

Unlocking Performance: Task Parallelism, SME Microkernels, and Exascale Execution

Shiting Long
Supervisor: Dirk Pleiter (KTH), Andreas Frommer (BUW)
October 2, 2026 — KTH Royal Institute of Technology



Task-Based Programming

Ways to Express Parallelism on Shared Memory Systems

Data Decomposition

Divide the data into independent portions.

Data $\longrightarrow$ Data_1 Data_2 … Data_n

Each thread do work on one portion.

Work $\longrightarrow$ Work_1 Work_2 … Work_n

e.g., `#pragma omp parallel for`

Task Decomposition

Divide the work into tasks.

Work $\longrightarrow$ T_1 T_2 … T_m

Tasks may have different sizes and dependencies.



Introduction to Task-Based Programming

- **Task**
  - A *task* is defined as a fine-grained, logical unit of work encompassing a specific function and its associated data set.
- **Task Scheduling**
  - A scheduler manages the assignment of tasks to execute on a selected thread in a pre-allocated thread pool.
- **Synchronization**
  - Synchronization is managed through data dependencies. A task is only eligible for execution once its input dependencies are satisfied.



Task-Based Programming Models

Different programming models provide support for **task-based parallelism**, e.g.:

- **OpenMP Tasks**
  - Task-based constructs provided by the OpenMP standard (>= 3.0).
- **OmpSs-2**
  - Task-based programming model developed by Barcelona Supercomputing Center (BSC), with explicit data dependencies and runtime scheduling.
- **HPX**
  - C++-based asynchronous runtime for task-based and distributed parallelism.

