Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs using Asynchronous Progress Control

堀越将司 [a)]，Balazs Gerofi [b)]，石川裕 [c)]，中島研吾 [b,d)]

(a) インテル，(b) 理化学研究所，(c) 国立情報学研究所，(d) 東京大学情報基盤センター

本稿では，2019 年 5 月，2021 年 10 月に実施された大規模 HPC チャレンジの結果を報告する。本大規模 HPC チャレンジの成果をまとめた論文「Masashi Horikoshi, Balazs Gerofi, Yutaka Ishikawa and Kengo Nakajima, Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs using Asynchronous Progress Control」は，2022 年 1 月にオンラインで開催された HPC Asia 2022 でのワークショップ IXPUG（Intel eXtreme Performance Users Group）[1] に採択され，Proceedings（ACM Digital Library）に掲載された。本稿は出版元である ACM の許可を受け次ページ以降に上記論文の全文を掲載するものである。

1) https://www.ixpug.org/events/ixpug-hpc-asia-2022

Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs using Asynchronous Progress Control

Masashi Horikoshi
Accelerated Computing Systems and Graphics (AXG) Group, Intel Corporation
Japan
Masashi.Horikoshi@intel.com

Balazs Gerofi
RIKEN Center for Computational Science (R-CCS)
Japan
bgerofi@riken.jp

Yutaka Ishikawa
National Institute of Informatics
Japan
yutaka_ishikawa@nii.ac.jp

Kengo Nakajima
Information Technology Center, The University of Tokyo/ RIKEN Center for Computational Science (R-CCS)
Japan
nakajima@cc.u-tokyo.ac.jp

ABSTRACT

Preconditioned parallel solvers based on the Krylov iterative method are widely used in scientific and engineering applications. Communication overhead is a critical issue when executing these solvers on large-scale massively parallel supercomputers. In this work, we investigate communication-computation overlapping by *asynchronous progress control* to various types of preconditioned conjugate gradient methods for parallel finite-element applications. Performance of the developed method is evaluated using up to 4,096 nodes of the Oakforest-PACS system at JCAHPC, equipped with Intel® Xeon Phi™ Manycore Processors. We show that the performance of the iterative solver can be improved by up to 38% on 4,096 nodes. We apply the IHK/McKernel lightweight multi-kernel operating system and find that it can provide 20% and 6% improvements on 2,048 and 4,096 node respectively. Furthermore, we investigate the effects of asynchronous communication progression on the IHK/McKernel and find that it can provide an 2-3% improvement from 128 node to 1,024 node.

CCS CONCEPTS

• **Networks** → **Network performance analysis**; **Network measurement**; • **Applied computing** → **Earth and atmospheric sciences**; **Engineering**.

KEYWORDS

iterative solver, MPI, non-blocking communication, asynchronous progress threads, lightweight kernel

HPCAsia 2022 Workshop, January 11–14, 2022, Virtual Event, Japan

ACM ISBN 978-1-4503-9564-9/22/01…$15.00
https://doi.org/10.1145/3503470.3503474

ACM Reference Format:
Masashi Horikoshi, Balazs Gerofi, Yutaka Ishikawa, and Kengo Nakajima. 2022. Exploring Communication-Computation Overlap in Parallel Iterative Solvers on Manycore CPUs using Asynchronous Progress Control. In *International Conference on High Performance Computing in Asia-Pacific Region Workshops (HPCAsia 2022 Workshop), January 11–14, 2022, Virtual Event, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3503470.3503474

1 INTRODUCTION

Preconditioned parallel solvers based on the Krylov iterative method are widely used in scientific and engineering applications. Communication overhead is a critical issue when executing these solvers on large-scale massively parallel supercomputers. In the present work, we introduced communication-computation overlapping by *asynchronous progress control* [1], which is supported by Intel® MPI Library from version 2019, to various types of preconditioned Krylov iterative methods, such as *pipelined* conjugate gradient method [2]. We focus on optimization of global collective communications for dot products in parallel Krylov iterative solvers. Target linear equations are derived from GeoFEM/Cube [3], which is a parallel finite-element application on massively parallel supercomputers. Performance of the developed method was evaluated using up to 4,096 nodes of the Oakforest-PACS (OFP) system at JCAHPC with Intel Xeon Phi Manycore Processors[4], and the performance of the iterative solver was evaluated.

In the previous works [5][6], we examined the impact of the IHK/McKernel [10], lightweight multi-kernel OS developed at RIKEN R-CCS, which provides efficient and scalable execution environment on large-scale systems by reducing OS noise and communication overhead. Improvement of the performance of parallel multigrid solvers was 10-20% at 2,048 nodes of OFP. In the present work, effects of IHK/McKernal were also evaluated for Asynchronous Progress Control.

The rest of this paper is organized as follows. Section refsect:appl provides an overview of the target application and algorithms. Section 3 provides a brief summary of the target hardware system.

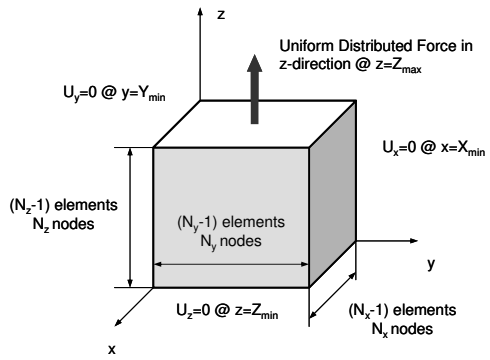


Figure 1: Simple cube geometry for 3D linear elasticity problems in GeoFEM/Cube [3]

Section 4 overviews Intel's Asynchronous Progress Control. Section 5 presents overview of IHK/McKernel, and special configuration of threads for Intel's Asynchronous Progress Control. Section 6 describes numerical experiments and results. Finally, Section 8 concludes the paper and offers future perspective.

