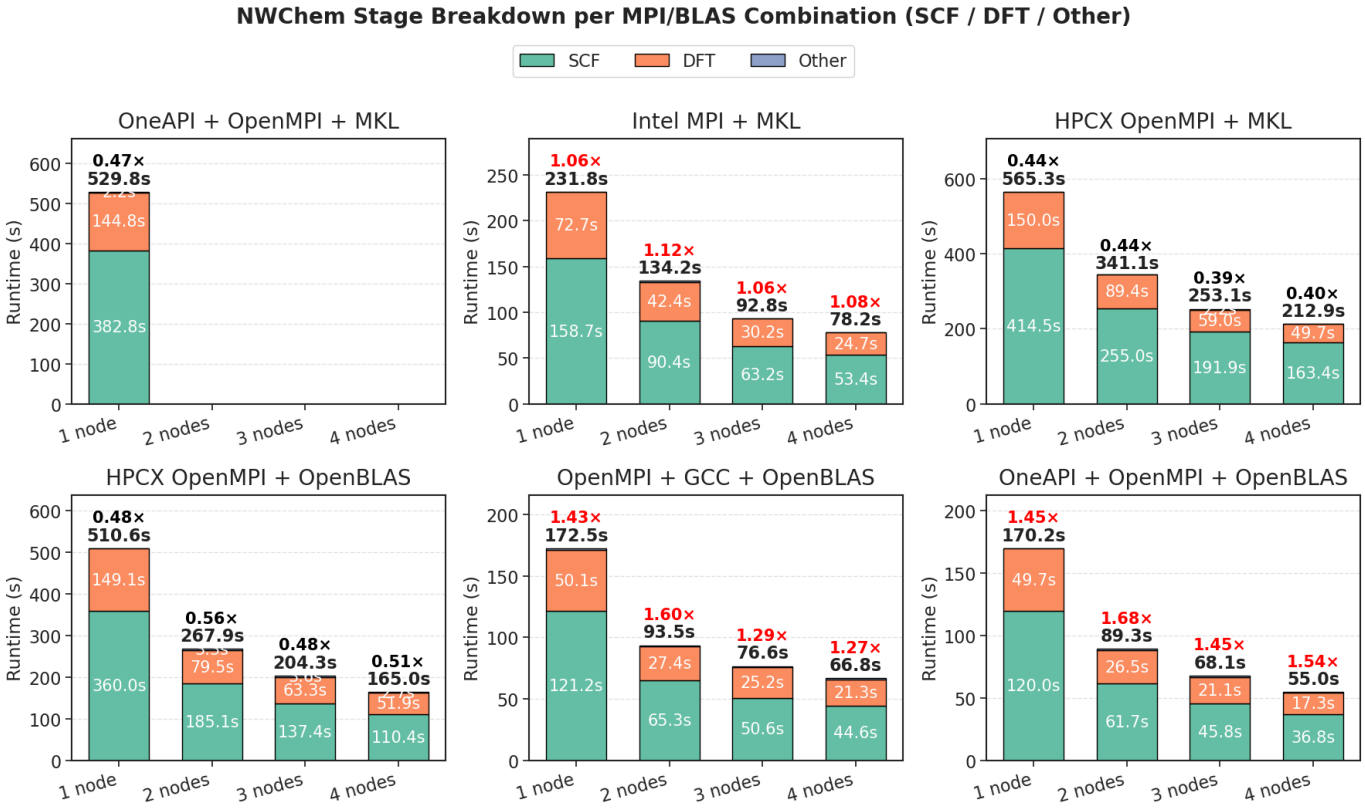
This reduces inter-rank communication and makes better use of shared caches and NUMA domains.
We observed that OpenMP+MPI is not allowed in SCF and DFT workload, but we can still do the NUMA binding.



3. Communication and Load-Balancing Optimization

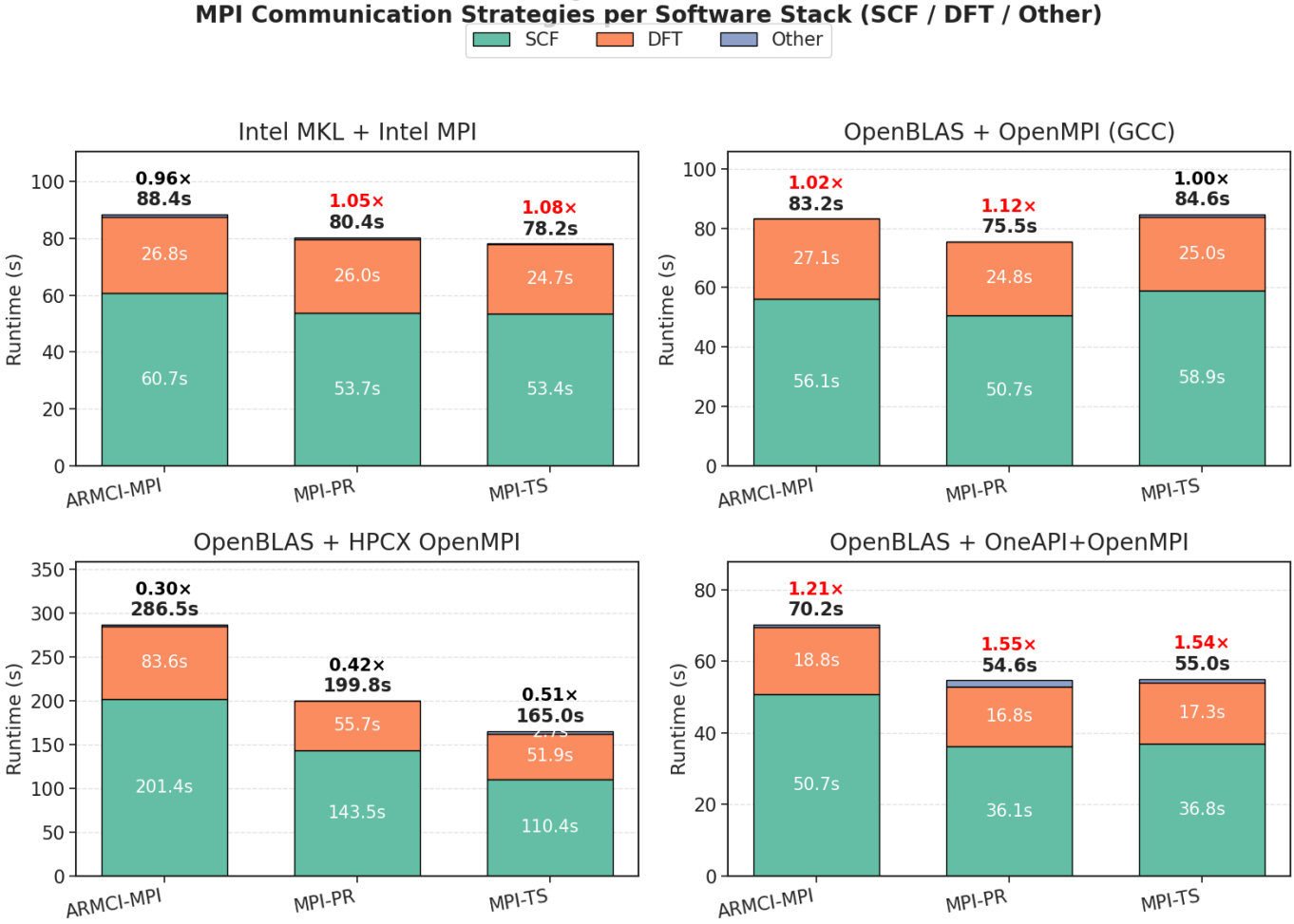
Communication imbalance—evident from excessive time in **MPI_Barrier** and **MPI_Iprobe**—was addressed by experimenting with **different MPI implementations and configurations**.

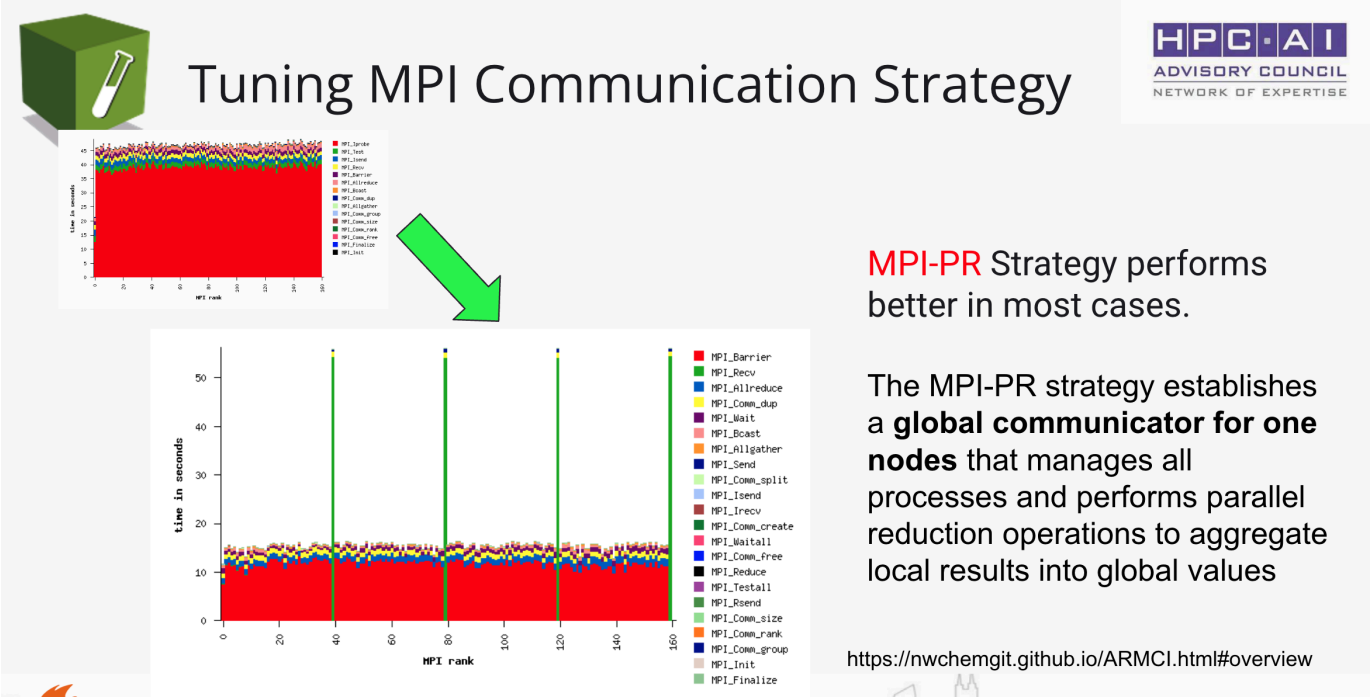
We tested **Intel MPI**, **hpcx**, and **OpenMPI**, tuning runtime parameters and collective algorithms.