Application: Block LU Decomposition

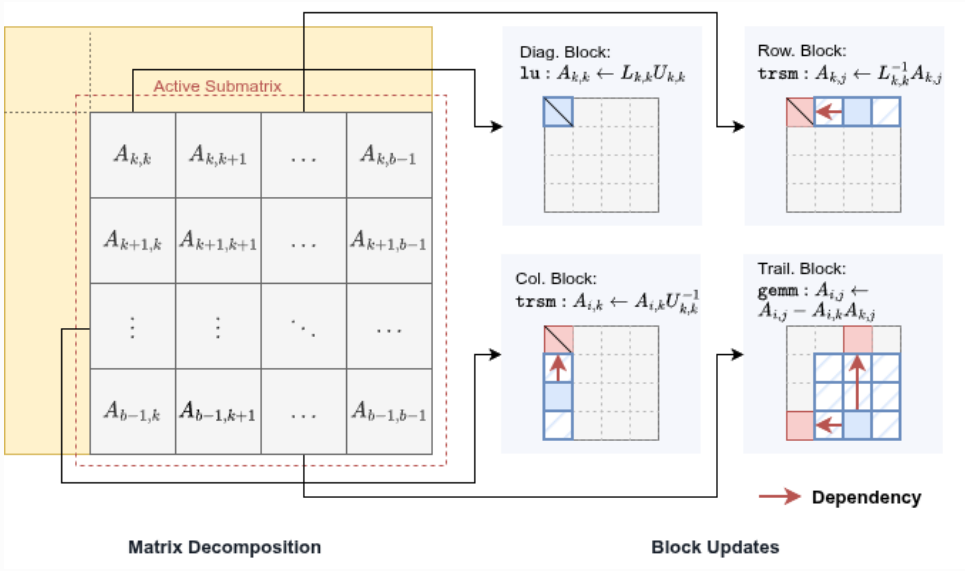


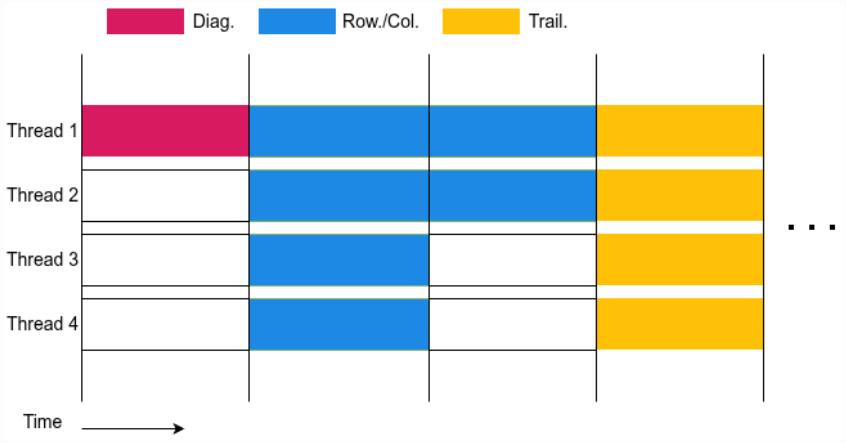
Figure: Block LU decomposition, demonstration.

Task-based implementation (OpenMP)

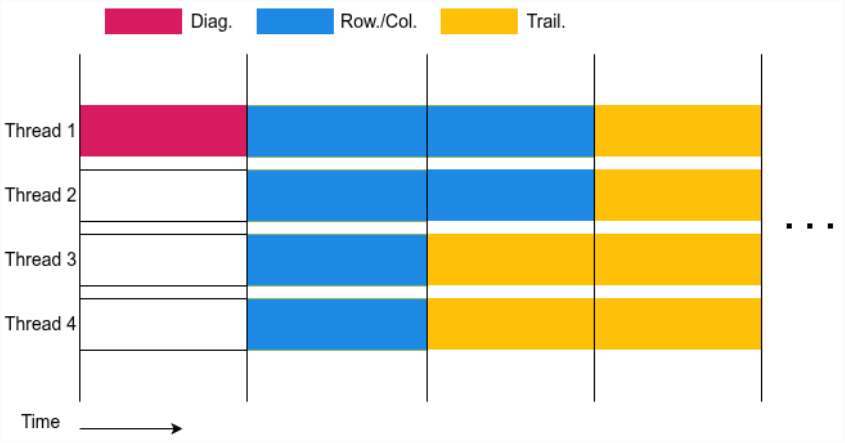
```
#pragma omp task  depend(inout:A_kk)
```
$$A_{kk} \leftarrow L_{kk}U_{kk}$$
```
#pragma omp task  depend(inout:A_kj)
                  depend(in:A_kk)
```
$$A_{kj} \leftarrow L_{kk}^{-1}A_{kj}$$
```
#pragma omp task  depend(inout:A_ik)
                  depend(in:A_kk)
```
$$A_{ik} \leftarrow A_{ik}U_{kk}^{-1}$$
```
#pragma omp task  depend(inout:A_ij)
                  depend(in:A_ik,A_kj)
```
$$A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$$

Block LU Decomposition: Parallel for vs. Task

Execution Graph



Parallel for



Task

Task runtime can adapt to **changing levels of available parallelism**.