2 TARGET APPLICATION AND ALGORITHMS

2.1 GeoFEM/Cube

GeoFEM/Cube [3] is the target application, and it solves 3D linear elasticity problems in simple cube geometries using a parallel FEM. Tri-linear hexahedral elements are used for the discretization. Material properties are defined as homogeneous, where Poisson's ratio is set to 0.30 for all elements and Young's modulus is 1.00. The boundary conditions are described in Fig 1. Large-scale linear equations with sparse matrices derived from the application are solved by preconditioned Krylov iterative methods . The original GeoFEM/Cube adopts the incomplete Cholesky conjugate gradient (ICCG) method [11] with preconditioning by incomplete Cholesky factorization without fill-ins (IC(0)) [11]. The additive Schwarz domain decomposition (ASDD) for overlapped regions [12] is introduced for stabilization of parallel IC(0) with block-Jacobi-type localization. The GeoFEM/Cube application is parallelized by domain decomposition using the Message-Passing Interface (MPI) [3]. In the OpenMP/MPI hybrid parallel programming model, multithreading by OpenMP is applied to each partitioned domain. GeoFEM/Cube is written in Fortran with MPI and OpenMP.

In GeoFEM/Cube, the entire model is divided into domains, and each domain is assigned to an MPI process, as shown in Fig.2. The local data structure in GeoFEM/Cube is node-based with overlapping elements, and as such is appropriate for the preconditioned iterative solvers used in GeoFEM/Cube (Fig.3).

GeoFEM/Cube generates distributed local meshes and coefficient matrices in a parallel manner using MPI. In GeoFEM/Cube, each MPI process has $(Nx/Px) \times (Ny/Py) \times (Nz/Pz)$ vertices, while the total number of vertices in each direction is (Nx, Ny, Nz) and the number of partitions in each direction is (Px, Py, Pz).

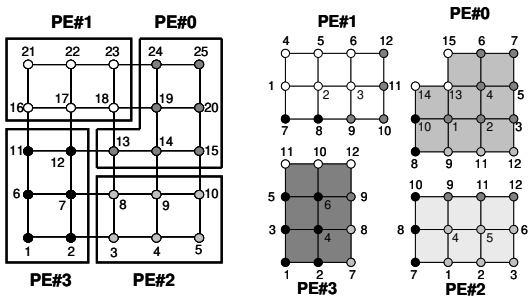


Figure 2: Node-based partitioning in GeoFEM/Cube [3]

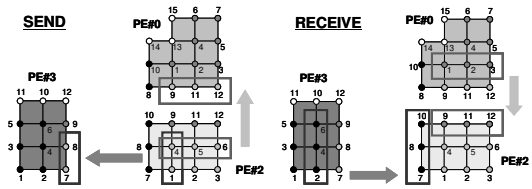


Figure 3: Communications among MPI processes [3]

2.2 Pipelined CG

Algorithm 1 shows the preconditioned conjugate gradient method (PCG) [11], which includes SpMV, dot products, the DAXPY computations, and preconditioning.

Algorithm 1 Preconditioned Conjugate Gradient Method (PCG) [11]

1: $r^{(0)} = b - Ax^{(0)}$; $u^{(0)} = M^{-1}r^{(0)}$; $p^{(0)} = u^{(0)}$; $\rho_0 = (r^{(0)}, u^{(0)})$
2: **for** $k = 0, 1, \ldots,$ until convergence **do:**
3: $q^{(k)} = Ap^{(k)}$
4: $\alpha_k = \rho_k / (p^{(k)}, q^{(k)})$;
5: $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
6: $r^{(k+1)} = r^{(k)} - \alpha_k q^{(k)} \Rightarrow check\ convergence: |r^{(k+1)}|$
7: $u^{(k+1)} = M^{-1} r^{(k+1)}$
8: $\rho_{k+1} = (r^{(k+1)}, u^{(k+1)})$
9: $\beta_{k+1} = \rho_{k+1} / \rho_k$
10: $p^{(k+1)} = u^{(k+1)} + \beta_{k+1} p^{(k)}$
11: **end for**

In computations on parallel computers with distributed memory, SpMV (Sparse Matrix-Vector Multiplication), dot products, and preconditioning may require communication. Although there are various types of communication patterns in preconditioning, SpMV relies upon point-to-point communication with neighbors and dot products rely upon global collective communication [3][11].

If the code is implemented by MPI, `MPI_Isend` and `MPI_Irecv` with `MPI_Wait/Waitall` are used in SpMV, and `MPI_Allreduce` is

used in dot products. The overhead for such communications is a critical issue on massively parallel supercomputers with more than 10^4 MPI processes.

In recent years, many algorithms and methods for avoiding and reducing communication overhead have been proposed. Methods based on the matrix powers kernel [13] reduce the number of global communications at each iteration by extending the halo region in Figs. 2 and 3. They generally require complicated data structures and are not suitable for general preconditioning methods, such as incomplete LU (ILU).

In [2], several methods for conjugate gradient methods, which reduce overhead of global collective communications for dot products, are introduced. The sequence of Krylov iterations is changed using recurrence relations without changing the original algorithm. Because the order of computation is changed, rounding errors are propagated differently. Therefore, convergence may be affected in ill-conditioned problems. Authors applied computing with single precision to such algorithms, and results show that original CG is the most robust [14]. This does not happen for the cases with double precision for ill-conditioned problems in the present work.

Algorithm 2 shows PCG using Chronopouos/Gear CG method [15], where 2 dot products are combined into a single reduction. Thus, overhead for calling `MPI_Allreduce` may be reduced.

Algorithm 2 Preconditioned Chronopoulos/Gear Conjugate Gradient Method [15]

1: $r^{(0)} = b - Ax^{(0)}$; $u^{(0)} = M^{-1}r^{(0)}$; $p^{(0)} = u^{(0)}$; $q^{(0)} = Ap^{(0)}$; $\gamma_0 = (r^{(0)}$
2: **for** $k = 0, 1, \ldots,$ until convergence **do:**
3: $p^{(k+1)} = u^{(k)} + \beta_k p^{(k)}$
4: $q^{(k+1)} = w^{(k)} + \beta_k q^{(k)}$
5: $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k+1)}$
6: $r^{(k+1)} = r^{(k)} - \alpha_k q^{(k+1)} \Rightarrow check\ convergence: |r^{(k+1)}|$
7: $u^{(k+1)} = M^{-1}r^{(k+1)}$
8: $w^{(k+1)} = Au^{(k+1)}$
9: $\gamma_{k+1} = (r^{(k+1)}, u^{(k+1)})$
10: $\delta_{k+1} = (w^{(k+1)}, u^{(k+1)})$
11: $\beta_{k+1} = \gamma_{k+1}/\gamma_k$
12: $\alpha_{k+1} = \gamma_{k+1}/(\delta_{k+1} - \beta_{k+1}\gamma_{k+1}/\alpha_k)$
13: **end for**

The pipelined method [2] reduces communications overhead by overlapping global collective communications and computations.