Moreover, we explored **MPI process remapping** to improve affinity and reduce latency.

For the most communication-heavy phases, we evaluated **MPI-PR (Persistent Request)** strategies to overlap communication with computation, which reduced idle waiting time.





4. Memory Allocation

Memory locality significantly impacts the performance of NWChem's Global Arrays.
We applied bayesian Optimization on Memory allocation.
This tuning effectively reduced remote memory access and improved memory bandwidth utilization across nodes.

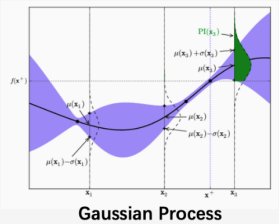
Use Bayesian Optimization on Memory allocation

HPC-AI ADVISORY COUNCIL NETWORK OF EXPERTISE

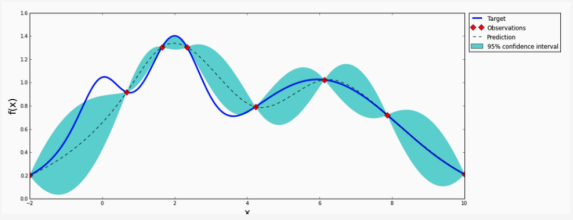
Algorithm 1 Sequential Model-Based Optimization

Input: $f, \mathcal{X}, S, \mathcal{M}$
$\mathcal{D} \leftarrow \text{INITSAMPLES}(f, \mathcal{X})$
for $i \leftarrow |\mathcal{D}|$ **to** T **do**
  $p(y \mid \mathbf{x}, \mathcal{D}) \leftarrow \text{FITMODEL}(\mathcal{M}, \mathcal{D})$
  $\mathbf{x}_i \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} S(\mathbf{x}, p(y \mid \mathbf{x}, \mathcal{D}))$
  $y_i \leftarrow f(\mathbf{x}_i)$ ▷ Expensive step
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{x}_i, y_i)$
end for

SMBO



Gaussian Process



Unlike grid or random search, **Bayesian Optimization** builds a **probabilistic model (surrogate)** of the **performance function** and iteratively selects new configurations based on an **acquisition function** that balances exploration and exploitation.
This approach significantly **reduces the number of expensive experiments** while converging toward optimal parameters of memory params, enabling us to achieve higher performance with fewer trials.

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. 2012. *Practical Bayesian Optimization of Machine Learning Algorithms.* arXiv preprint arXiv:1206.2944. https://doi.org/10.48550/arXiv.1206.2944

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. *Optuna: A Next-generation Hyperparameter Optimization Framework.* In **Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)**, August 4–8, 2019, Anchorage, AK, USA. ACM, New York, NY, USA, 10 pages. https://doi.org/10.48550/arXiv.1907.10902

Use Bayesian Optimization on Memory allocation

HPC-AI ADVISORY COUNCIL NETWORK OF EXPERTISE

Framework: Optuna
Performance/Optimized functions:
Wall Time

Round	MPI	Stack	Heap	Global	Wall Time
1	MPI_PR	6500	150	10000	55.3
2	MPI_PR	14500	400	10000	118.7
3	MPI_PR	11000	350	24000	56.6
4	ARMCI-MPI	4000	400	31000	87
5	MPI_PR	2000	200	7000	87.1
6	MPI_TS	16000	250	26000	62.2
7	MPI_TS	12500	100	4000	54

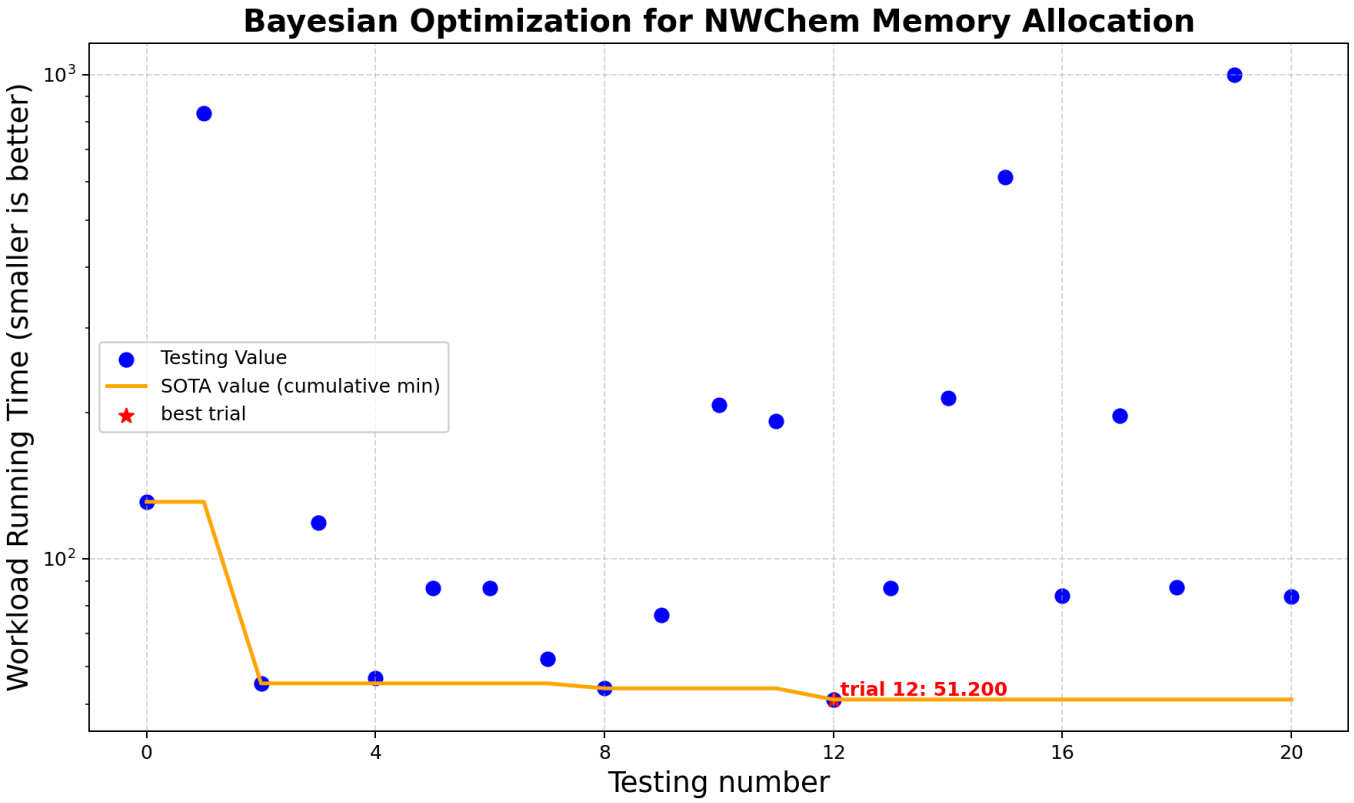
Input Control Config
Stack(MB): [4000, 16000], 500 per samples
Heap(MB): [50, 500], 50 per samples
Global(MB): [4000, 32000], 500 per samples
MPI: [MPI-PR, MPI-TS, ARM-CI]

4 nodes
OneAPI icc-OpenMPI+OpenBLAS

Algorithm 1 Sequential Model-Based Optimization

Input: $f, \mathcal{X}, S, \mathcal{M}$
$\mathcal{D} \leftarrow \text{INITSAMPLES}(f, \mathcal{X})$
for $i \leftarrow |\mathcal{D}|$ **to** T **do**
  $p(y \mid \mathbf{x}, \mathcal{D}) \leftarrow \text{FITMODEL}(\mathcal{M}, \mathcal{D})$
  $\mathbf{x}_i \leftarrow \arg\max_{\mathbf{x} \in \mathcal{X}} S(\mathbf{x}, p(y \mid \mathbf{x}, \mathcal{D}))$
  $y_i \leftarrow f(\mathbf{x}_i)$ ▷ Expensive step
  $\mathcal{D} \leftarrow \mathcal{D} \cup (\mathbf{x}_i, y_i)$
end for

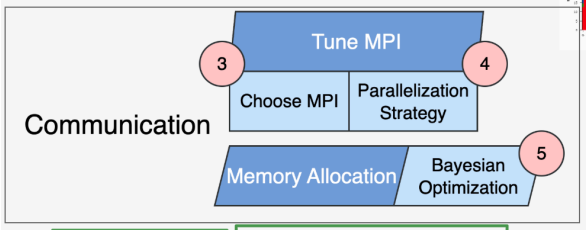
SMBO

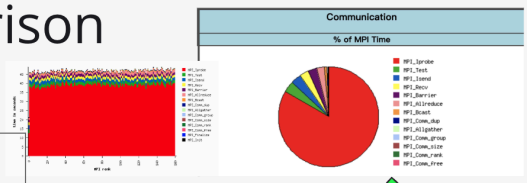


Summary of Improvements

After applying these optimizations, we observed measurable performance gains across SCF and DFT workloads.
The optimized compiler + BLAS configuration delivered faster computation kernels.
Collectively, these improvements led to **better scalability and reduced wall time on up to four nodes**, demonstrating the importance of **co-designing communication, computation, and memory strategies** for scientific HPC applications.