Block LU Decomposition: Performance

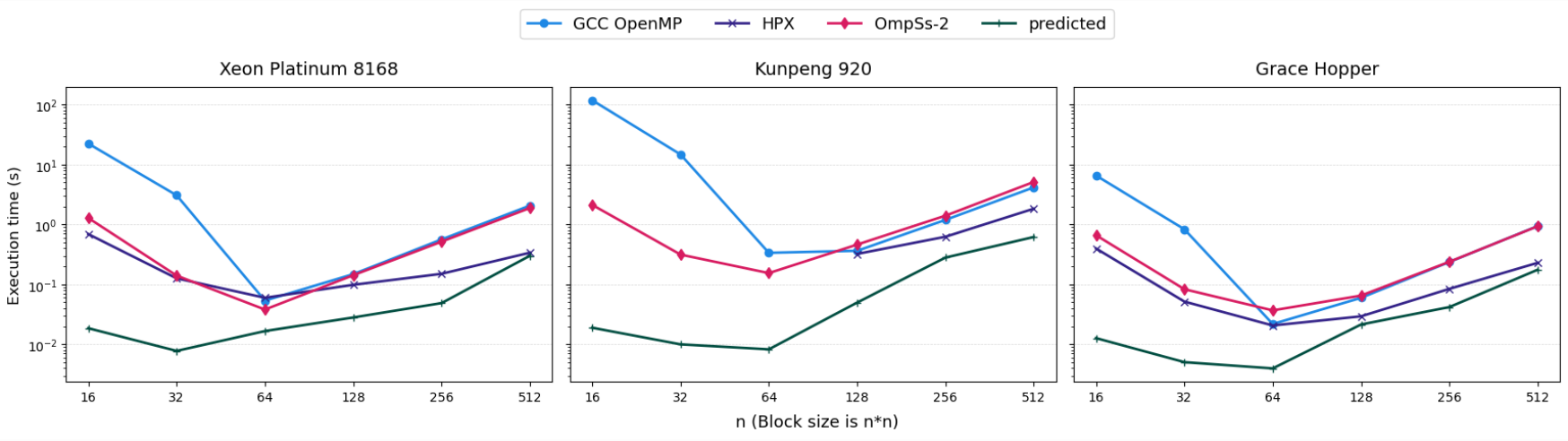
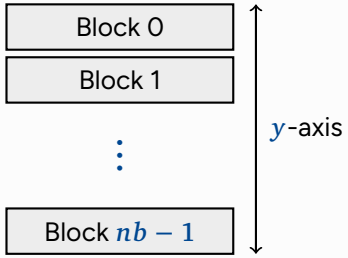


Figure: Performance of the LU benchmark with varying n (block size), matrix dimension size $N = 2048$.

Application: 2D Stencil

- Red-Black Gauss-Seidel on 2D Poisson
- **Implementation Strategy**
  - **Data Decomposition:**



- **Task Definition**
  Each update (Red or Black) per block is a task.
- **Dependencies**
  Tasks of a color depend on the data of the other color on adjacent blocks.

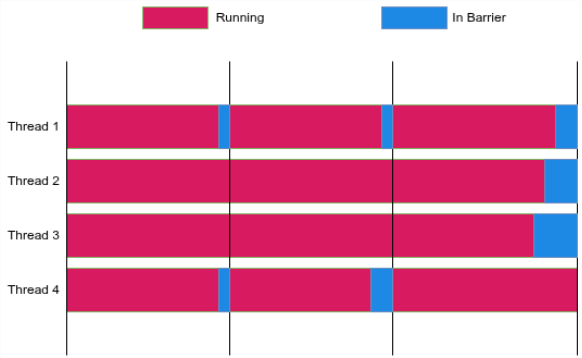
Long et al., *arXiv*, 2026. DOI: 10.48550/arXiv.2607.01735

2D Stencil: Barrier Stall

- **Varying latencies occur during execution, forcing threads to wait at barriers.**
- **Causes of Latencies**
  - **OS Noise:** Interrupts, context switches, or background processes on any node.
  - **Resource Contention:** Multiple threads/ranks competing for shared network interfaces, memory bandwidth, or CPU cycles.



Ideal Case



Practical Case

A similar effect can occur in task-based programming: inefficient task scheduling can introduce overhead and idle time.