Algorithm 3 shows Preconditioned Pipeined CG method [2]. This method is derived from Algorithm 2.

In pipelined CG [2], collective communication for dot products can be overlapped with heavier computations, such as SpMV and preconditioning. In the original CG algorithm (Algorithm1), dot products ($\alpha_k = \rho_k/(p^{(k)}, q^{(k)})$ in line-4, and $\rho_{k+1} = (r^{(k+1)}, u^{(k+1)})$ in line-8) are used in the next lines (line-5 and line-9), while they are used after in SpMV and preconditioning in the pipelined algorithm. $\delta_k = (p^{(k)}, q^{(k)})$ is defined in line-3, and it it used in line-5, therefore computing of δ_k can be overlapped with $z^{(k)} = M^{-1}q^{(k)}$ in line-4. $\gamma_{k+1} = (r^{(k+1)}, u^{(k+1)})$ is defined in line-9, and it is used in line-11 after $w^{(k+1)} = Au^{(k+1)}$ in line-10.

Asynchronous collective communication (e.g., `MPI_Iallreduce`) supported in MPI-3 is effective for such procedures.

In [16], the authors implemented pipelined CG to the original GeoFEM/Cube with ICCG and ASDD using up to 12,288 cores (384 nodes) of the Reedbush-U system [17] with Intel® Xeon® E5-2695v4 (code name: Broadwell-EP) CPUs using Intel® MPI 2017. Fig. 4 compares the original and pipelined CG method for strong scaling. Pipelined CG provides much better scalability than the original CG. Pipelined CG is effective in the very limited case of strong scaling, where the problem size per MPI process is very small.

Algorithm 3 Preconditioned Pipelined Conjugate Gradient Method [2]

1: $r^{(0)} = b - Ax^{(0)}$; $u^{(0)} = M^{-1}r^{(0)}$; $p^{(0)} = u^{(0)}$; $q^{(k)} = Ap^{(k)}$; $w^{(0)} = Au^{(0)}$; $\alpha_0 = (r^{(0)}, u^{(0)})/(w^{(0)}, u^{(0)})$; $\beta_0 = 0$; $\gamma_0 = (r^{(0)}, u^{(0)})$
2: **for** $k = 0, 1, \ldots,$ until convergence **do:**
3: $\delta_k = (p^{(k)}, q^{(k)})$
4: $z^{(k)} = M^{-1}q^{(k)}$
5: $\alpha_k = \gamma_k/\delta_k$
6: $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
7: $r^{(k+1)} = r^{(k)} - \alpha_k q^{(k)} \Rightarrow check\ convergence: |r^{(k+1)}|$
8: $u^{(k+1)} = u^{(k)} - \alpha_k z^{(k)}$
9: $\gamma_{k+1} = (r^{(k+1)}, u^{(k+1)})$
10: $w^{(k+1)} = Au^{(k+1)}$
11: $\beta_{k+1} = \gamma_{k+1}/\gamma_k$
12: $p^{(k+1)} = u^{(k+1)} + \beta_{k+1}p^{(k)}$
13: $q^{(k+1)} = w^{(k+1)} + \beta_{k+1}q^{(k)}$
14: **end for**

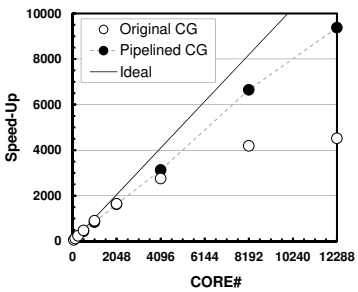


Figure 4: Effects of pipelined CG [16] for GeoFEM/Cube with ICCG and ASDD using up to 12,288 cores (384 nodes) of Reedbush-U [16] and strong scaling; total problem size: 28,311,552 DOF

Finally, Algorithm 4 shows Preconditioned Gropp's CG method [2][18]. While this method is similar to Algorithm 3, computations are smaller than those of Algorithm 3. Dot products in line-3 and line-4 (γ_k, δ_k) are overlapped with preconditioning in line-5 and SpMV in line-6.

In the present work, following four algorithms for the conjugate gradient method are evaluated. Each method includes one *SpMV*

(Sparse Matrix Vector Multiply), one preconditioning and three dot products for each iteration. Number of *DAXPY*'s (constant times a vector plus a vector (axpy) in double-precision) in each method as follows. 8 DAXPY's in a iteration of Pipelined CG (Alg.3), while it is 3 in the original CG (Alg.1):

- Algorithm 1: Original CG (3 DAXPY's in one iteration) [11]
- Algorithm 2: Chronopoulos/Gear CG (4 DAXPY's) [15]
- Algorithm 3: Pipelined CG (8 DAXPY's) [2]
- Algorithm 4: Gropp's CG (5 DAXPY's) [2][18]

Algorithm 4 Preconditioned Gropp's Conjugate Gradient Method [2][18]

$r^{(0)} = b - Ax^{(0)}$; $u^{(0)} = M^{-1}r^{(0)}$; $w^{(0)} = Au^{(0)}$; $z^{(0)} = q^{(0)} = s^{(0)} = p^{(0)} = 0$
for $k = 0, 1, \ldots,$ until convergence **do:**
  $\gamma_k = (r^{(k)}, u^{(k)})$
  $\delta_k = (w^{(k)}, u^{(k)})$
  $m^{(k)} = M^{-1}w^{(k)}$
  $n^{(k)} = Am^{(k)}$
  if $k > 0$ **then**
    $\beta_k = \gamma_k/\gamma_{k-1}$; $\alpha_k = \gamma_k/(\delta_k - \beta_k\gamma_k/\alpha_{k-1})$
  else
    $\beta_0 = 0$; $\alpha_k = \gamma_k/\delta_k$
  end if
  $z^{(k+1)} = n^{(k)} + \beta_k z^{(k)}$
  $s^{(k+1)} = m^{(k)} + \beta_k s^{(k)}$
  $q^{(k+1)} = w^{(k)} + \beta_k q^{(k)}$
  $p^{(k+1)} = u^{(k)} + \beta_k p^{(k)}$
  $x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}$
  $r^{(k+1)} = r^{(k)} - \alpha_k q^{(k)} \Rightarrow$ *check convergence* : $|r^{(k+1)}|$
  $u^{(k+1)} = u^{(k)} - \alpha_k s^{(k)}$
  $w^{(k+1)} = w^{(k)} - \alpha_k z^{(k)}$
end for