Profile Comparison





HPM Counter Statistics

Event	Ntasks	Avg	Min(rank)	Max(rank)

Communication Event Statistics (100.00% detail, 3.0808e-03 error)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Barrier	0	3756952	1848.092	5.069e-04	8.496e-01	70.35	19.78
MPI_Comm_dup	0	800	153.714	1.041e+00	1.044e+00	5.85	1.65
MPI_Recv					2.135e+00	5.32	1.50
MPI_Allreduce					1.097e-01	2.63	0.74
MPI_Allreduce					1.137e-02	1.77	0.50
MPI_Recv					7.760e-01	1.53	0.43
MPI_Allreduce					4.672e-02	1.16	0.33
MPI_Allgather					1.114e-02	0.85	0.24
MPI_Bcast					1.481e-02	0.62	0.17
MPI_Wait					3.648e-02	0.57	0.16
MPI_Wait					3.372e-02	0.55	0.16
MPI_Recv					5.295e-01	0.53	0.15

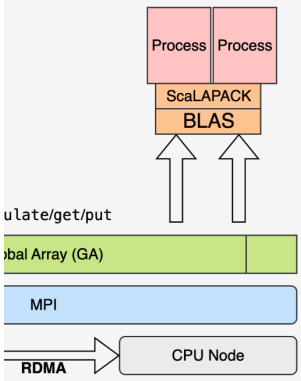
MPI +54%

Memory +6%

MPI-PR +5%~+20%

MPI Barrer: 83.51% -> 70.35%

Conclusion



Strategy

- CPU: **4 nodes 160 cores**
- BLAS: **OpenBLAS**
- Specific **Compile Flags**
- **NUMA mapping**
- MPI: **Intel icc compiled OpenMPI**
- Communication Strategy: **MPI-PR**
- Specific **Memory allocation**

Memory	Best Allocation
Stack	9000MB
Heap	400MB
Global	18500MB

Acceleration

- 246.7s -> 51.2s. **4.82X faster(-79%Time)** than single node baseline.
- 84.6s -> 51.2s. **1.652X faster(-40%Time)** than 4 nodes baseline.

Suggestions for Running NWChem

HPC-AI ADVISORY COUNCIL NETWORK OF EXPERTISE

Notice for Users

- Intel MPI **2022** doesn't have **ifort**
- **Choose your best MPI**
- **AOCL** may not work well on Intel Machines

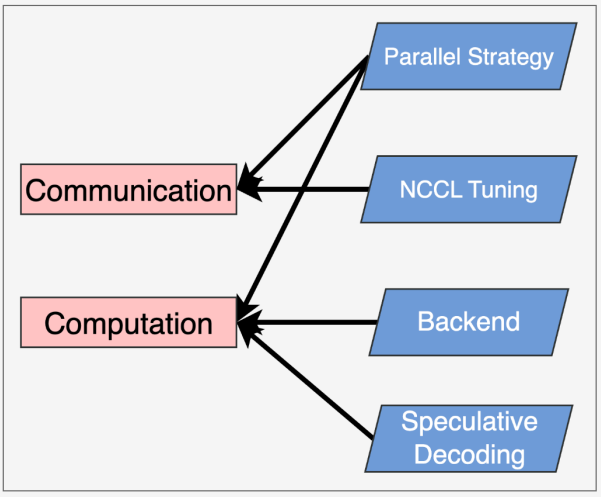
Suggestion for Developers

- **Tree-based Barrier**
- **Need OpenMP+MPI support**
- Better algorithms on operators like sort

DeepSeek-R1 Optimization

Deepseek-R1 Optimization Roadmap

HPC-AI ADVISORY COUNCIL NETWORK OF EXPERTISE



TP, DP, PP

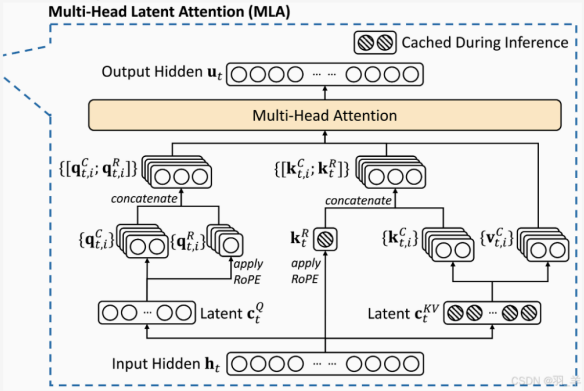
Tree/Ring, LL, LL128

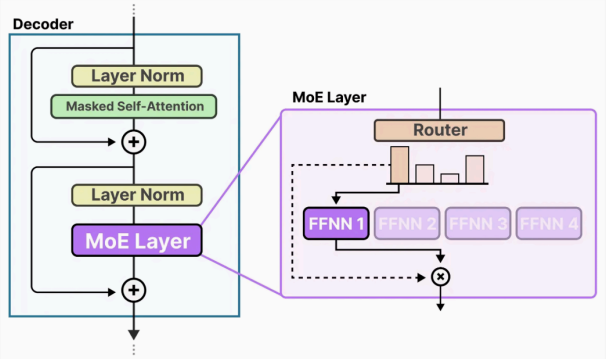
FlashAttention3, FlashMLA

auto-regressive boosting

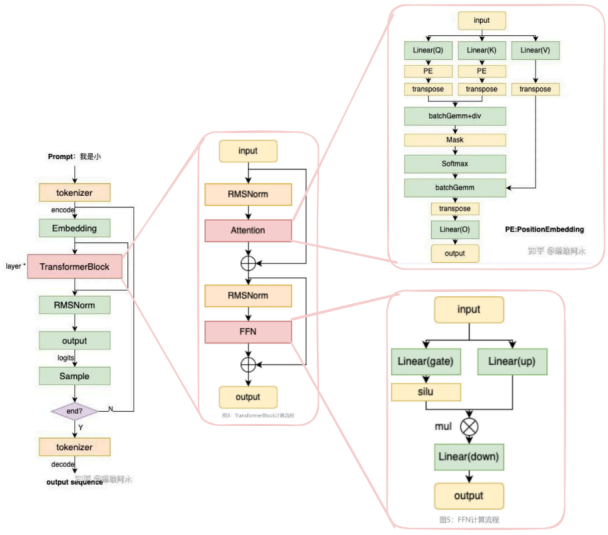
Deepseek-R1 Introduction

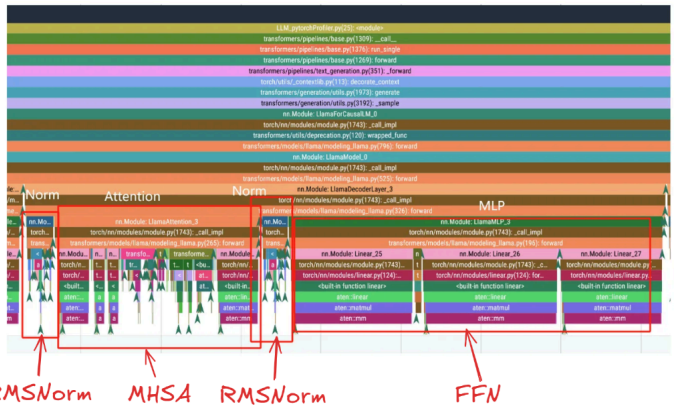
DeepSeek-R1 (671B) achieves strong performance across various reasoning and language tasks.
Its **MLA architecture** enables more efficient use of attention, while the **MoE design** activates only **37 B parameters** per inference.



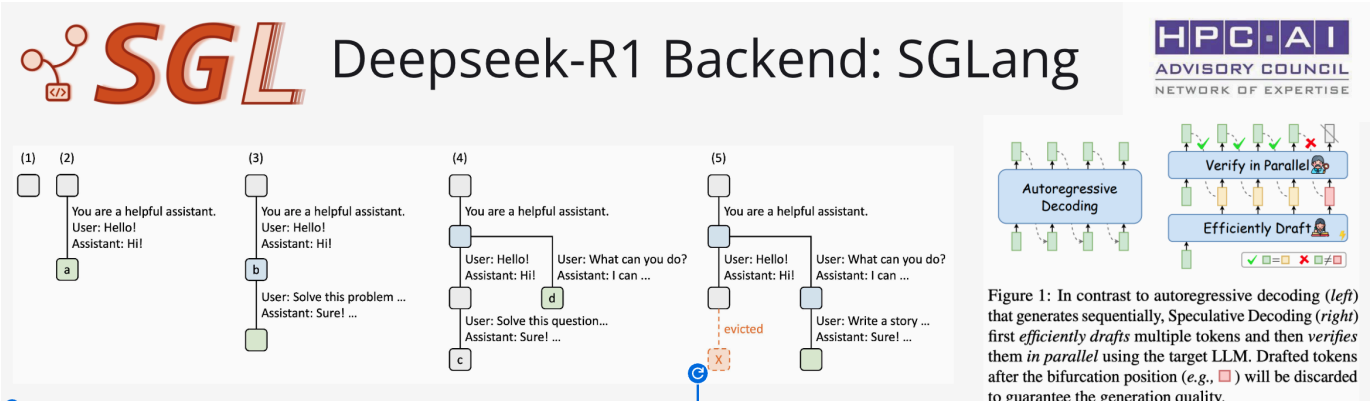


DeepSeek-R1 (671B) achieves strong performance across various reasoning and language tasks.
Its MLA architecture enables more efficient use of attention, while the MoE design activates only 37 B parameters per inference.





SGLang supports Tensor Parallelism (TP), Pipeline Parallelism (PP), and Data Parallelism (DP), enabling efficient large-scale distributed inference across multiple GPUs and nodes.
It also integrates advanced features such as speculative decoding and Radix Attention, which significantly improve decoding throughput and reduce latency.
Through studying and experimenting with SGLang, we gained a deep understanding of its scheduling and parallel strategies. In our benchmarks, it demonstrated strong scalability and excellent performance efficiency on modern GPU clusters.