2D Stencil: Performance

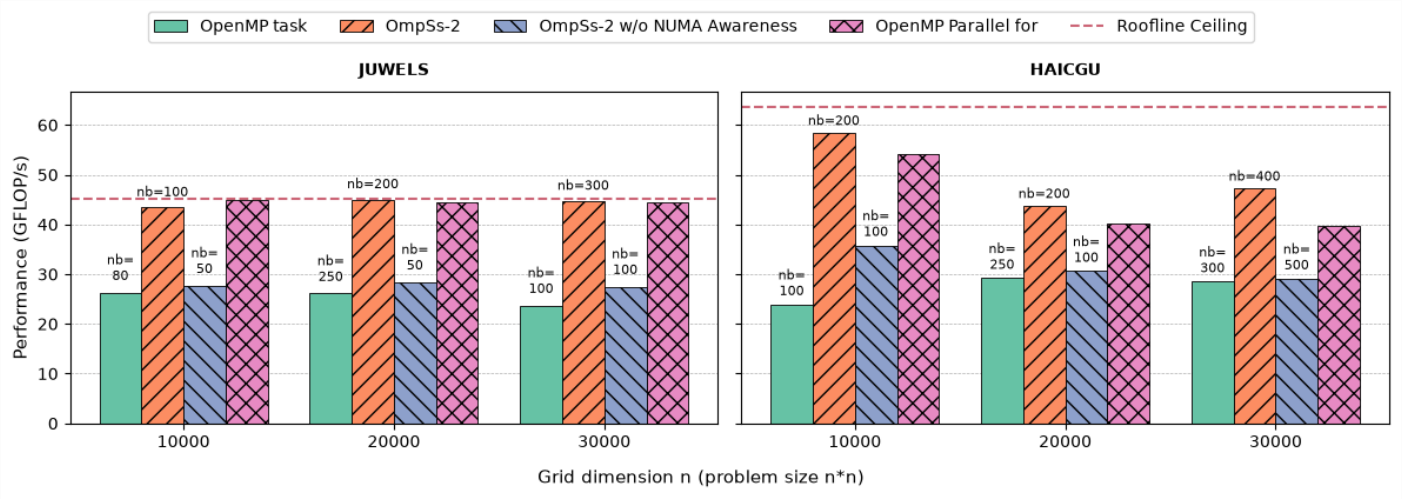


Figure: Performance with nb scaling on JUWELS and HAICGU. nb is chosen such that it is evenly divisible by the matrix dimension n.



Instruction Level Parallelism



SIMD Instructions

- **SIMD:** Single Instruction, Multiple Data
- SIMD instructions perform the same operation on multiple elements of a **vector** simultaneously.
- Efficient use of SIMD requires a suitable data layout.
- Example: **Arm SVE**.

Matrix Instructions

- Matrix intructions implies an instruction set that supports matrix multiplication.
- Example: **Arm SME**.
  - An outer product engine:

$$\underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}}_{z_0} \underbrace{\begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix}}_{z_1^T} = \underbrace{\begin{bmatrix} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{bmatrix}}_{ZA_0}$$

  - A 2D register array ZA for accumulation.
  - $ZA_0 \mathrel{+}= z_0 z_1^T$.
- SME is beneficial for kernels in need for outer product operations:
  - **GEMM:** $C \leftarrow AB + C$, where $A \in \mathbb{R}^{m \times k}$, $B \in \mathbb{R}^{k \times n}$, and $C \in \mathbb{R}^{m \times n}$.



SME for LQCD Kernel

- SME can be applied to LQCD kernels when processing **multiple right-hand sides**.
- **Benchmark:** $A \cdot M$
  - A: 3×3 complex matrix representing an SU(3) gauge link.
  - M: $3 \times b$ complex matrix containing b right-hand sides.
- We use **instruction emulation** to evaluate the implementations.
  - SME can significantly reduce the number of instructions compared with SVE.

Long et al., *HiPC*, 2025. DOI: 10.1109/HiPC66333.2025.00038



Exploration of LineShine

LineShine: An Exascale System

- **#1 on the TOP500 June 2026**
- **2.198 Exaflop/s** HPL performance
- 22,680 compute nodes, and over 13 million processor cores
- Installed at the National Supercomputing Centre in Shenzhen (NSCCSZ)

Lu, Feng, and Fu, *Frontiers of Computer Science*, 2026. DOI: 10.1007/s11704-026-61591-w