3 TARGET HARDWARE (OAKFOREST-PACS, OFP)

The Oakforest-PACS system (OFP) [4] is the premiere supercomputer system at the *Joint Center for Advanced High-Performance Computing (JCAHPC)* [4] , which was established by the University of Tokyo and University of Tsukuba. The system consists of 8,208 nodes of Intel Xeon Phi 7250 (code name: Knights Landing, or *KNL*), and *Intel® Omni-Path Architecture* (Intel® OPA) provides a 100 Gbps interconnection in a fat-tree topology with full bisection bandwidth. Each Xeon Phi 7250 node is built using 68 modified Atom® (code name: Silvermont) cores running at 1.4 GHz, and the memory unit consists of 96 GB of DDR4 RAM and 16 GB of stacked 3D MCDRAM, which can be utilized as an L3 cache or high-bandwidth memory. Each core has two 512-bit vector units and supports AVX-512 SIMD instructions. Each core can host four threads (*i.e.,* 272 overall logical CPUs on the entire chip) and is equipped with 2 512-bit floating-point vector ALU. The total theoretical computational performance is 25 PFLOPS, and the system achieved 13.55 PFLOPS on the HPL benchmark. In the present work, only MCDRAM was used for memory in the flat/quadrant mode on the OFP. Intel's compiler and MPI library (2018) were used. Table 1 summarizes the specifications of each node of OFP.

Table 1: Summary of the performance of single node of the Oakforest-PACS(OFP)

Architecture	Intel Xeon Phi 7250 (Knights Landing)
Frequency(GHz)	1.40
Core # /CPU (socket) (Maximum Effective Thread #)	68 (272)
Peak Performance (GFLOPS)	3,046.4
Memory Size (GB)	MCDRAM:16,DDR4:96
Memory Bandwidth (GB/sec)	MCDRAM:490 [20], DDR4:84
Compiler & MPI Library	Intel® Parallel Studio 2019 XE, Intel MPI 2019

4 ASYNCHRONOUS PROGRESS CONTROL

4.1 Overview

MPI Non-blocking communication in application thread (e.g. for `MPI_Isend`) is expected to run asynchronously. To do that, progress thread is required if MPI communication is not completely offloaded. But turning on the progress thread (e.g. for `MPI_test`) requires `MPI_THREAD_MULTIPLE`. But in most case, MPI progress thread is not practically used due to poor MPI performance due to serialization around one queue for all threads, thread switching overhead, and process-wide MPI objects management overhead. Because those restrictions are closely related with MPI 3.1 standard (defined as "threads are not separately addressable: a rank in a send or receive call identifies a process, not a thread"). Those history and practices are well summarized in [21]. To overcome such restrictions, Intel MPI 2019 developed new asynchronous progress design, which is based on MPICH asynchronous design. From Intel MPI version 2019, it supports the new asynchronous progress threads that users to manage communication in parallel with application's computation to achieve better communication and computation overlapping. Asynchronous progress control on the Intel MPI has a full support for MPI point-to-point operations, blocking collectives, and a partial support for non-blocking collectives (`MPI_Ibcast`, `MPI_Ireduce`, and `MPI_Iallreduce`).

Setting the `I_MPI_ASYNC_PROGRESS_PIN` environment variable on the Intel MPI allows to control a pinning of the asynchronous progress threads to logical processor cores. By using this feature [1], we can provide separate dedicated cores for asynchronous progress threads.

4.2 Core Configuration

By default, Intel MPI allocates cpu core resource to the asynchronous progress threads from last logical core. For the 68 cores Intel

[1]To enable asynchronous progress control on the Intel MPI, environment variable I_MPI_ASYNC_PROGRESS=on, after loading release_mt environment, is required.

Xeon Phi processor, it has 272 logical cores so the default pinning values are from 271 to 264 in the case of 8 MPI processes and 8 asynchronous progress threads, for example. It depends on application but there are several ways to allocate logical cores for asynchronous progress threads on the many core processor. For simplicity on the GeoFEM application, we fixed the number of MPI process as 8 and used 8 OpenMP threads per MPI process. Total 64 logical core by using same number of physical core for better GeoFEM performance (1 hardware thread per physical core used).

We exclude first two physical and last two physical cores of the Xeon Phi from GeoFEM computation by using environment variable `I_MPI_PIN_PROCESSOR_EXCLUDE_LIST=0,1,68,69,136,137,204,205` in order to avoid first core (logical cores 0,68,136, and 204) which is affected by OS noise and jitters. Logical cores from 2 to 65 are used for GeoFEM computation. Fig. 5 illustrates such situation as experimental setup. We have tested 5 different mapping of 8 asynchronous progress threads (1 progress thread per MPI process) in addition to the case with asynchronous progress control. First method (Fig. 6) is using default setup of Intel MPI which uses from logical core 268 to 271. Second method (Fig. 7) is using first two physical cores (logical core 0,1,68,69,136,137,204 and 205). Since first two cores are not used by computation, we can allocate 8 logical cores by 2 free physical cores.Third method (Fig. 8) is using last two physical cores (66,67,134,135,202,203,270 and 271). This method is basically same as second method but it is not affected by OS noise and jitters. Forth method (Fig. 9) is using first two and last two cores (0,1,68,69,66,67,134 and 135). This method is using all 4 free physical core to provide 8 asynchronous progress threads. Last method (Fig. 10) is using same physical core as MPI process (138,146,154,162,170,178,186 and 192). This method does not use any additional physical core for asynchronous progress threads and use other hardware thread (e.g. hardware thread #2) on the same physical core of MPI process. For example, logical core 2 (hardware thread #0 on the physical core 2) is used for MPI process 0, and logical core 138 (hardware thread #2 on the physical core 2) is used one asynchronous progress thread. Since using same physical core for MPI communication and computation, this method is expected to have better cache hit and to reduce internal data transfer during overlapping. Table 2 is the list of core numbers for `I_MPI_ASYNC_PROGRESS_PIN`.