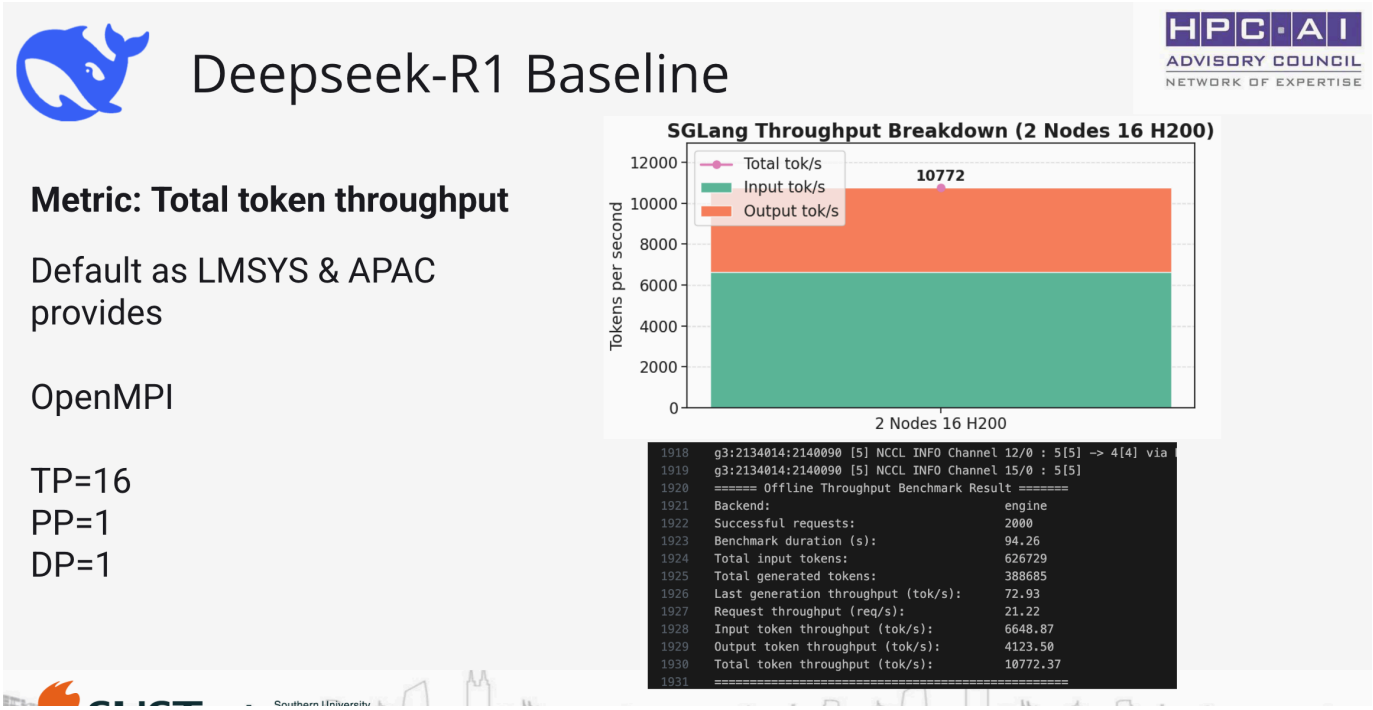


Baseline Setup

For the **GPU track**, we conducted experiments on the **organizers' reference H200 cluster**, using a **two-node, 16-GPU configuration** (8 × H200 per node, connected with NVLink + InfiniBand).
We evaluated multiple **parallelism strategies**, including:

- **Tensor Parallelism (TP)** – splitting matrix multiplications across GPUs,
- **Pipeline Parallelism (PP)** – dividing layers across nodes to balance memory usage, and
- **Data Parallelism (DP)** – replicating models to process multiple requests concurrently.

Our baseline employed the **SGLang engine** running with the **FA3 backend**, which served as a stable starting point for functional and performance correctness.

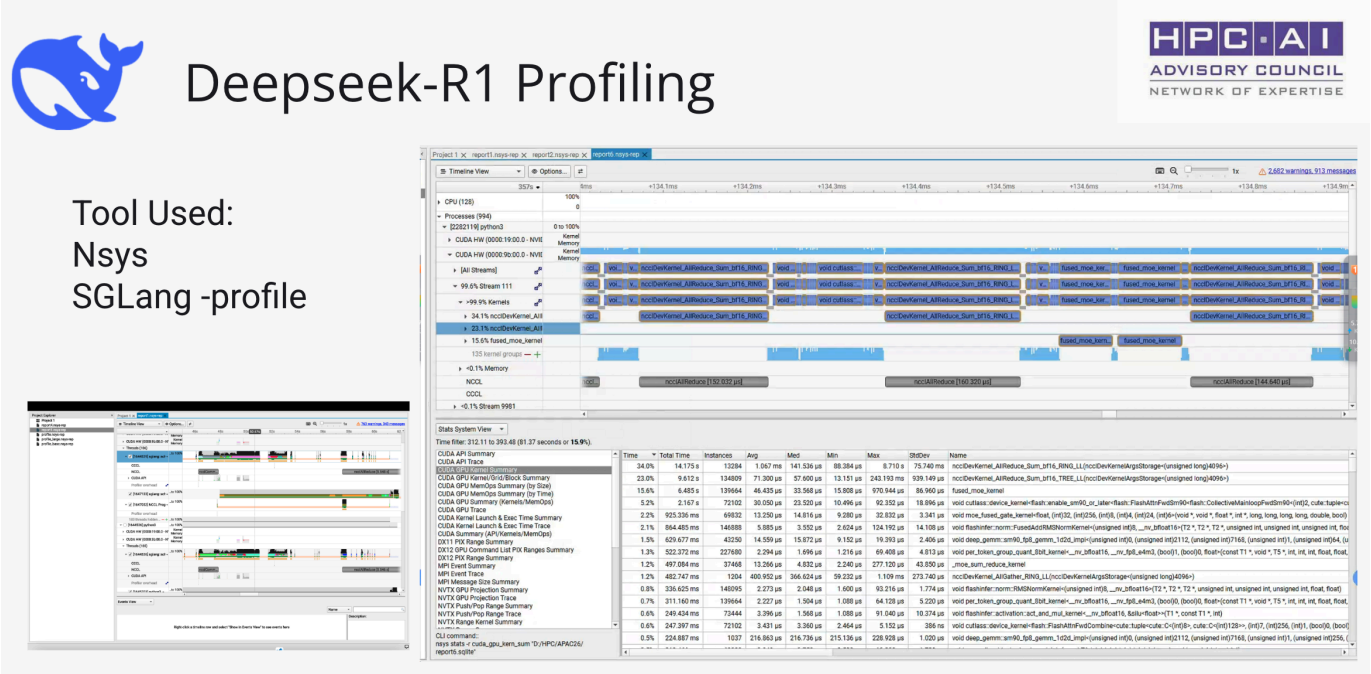


Bottleneck Profiling

We used **NVIDIA Nsight Systems (nsys)** to profile both prefill and decode stages.
Profiling results indicated that the main performance bottleneck came from **communication overhead** rather than computation.
Specifically, NCCL traces showed numerous small but frequent data exchanges between GPUs, leading to

high communication latency and **low link utilization**.

In multi-node cases (PP > 1 or DP > 1), synchronization between pipeline stages further aggravated the delay, dominating overall iteration time.



Deepseek-R1 Profiling

Tool Used:
Nsys
SGLang -profile

We observed **small runtime but large instances** of NCCL events happens in every token generations:

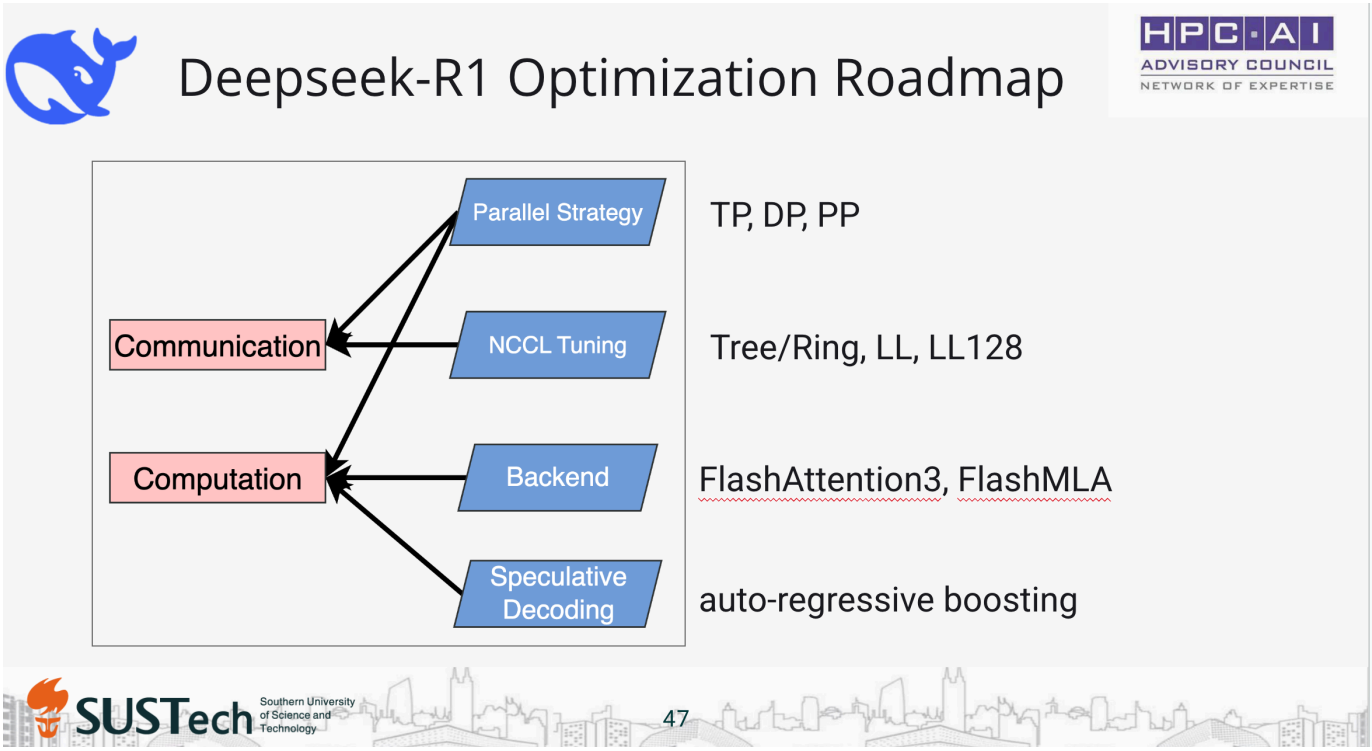
Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
34.0%	14.159 s	13283	1.066 ms	141.536 µs	88.384 µs	8.710 s	75.742 ms	ncclDevKernel_AllReduce_Sum_bf16_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
23.1%	9.612 s	134809	71.300 µs	57.600 µs	13.151 µs	243.193 ms	939.149 µs	ncclDevKernel_AllReduce_Sum_bf16_TREE_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
15.6%	6.485 s	139664	46.435 µs	33.568 µs	15.808 µs	970.944 µs	86.960 µs	fused_moe_kernel
5.2%	2.167 s	72102	30.050 µs	23.520 µs	10.496 µs	92.352 µs	18.896 µs	void cutlass::device_kernel<flash::enable_sm90_or_later<flash::FlashAttnFwdSm90<flash::CollectiveMainloopFwdSm90<(int)2, cute::tuple<c
2.2%	925.336 ms	69832	13.250 µs	14.816 µs	9.280 µs	32.832 µs	3.341 µs	void moe_fused_gate_kernel<float, (int)32, (int)256, (int)8, (int)4, (int)24, (int)6>(void *, void *, float *, int *, long, long, long, long, double, bool)
2.1%	864.485 ms	146888	5.885 µs	3.552 µs	2.624 µs	124.192 µs	14.108 µs	void flashinfer::norm::FusedAddRMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, T2 *, unsigned int, unsigned int, unsigned int, floa
1.5%	629.677 ms	43250	14.559 µs	15.872 µs	9.152 µs	19.393 µs	2.406 µs	void deep_gemm::sm90_fp8_gemm_1d2d_impl<(unsigned int)0, (unsigned int)2112, (unsigned int)7168, (unsigned int)1, (unsigned int)64, (u
1.3%	522.337 ms	227676	2.294 µs	1.696 µs	1.216 µs	69.408 µs	4.813 µs	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)1, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float,
1.2%	497.084 ms	37468	13.266 µs	4.832 µs	2.240 µs	277.120 µs	43.850 µs	_moe_sum_reduce_kernel