LineShine: Single-Node Architecture

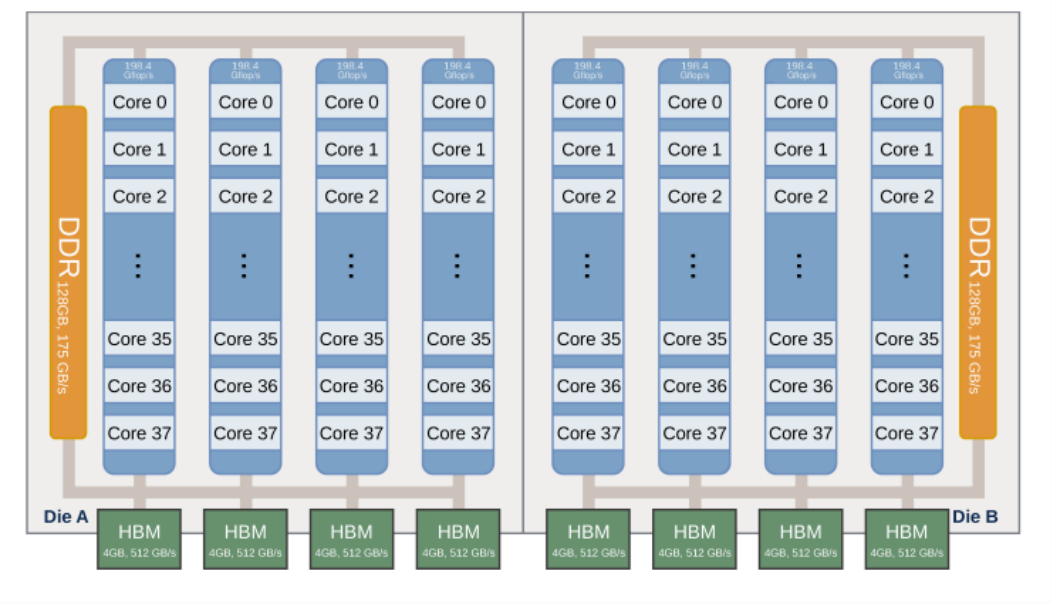


Figure: Architecture of the LX2 processor

Xu et al., *arXiv*, 2026. DOI: 10.48550/arXiv.2609.13115

LineShine: Single-Node Architecture Specs

Compute node

- $2\times$ Armv9-based **LX2** processors
- $2 \times 304 = 608$ physical cores
- SVE + SME
- FP64, FP32, FP16, and INT8 support

Per LX2 processor

- 8 NUMA domains
- 38 cores per NUMA domain
- Up to 60.3 TFLOP/s FP64

Memory hierarchy

- **HBM:** 32 GB per LX2, 4 TB/s
- **DDR5:** 256 GB per LX2, 8 channels

Challenges on LineShine

- Increasing the number of cores increases the demand on the memory system.
- Data moves through a hierarchy of **registers, caches, high-bandwidth memory (HBM), and main memory**, each with different capacities, bandwidths, and latencies.
- **Data locality** becomes critical: nearby data can be accessed much faster.
- **memory contention** and **NUMA effects** can limit scalability.

Achieving good performance requires both parallelism and efficient use of the memory hierarchy.

BLIS GEMM ($C \leftarrow \alpha AB + \beta C$)

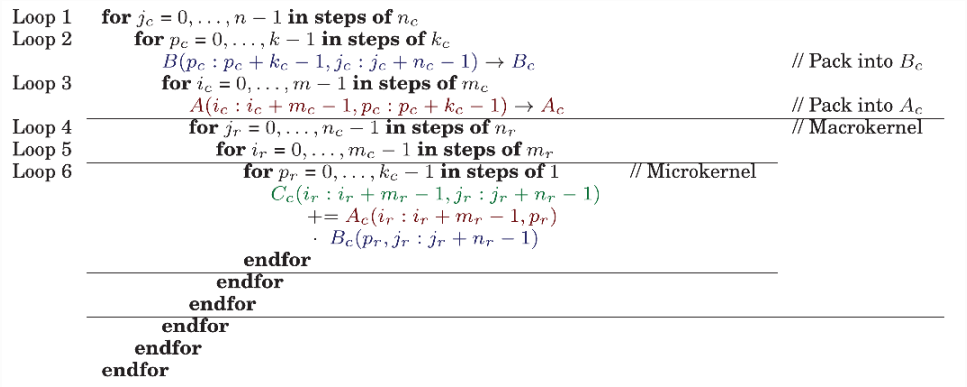


Figure: BLIS implementation of GEMM

Low et al., *ACM Transactions on Mathematical Software*, 2016.
DOI: 10.1145/2925987