Table 2: Summary of mapping method of asynchronous progress thread

Mapping method	`I_MPI_ASYNC_PROGRESS_PIN=`
#0. Without asynchronous thread	N/A (`I_MPI_ASYNC_PROGRESS=off`)
#1. Intel MPI default	(271,270,269,268,267,266,265,264)
#2. First 2 physical cores	0,1,68,69,136,137,204,205
#3. Last 2 physical cores	66,67,134,135,202,203,270,271
#4. First 2 and last 2 cores	0,1,68,69,66,67,134,135
#5. Using other hardware thread on same core	138,146,154,162,170,178,186,192

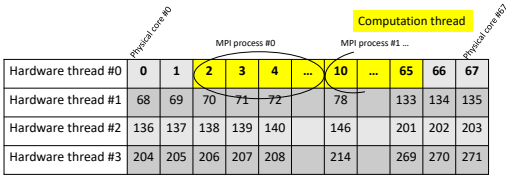


Figure 5: Figure of core mapping method #0 on Xeon Phi processor

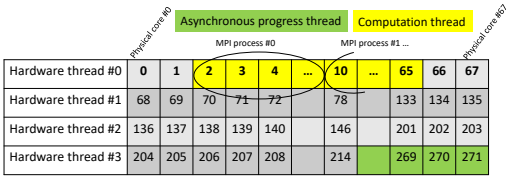


Figure 6: Figure of core mapping method #1 on Xeon Phi processor

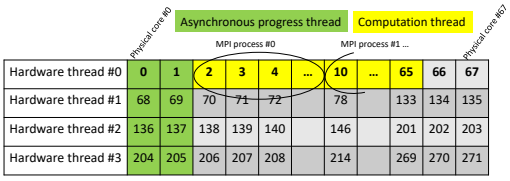


Figure 7: Figure of core mapping method #2 on Xeon Phi processor

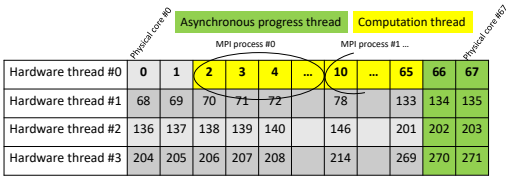


Figure 8: Figure of core mapping method #3 on Xeon Phi processor

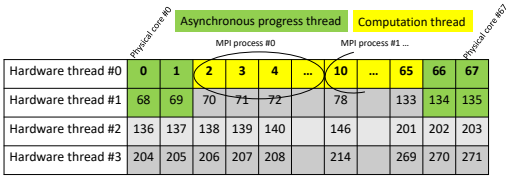
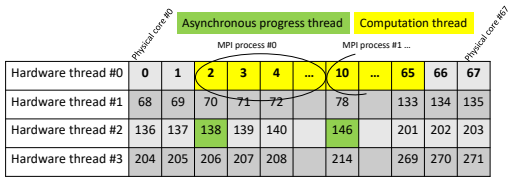


Figure 9: Figure of core mapping method #4 on Xeon Phi processor



Hardware thread #0	**0**	**1**	**2**	**3**	**4**	**...**	**10**	**...**	**65**	**66**	**67**
Hardware thread #1	68	69	70	71	72		78		133	134	135
Hardware thread #2	136	137	138	139	140		146		201	202	203
Hardware thread #3	204	205	206	207	208		214		269	270	271

Figure 10: Figure of core mapping method #5 on Xeon Phi processor

4.3 Experiments of various mapping methods on 512 node

We evaluated which mapping method is best for GeoFEM by using smaller test case on smaller cluster size. We set a problem size of GeoFEM as (256, 128, 128) for 512 node test as smaller proxy of full (512, 256, 256) size, which is equivalent to 100,663,296 DOFs. We investigate all method from #0 to #5 on the 512 node by using best performing number during 5 executions for each algorithm. Figure 11 shows relative performance of algorithm 4:Gropp's CG which is using `MPI_Iallreduce`. Except mapping method #2, all methods of asynchronous progress control enabled shows better performance than method #0. Method #4 is best performing core configuration with 37% improvement. While method #1 and #5 need only free logical cores, method #3 and #4 need more hardware resource as free physical cores. Method #4 needs 2 more physical cores than Method #3. It is reasonable results that more hard ware resource gives more performance improvement. We are not sure why method #2 showed poor performance. One possible reason is OS noise and jitters. Message size of GeoFEM's `MPI_Iallreduce` is 16 byte. It is well known such small size global synchronized collective communications are affected by OS noise and jitters [22].

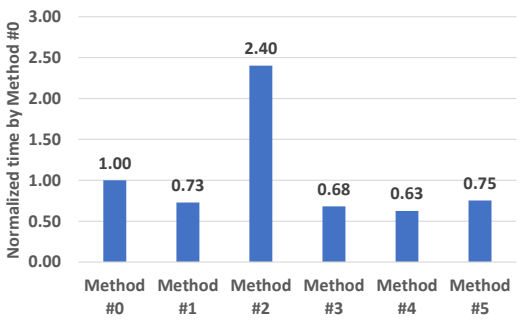


Figure 11: Result of 512 node experiment. Method #0 to #5 shows relative performance. Normalized by Method #0 (without asynchronous progress control).

5 IHK/MCKERNEL

This section gives a general overview of IHK/McKernel and describes developments targeted for asynchronous communication threads on Oakforest-PACS.

5.1 Overview

The IHK/McKernel multi-kernel operating system is comprised of two main components. Interface for Heterogeneous Kernels (IHK) [7], a low-level software infrastructure, provides capabilities for partitioning resources in a many-core environment (*e.g.*, CPU cores and physical memory) and it enables management of lightweight kernels. McKernel is a lightweight co-kernel developed on top of IHK. An overview of the multi-kernel architecture is depicted in Fig. 12.

IHK is capable of allocating and releasing host resources dynamically and no reboot of the host machine is required when altering configuration. It is implemented as a collection of Linux kernel modules without any modifications to the Linux kernel itself, which enables straightforward deployment of the multi-kernel stack on a wide range of Linux distributions. Besides resource and LWK management, IHK also facilitates an Inter-kernel Communication (IKC) layer, which is used for implementing system call delegation (discussed below).