Deepseek-R1 Profiling

Tool Used:
Nsys
SGLang -profile

Better MOE Kernels in backend library may compute faster for the model.

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
34.0%	14.159 s	13283	1.066 ms	141.536 µs	88.384 µs	8.710 s	75.742 ms	ncclDevKernel_AllReduce_Sum_bf16_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
23.1%	9.612 s	134809	71.300 µs	57.600 µs	13.151 µs	243.193 ms	939.149 µs	ncclDevKernel_AllReduce_Sum_bf16_TREE_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
15.6%	6.485 s	139664	46.435 µs	33.568 µs	15.808 µs	970.944 µs	86.960 µs	fused_moe_kernel
5.2%	2.167 s	72102	30.050 µs	23.520 µs	10.496 µs	92.352 µs	18.896 µs	void cutlass::device_kernel<flash::enable_sm90_or_later<flash::FlashAttnFwdSm90<flash::CollectiveMainloopFwdSm90<(int)2, cute::tuple<c
2.2%	925.336 ms	69832	13.250 µs	14.816 µs	9.280 µs	32.832 µs	3.341 µs	void moe_fused_gate_kernel<float, (int)32, (int)256, (int)8, (int)4, (int)24, (int)6>(void *, void *, float *, int *, long, long, long, long, double, bool)
2.1%	864.485 ms	146888	5.885 µs	3.552 µs	2.624 µs	124.192 µs	14.108 µs	void flashinfer::norm::FusedAddRMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, T2 *, unsigned int, unsigned int, unsigned int, floa
1.5%	629.677 ms	43250	14.559 µs	15.872 µs	9.152 µs	19.393 µs	2.406 µs	void deep_gemm::sm90_fp8_gemm_1d2d_impl<(unsigned int)0, (unsigned int)2112, (unsigned int)7168, (unsigned int)1, (unsigned int)64, (u
1.3%	522.337 ms	227676	2.294 µs	1.696 µs	1.216 µs	69.408 µs	4.813 µs	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)1, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float,
1.2%	497.084 ms	37468	13.266 µs	4.832 µs	2.240 µs	277.120 µs	43.850 µs	_moe_sum_reduce_kernel
1.2%	482.747 ms	1204	400.952 µs	366.624 µs	59.232 µs	1.109 ms	273.740 µs	ncclDevKernel_AllGather_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
0.8%	336.583 ms	148092	2.272 µs	2.048 µs	1.600 µs	93.216 µs	1.772 µs	void flashinfer::norm::RMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, T2 *, unsigned int, unsigned int, unsigned int, float, float)
0.7%	311.160 ms	139664	2.227 µs	1.504 µs	1.088 µs	64.128 µs	5.220 µs	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)0, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float,
0.6%	249.434 ms	73444	3.396 µs	1.568 µs	1.088 µs	91.040 µs	10.374 µs	void flashinfer::activation::act_and_mul_kernel<__nv_bfloat16, &silu<float>>(T1 *, const T1 *, int)
0.6%	247.397 ms	72102	3.431 µs	3.360 µs	2.464 µs	5.152 µs	386 ns	void cutlass::device_kernel<flash::FlashAttnFwdCombine<cute::tuple<cute::C<(int)8>, cute::C<(int)128>>, (int)7, (int)256, (int)1, (bool)0, (bool)