What is BLIS?

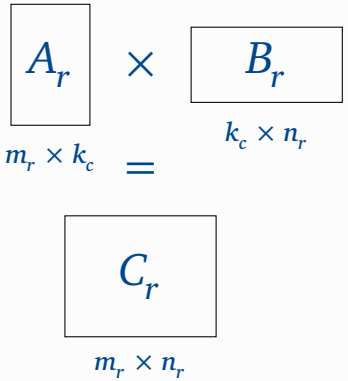
- A framework for implementing high-performance **BLAS** (Basic Linear Algebra Subprograms) operations.

Why multiple loops?

- Matrices are divided into smaller blocks that fit into different levels of the **memory hierarchy**.
- This improves **data reuse** and **cache locality**.

BLIS Microkernel

Microkernel



SVE

Different microkernels can use

- different m_r, n_r,
- different vectorization strategies.

These choices can affect overall performance.

SME

We implement a dedicated SME microkernel.

$$m_r = 16, \quad n_r = 32$$

Nassyr and Pleiter, *Euro-Par*, 2024. DOI: 10.1007/978-3-031-69766-1_4
Nassyr et al., *Euro-Par*, 2026. DOI: 10.1007/978-3-032-35248-4_9

Single-Core Microkernel Performance

SVE

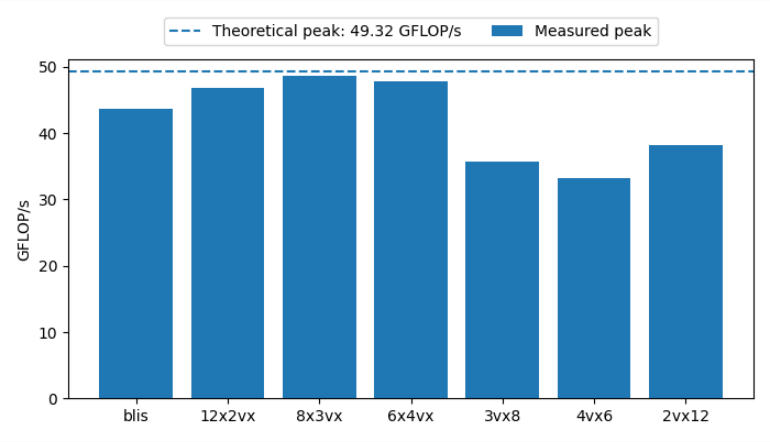


Figure: SVE Single-Core Microkernel Performance

SME

188.72 GFLOP/s

Theoretical peak:
198.36 GFLOP/s

95.1% of peak

Single-Core DGEMM Performance

Problem size: $m = k = n = 4000$

SVE

39.35 GFLOP/s

8x3vx microkernel

Close to the single-core peak.

SME

96.39 GFLOP/s

16×32 microkernel

Well below the single-core peak.

Acknowledgment

Acknowledgment

- Funding from the European Union
  - HORIZON MSCA programme AQTIVATE (Grant Agreement No. 101072344)
  - EoCoE-III project (Grant Agreement No. 101144014)
- Access to HPC systems
  - Jülich Supercomputing Centre for access to JUWELS and JURECA
  - Open Edge and HPC Initiative (OEHI) for access to HAICGU and LineShine
- Special thanks to NSCCSZ for hosting us on site


KTH
VETENSKAP
OCH KONST