McKernel has been developed from scratch and while it is designed explicitly for high-performance computing workloads it retains a Linux compatible application binary interface (ABI) so that it can execute unmodified Linux binaries. There is no need for recompiling applications or for any McKernel specific libraries. McKernel implements only a small set of performance sensitive system calls and the rest of the OS services are delegated to Linux. Specifically, McKernel implements memory management, it supports processes and multi-threading, it has a simple round-robin co-operative (tick-less) scheduler, and it supports standard POSIX signaling. It also implements inter-process memory mappings and it offers interfaces for accessing hardware performance counters.

For each OS process executed on McKernel there is a process running on Linux, which we call the *proxy-process*. The proxy process' main role is to assist system call offloading. Essentially, it provides the execution context on behalf of the application so that offloaded system calls can be invoked in Linux. For more information on system call offloading, refer to [8]. The proxy process also provides means for Linux to maintain various state information that would have to be otherwise kept track of in the co-kernel. McKernel for instance has no notion of file descriptors, but it simply returns the number it receives from the proxy process during the execution of an `open()` system call. The actual set of open files (*i.e.*, file descriptor table, file positions, etc.) are managed by the Linux kernel. Relying on the proxy process, McKernel provides transparent access to Linux device drivers not only in the form of offloaded system calls (*e.g.*, through `write()` or `ioctl()`), but also via direct device mappings. Details of the device mapping mechanism has been described elsewhere [9].

5.2 Support for Asynchronous Progress

As shown in Fig. 12, IHK partitions CPU cores into Linux and lightweight kernel (LWK) domains. Only the LWK CPUs are exposed to

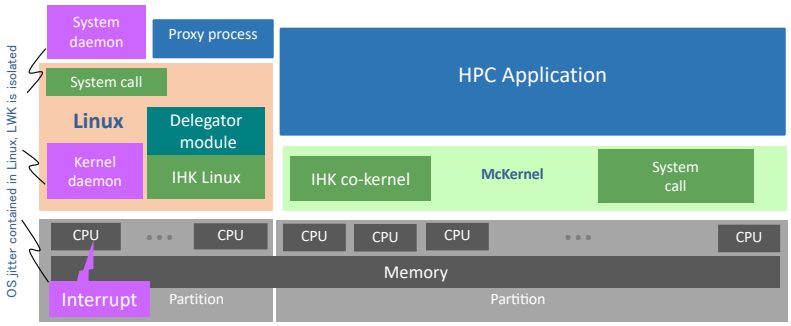


Figure 12: Architectural overview of IHK/McKernel

application running on McKernel, *e.g.*, on the Xeon Phi processor used in this study only 256 logical CPU cores from the overall 272 are visible in McKernel. In addition, McKernel renumbers CPUs and provides an automatic process pinning mechanism that integrates with MPI to ease the burden on users for dealing with CPU numbering related issues.

With respect to asynchronous progress threads, the main challenge that had to be addressed in McKernel was the enablement of additional CPU cores in LWK without directly exposing them to computation threads of the application. For this purpose McKernel's internal CPU management code as well as the process pinning mechanism has been enhanced for asynchronous progress awareness. Specifically, CPU cores dedicated to auxiliary threads are kept transparent from application code and MPI progress threads are pinned automatically to those CPUs. We use the second tile of the Xeon Phi and pin one progress thread per logical CPU on each core.

6 NUMERICAL EXPERIMENTS

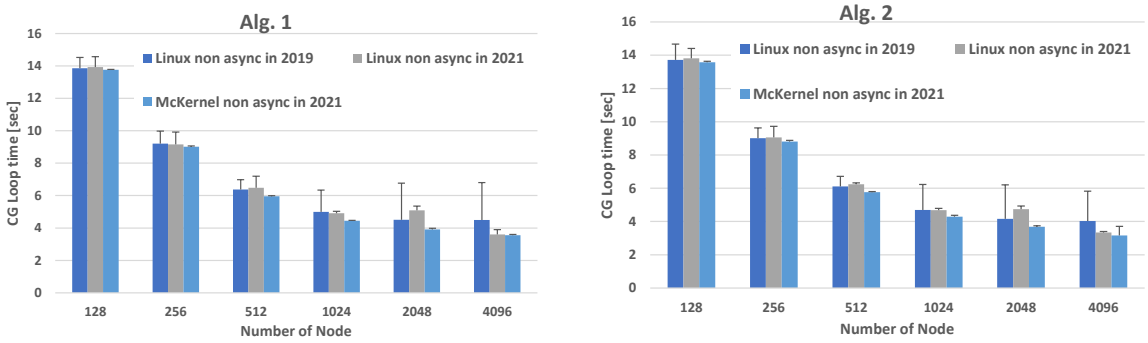
6.1 Performance results on Linux (measured in 2019)

Since method #4 is the best core mapping so we measured real large problem size up to 4096 node by using asynchronous progress thread with method #4 in addition to method #0 as a baseline. First measurement was done on May 2019 by using Intel tools version 2019 Update 1. We fixed the problem size as (512, 256, 256), which has 100,663,296 DOFs, and performed strong scaling measurements. Fig. 13a-14b show bar graphs as best performing numbers during 5 executions on each run, which have also error bars as worst performing numbers of 5 runs. Detailed performance numbers are also listed on the Table 7 in the appendix as a reference.

Since Alg. 1-4 do not have global asynchronous MPI calls (`MPI_Iallreduce`), but using `MPI_Allreduce`, it is not related with asynchronous progress setups, essentially. Fig. 13a-14b have measurement results for those algorithms marked as "Linux non async in 2019". We can see performance improvement (or decreasing CG Loop time) with growing the number of node from 128 to 2,048 node. From 2,048 to 4,096 node, we are seeing saturation of performance or worse performance by poor scaling. These performance saturations are derived by time increase of `MPI_Allreduce` on the large number of node executions. For example, by profiling on 2048 node, total time of `MPI_Allreduce` calls for Alg. 1 is 59% of total CG Loop time while it is 15% on 128 node. It is what we would like to hide or overlap into computation by using asynchronous progress threads.