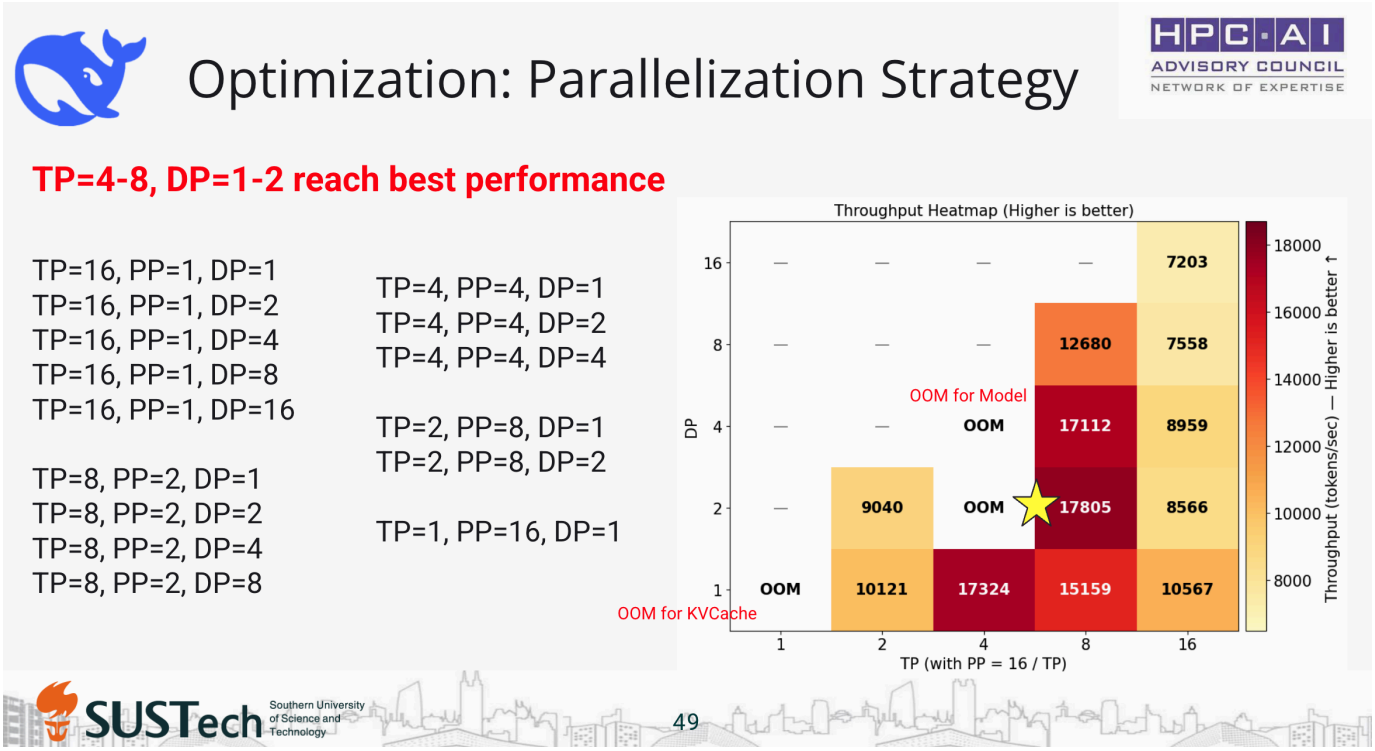
1. Parallelization Strategy

**

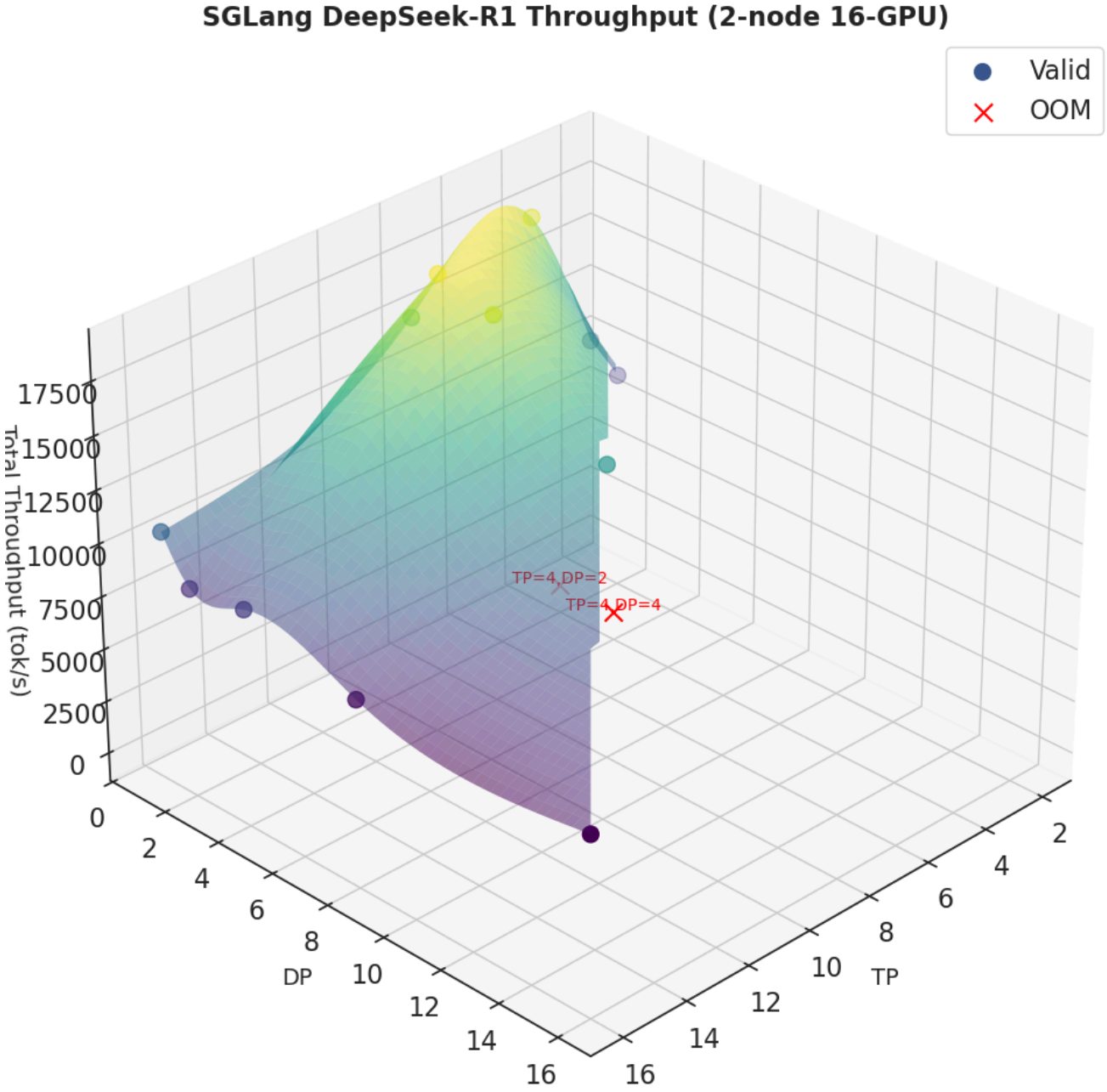
For TP=8 and PP=2, all 8 H200 GPUs within a node are used for TP, while PP spans nodes — minimizing inter-node communication.

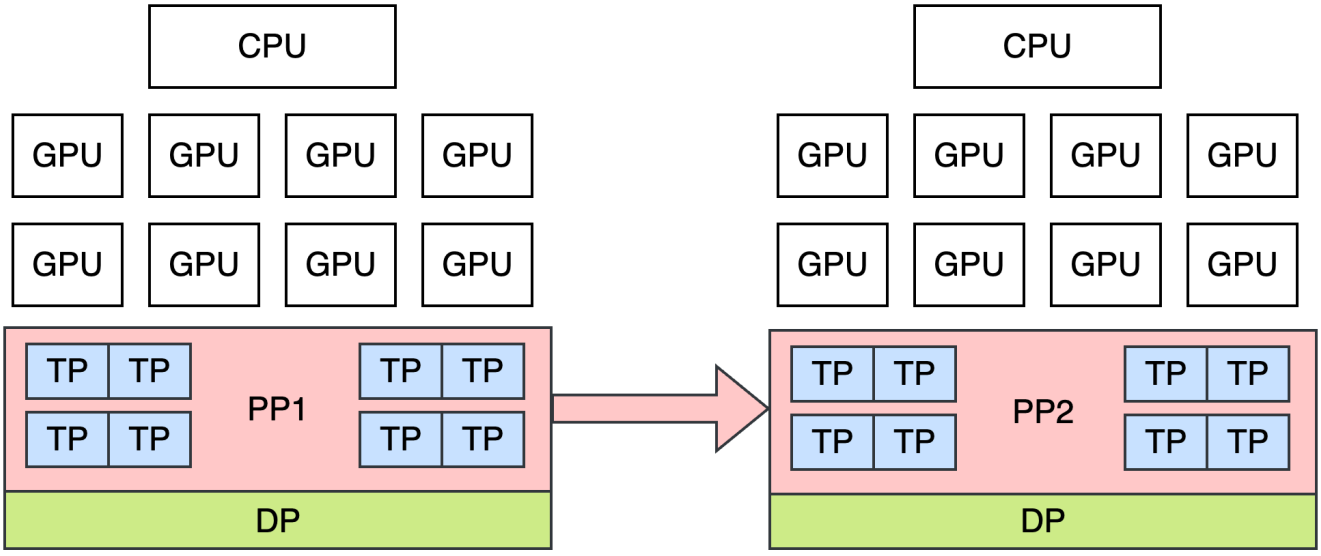
DP = 2 enables dual-node concurrency with minimal communication, doubling throughput while keeping TP/PP balanced inside each node.

**



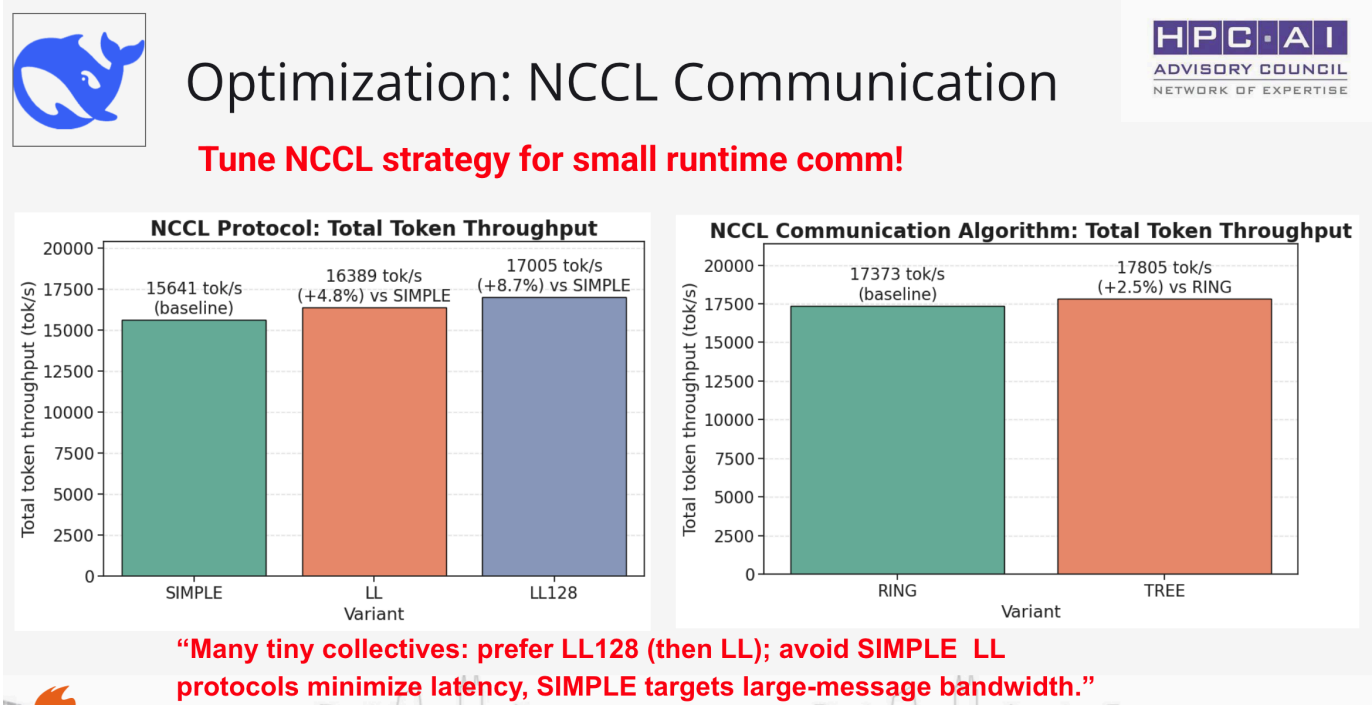
1. **Parallelism Strategy Exploration**
   We systematically tested combinations of (TP, PP, DP) to analyze the trade-off between inter- and intra-node communication.
   The configuration **TP = 8, PP = 2** achieved the best balance — using intra-node tensor parallelism to minimize cross-node data transfer while leveraging pipeline parallelism for memory scalability.





2. NCCL Communication

We tuned NCCL parameters and communication strategies to better suit the small-message, high-frequency workload pattern.
Tree-based algorithms replaced ring algorithms for latency-sensitive collectives, and the **LL128 protocol** was adopted for small transfers.