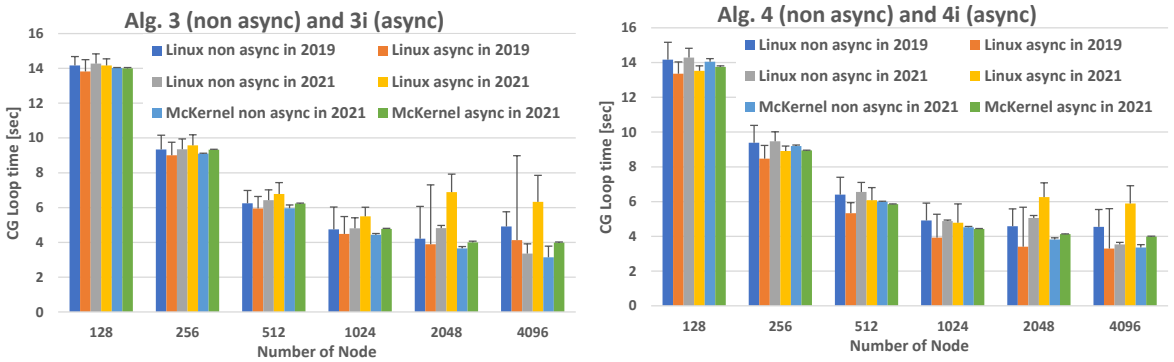
Performance of the original CG (Alg.1) saturates, as number of nodes is more than 2,048. On the contrast, Chronopoulos/Gear algorithm (Alg.2) keeps scalability up to 4,096 nodes, and much better than Alg.1, because communication overhead by `MPI_Allreduce` is reduced. Pipelined CG (Alg.3) and Gropp's CG (Alg.4) are generally slower than Alg.1, because the amount of computations per each iteration is larger than Alg.1. Alg.3i and Alg.4i with `MPI_Iallreduce` are also slower, but both of them are much faster than Alg.1, and competitive with Alg.2. The reason of this improvement is not clear, while `MPI_Iallreduce` may reduce the communication overhead at 4,096 nodes.

Effects of asynchronous progress threads are significant with more than 512 nodes, although performance of Alg.3i is not scaled at 4,096 nodes. Generally, the results show Alg.3i and Alg.4i provide much better scalability than Alg.1 by combination of `MPI_Iallreduce` and asynchronous progress threads. Fig. 14a-14b have the result of asynchronous progress threads for the Alg. 3i and 4i, which have asynchronous global MPI calls as `MPI_Iallreduce`. We can compare performance "Linux no async in 2019" and "Linux async in 2019" on Fig. 14a and 14b, respectively. We are seeing significant performance improvement on Alg. 4i while Alg. 3i was expected to have similar performance improvement by overlapping, but result of Alg. 3i showed smaller performance improvement than Alg. 4i. It is not clear why it shows such behavior. Table 3 show summarized performance improvement results which have comparison of original non asynchronous algorithm (Alg. 3 and 4) and asynchronous versions with asynchronous progress control (Alg. 3i and 4i). We successfully got up to 19% and 38% improvements for Alg. 3 to 3i and Alg. 4 to 4i, respectively, by overlapping effect of asynchronous progress control.



(a) Algorithm 1 (b) Algorithm 2

Figure 13: Results without asynchronous progress on Linux and McKernel



(a) Algorithms 3 and 3i (b) Algorithms 4 and 4i

Figure 14: Results comparing synchronous and asynchronous progress for algorithms 4 and 5 on Linux and McKernel

We can see growing size of error bars for all algorithms on top of "Linux no async in 2019" and "Linux async in 2019" in Fig. 13a-14b with increasing the number of node. In [5][6], we examined the impact of the IHK/McKernel [10], and seeing stable measurement results (small error bars) on large-scale systems by reducing OS noise and communication overhead. In the next section, we will discuss the efforts which we try applying IHK/McKernel for GeoFEM with asynchronous progress control.

6.2 Performance results on Linux (measured in 2021)

Since McKernel was required some modifications as described in Section 5, we needed revisited Linux measurements with newer version of environments including newer versions of BIOS, Firmware, OS, Interconnect driver (IFS), Parallel file system (Lustre) driver, macro task software (e.g. scheduler) and MPI library to get new baseline after 2 years. We used same method #4 as the best core

Table 3: Performance improvement of introducing overlaps by MPI_Iallreduce and asynchronous progress control on Linux (measured in 2019)

Node	Alg. 3 non asyc to 3i with async	Alg. 4 non async to 4i with async
128	1.03	1.06
256	1.04	1.11
512	1.05	1.20
1024	1.06	1.25
2048	1.08	1.34
4096	1.19	1.38

mapping with same large problem size up to 4096 node by using asynchronous progress control in addition to method #0 as a baseline similarly in 2019. This second measurement was done on Nov. 2021 by using Intel tools version 2019 Update 9. All performance numbers are listed on the Table 8 in the appendix as a reference.

Results marked as "Linux non async in 2021" on Fig. 13a-14b and "Linux async in 2021" on Fig. 14a-14b show the performance of result of re-measurements by using newer software and driver versions. We can see similar performance behaviors in Fig. 13a-14b without asynchronous progress control while seeing smaller run to run variances (smaller error bars). On Fig. 14a-14b, we can see worse performance results of "Linux non async in 2021" case than "Linux non async in 2019". We are not sure why `MPI_Iallreduce` version's GeoFEM (Alg. 3i and 4i) show poor performance than 2019. Since almost of all software environments are changing from 2019, it is very difficult to identify the root cause. Also due to security fixes and patches in two years, we can not revert entire system same as 2019.

We can compare the effect of asynchronous thread control, similarly, "Linux async in 2019" case and "Linux async in 2021" case on Fig. 14a-14b. Alg. 3i and 4i show better performance by using asynchronous progress threads up to 512 and 1024 node, respectively. But on the larger node counts, we are seeing worse performance results by using synchronous progress threads. It is unexpected results that effective on smaller node count while negative impact on larger node count. In 2019, asynchronous progress threads were effective with growing node count up to 4,096. As same reasons as non async 2021 results, we are not sure and have difficulties to identify the root causes.

Table 4 show summarized performance improvement and degradation results which have comparison of original non asynchronous algorithm (Alg. 3 and 4) and asynchronous versions with asynchronous progress control (Alg. 3i and 4i). As described in section 6.1, We got up to 19% and 38% improvements for Alg. 3 to 3i and Alg. 4 to 4i, respectively in 2019 but we can see the positive effect of asynchronous progress control on fewer node count only.

Table 4: Performance improvement and degradation of introducing overlaps by MPI_Iallreduce and asynchronous progress control on Linux (measured in 2021). Note: Below 1.00 means negative effect.