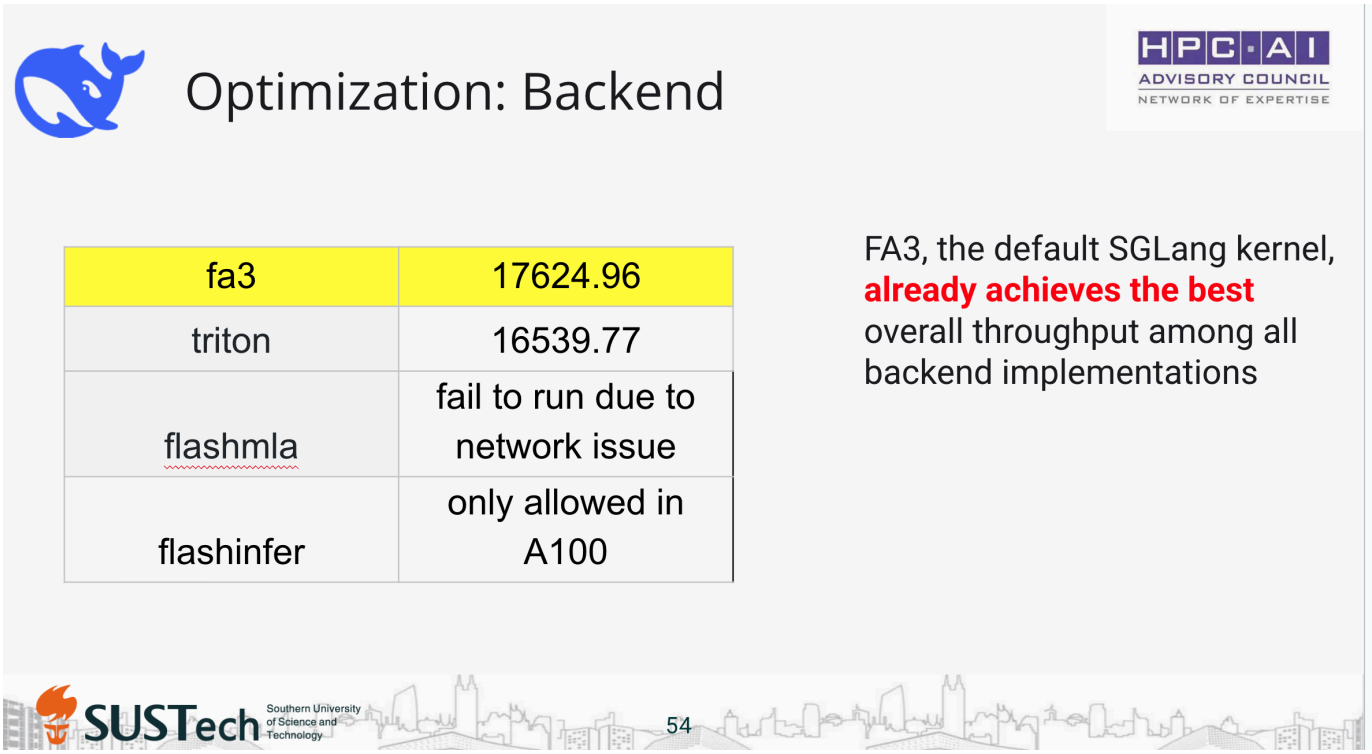


3. backend

**

FA3, the default SGLang kernel, already achieves the best overall throughput among all backend implementations

**

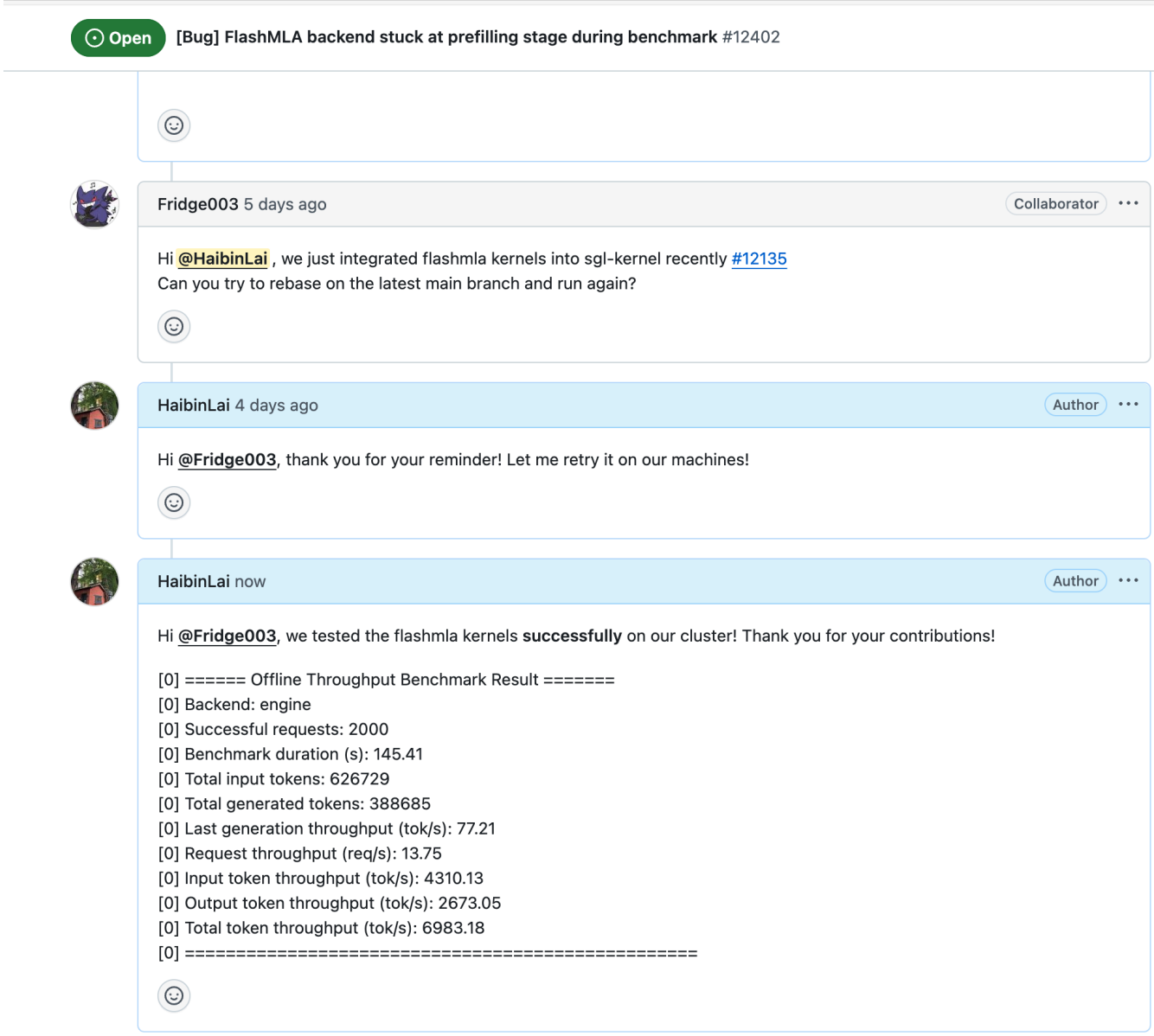


During the competition, we initially encountered difficulties running **FlashInfer** within the SGLang framework. Profiling showed communication stalls that prevented correct execution under certain (TP, PP, DP) configurations. After carefully reproducing and analyzing the issue, we reported our findings to the **SGLang official development team** through a detailed GitHub issue, as also suggested by the judges.

Subsequent investigation by the SGLang developers confirmed that the problem originated from an **in-development communication bug** in the FlashInfer backend. The issue has since been **officially fixed** in the latest SGLang release. We re-tested the workload on the **two-node, 16-GPU H200 cluster**, and **FlashInfer now runs successfully** with stable throughput and correct outputs.

issue https://github.com/sgl-project/sglang/issues/12402

This collaboration not only helped us complete our benchmark successfully but also contributed to improving the SGLang ecosystem. We are grateful to the organizers and the SGLang team for their timely support.



3. Speculative Decoding

**

Speculative decoding accelerates language model inference by using a smaller draft model to predict several tokens ahead, which the main model then verifies in parallel. To further improve throughput in the decode stage, we implemented **speculative decoding**, which generates multiple tokens concurrently using a lightweight draft model.

**

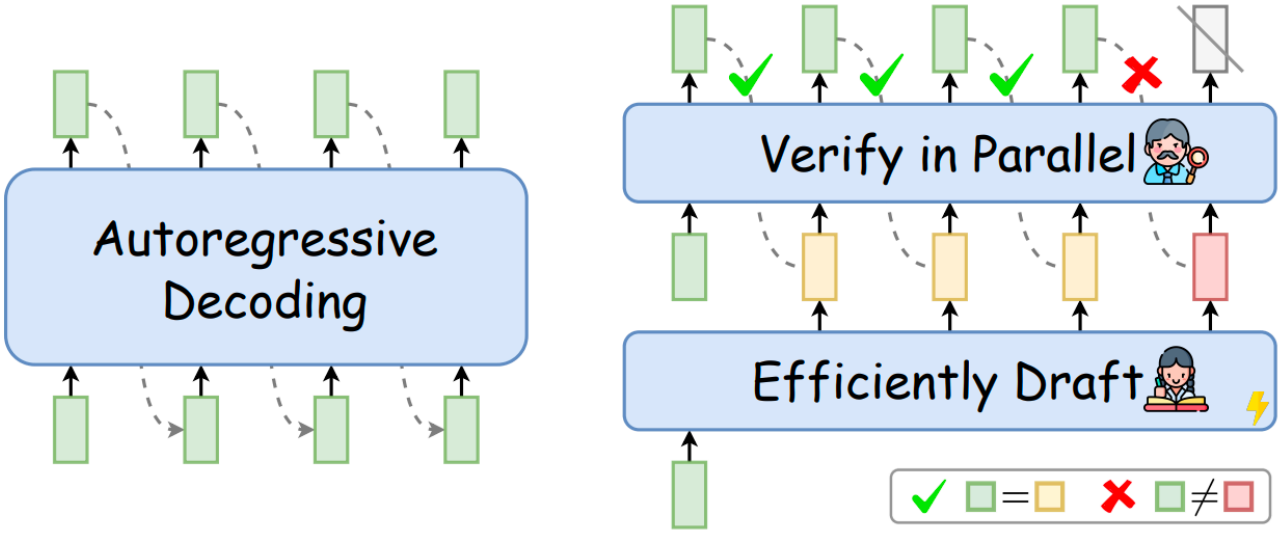
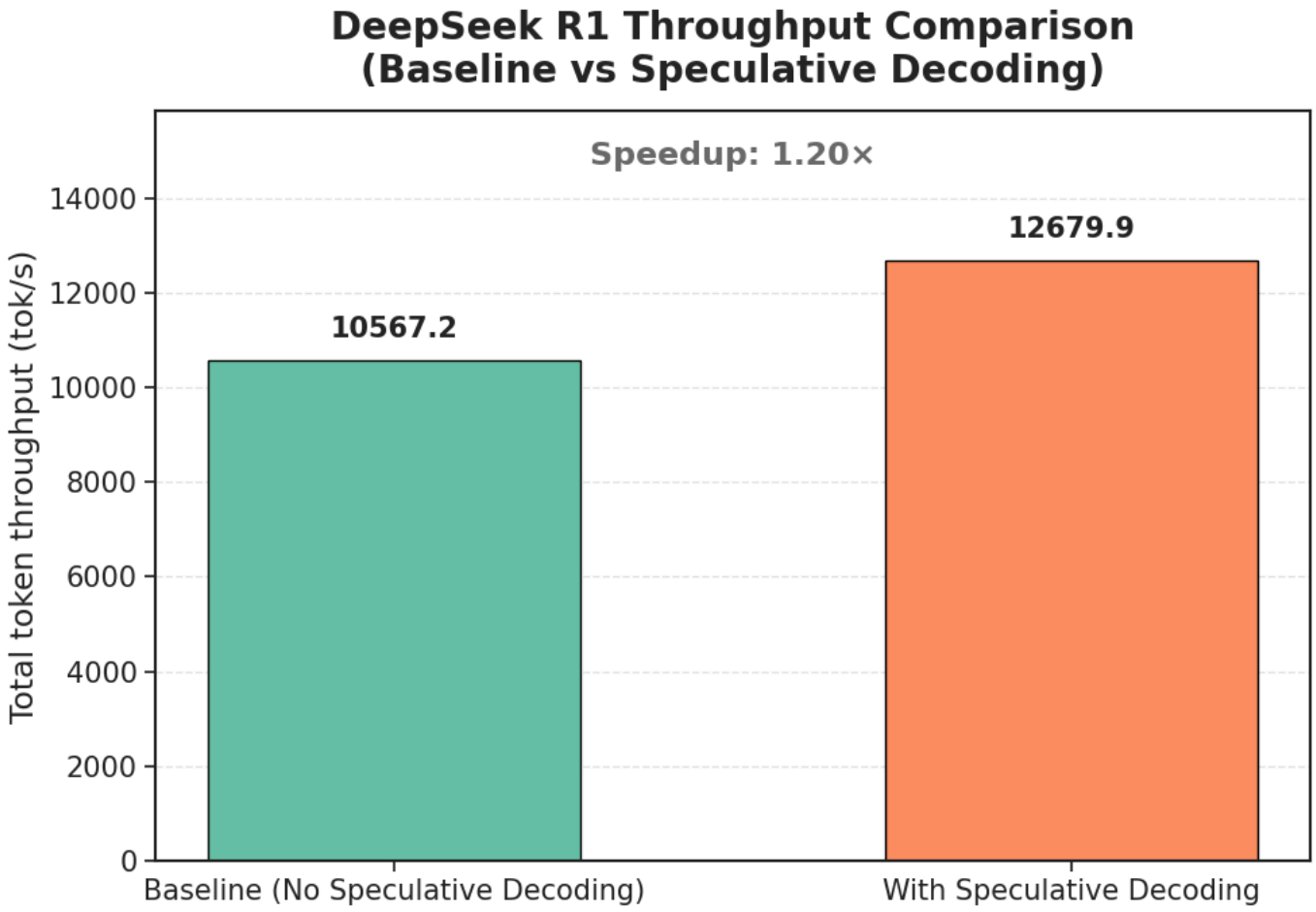


Figure 1: In contrast to autoregressive decoding (*left*) that generates sequentially, Speculative Decoding (*right*) first *efficiently drafts* multiple tokens and then *verifies* them *in parallel* using the target LLM. Drafted tokens after the bifurcation position (*e.g.*,) will be discarded to guarantee the generation quality.



But Speculative Decoding is not allowed with DP Parallelism!

Abort since not as much gain as DP

Summary of Results

After applying these optimizations, we observed:

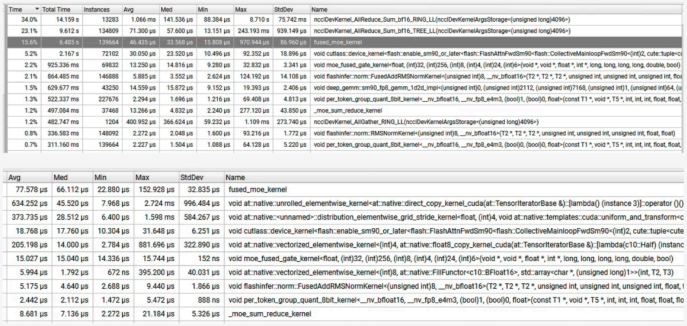
- **Reduced NCCL latency** and improved GPU utilization,
- **Better overlap** between compute and communication phases,
- **Higher decoding throughput**, especially when speculative decoding was enabled, and
- **Improved stability** across different backend configurations.

Overall, our optimized setup achieved **significant end-to-end speedups** over the baseline, demonstrating that **communication-aware parallelism** and **kernel-level backend tuning** are essential for scaling LLM inference efficiently on multi-GPU, multi-node systems.

Deepseek-R1 Conclusion

Token Throughput:
10772 -> 17805

+65.2% **TP=8, PP=2, DP=2**



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Name
34.0%	14.159 s	13283	1.066 ms	141.536 µs	88.384 µs	8.710 s	75.742 ms	ncclDevKernel_AllReduce_Sum_bf16_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
23.1%	9.612 s	134809	71.300 µs	57.600 µs	13.151 µs	243.193 ms	939.149 µs	ncclDevKernel_AllReduce_Sum_bf16_TREE_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
15.8%	6.603 s	139664	47.270 µs	23.968 µs	15.968 µs	972.944 µs	98.892 µs	fused_moe_kernel
5.2%	2.167 s	72102	30.050 µs	23.520 µs	16.496 µs	92.352 µs	18.896 µs	void cutlass::device_kernel<flash::enable_sm90_or_later<flash::FlashAttnFwdSm90<flash::CollectiveMainloopFwdSm90<(int)2, cute::tuple<c
2.2%	925.336 ms	69632	13.250 µs	14.816 µs	9.280 µs	32.832 µs	3.341 µs	void moe_fused_gate_kernel<float, (int)32, (int)256, (int)8, (int)4, (int)24, (int)6>(void *, void *, float *, int *, long, long, long, long, double, bool)
2.1%	864.485 ms	146888	5.885 µs	3.552 µs	2.624 µs	124.192 µs	14.108 µs	void flashinfer::norm::FusedAddRMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, T2 *, unsigned int, unsigned int, unsigned int, flo
1.5%	629.677 ms	43250	14.559 µs	15.872 µs	9.152 µs	19.393 µs	2.406 µs	void deep_gemm::sm90_fp8_gemm_1d2d_impl<(unsigned int)0, (unsigned int)2112, (unsigned int)7168, (unsigned int)1, (unsigned int)64, (u
1.3%	522.337 ms	227676	2.294 µs	1.696 µs	1.216 µs	69.408 µs	4.813 µs	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)1, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float,
1.2%	497.084 ms	37468	13.266 µs	4.832 µs	2.240 µs	277.120 µs	43.830 µs	_moe_sum_reduce_kernel
1.2%	482.747 ms	1204	400.952 µs	366.624 µs	59.232 µs	1.109 ms	273.740 µs	ncclDevKernel_AllGather_RING_LL(ncclDevKernelArgsStorage<(unsigned long)4096>)
0.8%	336.583 ms	148092	2.272 µs	2.048 µs	1.600 µs	93.216 µs	1.772 µs	void flashinfer::norm::RMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, T2 *, unsigned int, unsigned int, float, float)
0.7%	311.160 ms	139664	2.227 µs	1.504 µs	1.088 µs	64.128 µs	5.220 µs	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)0, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float,

Avg	Med	Min	Max	StdDev	Name
77.578 µs	66.112 µs	22.880 µs	152.928 µs	32.835 µs	fused_moe_kernel
634.252 µs	45.520 µs	7.968 µs	2.724 ms	996.484 µs	void at::native::unrolled_elementwise_kernel<at::native::direct_copy_kernel_cuda(at::TensorIteratorBase &)::[lambda() (instance 3)]::operator ()()
373.735 µs	28.512 µs	6.400 µs	1.598 ms	584.267 µs	void at::native::<unnamed>::distribution_elementwise_grid_stride_kernel<float, (int)4, void at::native::templates::cuda::uniform_and_transform<c
18.768 µs	17.760 µs	10.304 µs	31.648 µs	6.251 µs	void cutlass::device_kernel<flash::enable_sm90_or_later<flash::FlashAttnFwdSm90<flash::CollectiveMainloopFwdSm90<(int)2, cute::tuple<cute
205.198 µs	14.000 µs	2.784 µs	881.696 µs	322.890 µs	void at::native::vectorized_elementwise_kernel<(int)4, at::native::float8_copy_kernel_cuda(at::TensorIteratorBase &)::[lambda(c10::Half) (instanc
15.027 µs	15.040 µs	14.336 µs	15.744 µs	152 ns	void moe_fused_gate_kernel<float, (int)32, (int)256, (int)8, (int)4, (int)24, (int)6>(void *, void *, float *, int *, long, long, long, long, double, bool)
5.994 µs	1.792 µs	672 ns	395.200 µs	40.031 µs	void at::native::vectorized_elementwise_kernel<(int)8, at::native::FillFunctor<c10::BFloat16>, std::array<char *, (unsigned long)1>>(int, T2, T3)
5.175 µs	4.640 µs	2.688 µs	9.440 µs	1.866 µs	void flashinfer::norm::FusedAddRMSNormKernel<(unsigned int)8, __nv_bfloat16>(T2 *, T2 *, unsigned int, unsigned int, unsigned int, float, f
2.442 µs	2.112 µs	1.472 µs	5.472 µs	888 ns	void per_token_group_quant_8bit_kernel<__nv_bfloat16, __nv_fp8_e4m3, (bool)1, (bool)0, float>(const T1 *, void *, T5 *, int, int, int, float, float, flo
8.681 µs	7.136 µs	2.272 µs	21.184 µs	5.326 µs	_moe_sum_reduce_kernel

+10% **Tree, LL128**

Allreduce_Sum avg time:
1.553ms -> 1.066ms

Communication time drops, CUDA Graph launches drops

Suggestions for Running Deepseek R1

Notice for Users

- **Select TP, PP, DP suits for your hardware topology**
- **EP/PD needs large amounts GPU**
- Different backend takes costs to run
- Select mpi like hpcx

Suggestion for Developers

- **PP Support for Speculative Decoding**
- More profiling docs