Node	Alg. 3 non asyc to 3i with async	Alg. 4 non async to 4i with async
128	1.01	1.40
256	0.98	1.26
512	0.95	0.96
1024	0.87	0.82
2048	0.70	0.61
4096	0.53	0.43

6.3 Performance results on McKernel (measured in 2021)

For simplicity and little implementation cost, the modification for asynchronous progress control on McKernel is intended to use first two physical cores only while it is expected more improvement by allocating 4 unused physical cores to McKernel. 8 logical threads provided by first two physical cores handle asynchronous progress control in addition to IHK Linux for this time. It is almost equivalent to Fig. 7 or the method #2. All other measurements conditions are same as Linux in 2021.

Through Fig. 13a to 14b, which have case of non asynchronous progress control, we can see a little performance improvement on larger node count while almost same on lower node count. But we can clearly see error bars on McKernel are significantly smaller than that of Linux. This stable and less variant results are one of the benefits by McKernel. Similarly on Fig. 14a-14b, we can see much smaller error bars on McKernel, which are marked on top of "McKernel non async in 2021" and "McKernel async in 2021" bars.

Table 5 shows summarized performance improvement on McKernel without asynchronous progress control, compared with best Linux result without asynchronous progress control of 2019 and 2021. All other measurement results show better performance than Linux due to less noisy and low overhead nature of McKernel. For example, We can see up to 20% performance improvement on 2,048 node and 6% on 4,096 node for the Alg. 4. All performance numbers are listed on the Table 9 in the appendix as a reference.

Next, discussing asynchronous progress control on McKernel. Table 6 show summarized performance improvement and degradation results which have comparison of original non asynchronous algorithm (Alg. 3 and 4) and asynchronous versions with asynchronous progress control (Alg. 3i and 4i). We cannot see beneficial improvement on Alg.3i. With Alg. 4i, asynchronous progress threads give 2-3% performance improvement from 128 node to 1,024 node.

Table 5: Performance improvement on McKernel vs. best of Linux in 2019 and 2021. Note: Below 1.00 means negative effect.

Speedup	Alg. 1	Alg. 2	Alg. 3	Alg. 4
128	1.01	1.01	1.01	1.01
256	1.02	1.02	1.03	1.02
512	1.07	1.06	1.05	1.07
1024	1.10	1.09	1.07	1.08
2048	1.15	1.13	1.15	1.20
4096	1.01	1.06	1.07	1.06

7 RELATED WORK

Non-blocking collective operations were introduced in MPI several years ago [23]. However, only a few recent studies have explored thread placement strategies for asynchronous communication progress thread placement for non-blocking collective operations. Ohlmann et. al. presented Intel MPI's asynchronous progress control with an application case study demonstrating the benefits of progress thread placement [25].

Denis et. al. studied progress thread placement on many-core CPUs with a special focus on symmetric multithreading. Similarly to our findings, they also reported that running asynchronous progress on SMT threads of an application core can degrade performance due to cache effects in intranode communication [24].

Table 6: Performance improvement and degradation of introducing overlaps by MPI_Iallreduce and asynchronous progress control on McKernel (measured in 2021). Note: Below 1.00 means negative effect.

Node	Alg. 3 non asyc to 3i with async	Alg. 4 non async to 4i with async
128	1.00	1.02
256	0.98	1.03
512	0.95	1.02
1024	0.93	1.02
2048	0.91	0.92
4096	0.79	0.84

8 CONCLUSION AND FUTURE WORK

We have demonstrated that the asynchronous progress control is very effective with appropriate algorithm changes for CG solvers to overlap global synchronization and computation. We achieved 38% performance improvement on 4,096 node.

Although re-measurements on Linux in 2021 are not stable and worse than what measured in 2019 on larger node count more than 1,024, we are also able to confirm McKernel realized noise less and stable measurement in addition to performance improvement on GeoFEM. In most case, McKernel provided better performance especially on large node count even if compared with 2019 Linux results. Especially for 2,048 and 4,096 nodes, we got 20% and 6% improvements, respectively.

Furthermore, we have modified McKernel for asynchronous progress control and applied to real GeoFEM applications. From 128 node to 512 nodes, at least, we are able to see performance improvement by the combination of McKernel and asynchronous progress control. Though the effect is limited up to 1,024 node counts, we got at most 2-3% performance improvement from 128 node.

Work is ongoing and it is expected that we will run these performance experiments again after whole system tuning on Linux and McKernel to address the 2021 year's performance degradation than that of 2019. In the case of physical 4 cores allocation to IHK kernel for asynchronous progress control on McKernel will also be a future work. Furthermore, to investigate an effect on SMT threads more could lead to efficient use of asynchronous progress control for more MPI ranks per node or pure MPI parallelism situation without OpenMP threading.

ACKNOWLEDGMENTS

This work is supported by *JSPS Grant-in-Aid for Scientific Research (S)* (19H05662), and by *Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures* (jh20037-NAH, jh20041-NAH). Authors would like to thank Yoshio Sakaguchi, Nobuteru Mizoe and Toshiro Saiki from Fujitsu.

REFERENCES

[1] Intel's Asynchronous Progress Control: https://www.intel.com/content/www/us/en/develop/documentation/mpi-developer-guide-linux/top/additional-supported-features/asynchronous-progress-control.html
[2] Ghysels, P. and W. Vanroose, Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm, Parallel Computing 40-7, 224-238, 2014
[3] Nakajima, K., Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator, ACM/IEEE Proceedings of SC 2003, 2003
[4] Joint Center for Advanced High Performance Computing (JCAHPC): http://jcahpc.jp/
[5] Nakajima, K., Gerofi, B., Ishikawa, Y., Horikoshi, M., Parallel Multigrid Methods on Manycore Clusters with IHK/McKernel, IEEE Proceedings of 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 2019) in conjunction with SC19, Denver, CO, 2019
[6] Nakajima, K., Gerofi, B., Ishikawa, Y., Horikoshi, M., Efficient Parallel Multigrid Method on Intel Xeon Phi Clusters, ACM Proceedings of IXPUG HPC Asia 2021, 2021
[7] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori and Yutaka Ishikawa, Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures, 21th Intl. Conference on High Performance Computing HiPC, 2014
[8] Balazs Gerofi, Akio Shimada, Atsushi Hori and Yutaka Ishikawa, Partially Separated Page Tables for Efficient Operating System Assisted Hierarchical Memory Management on Heterogeneous Architectures, 13th Intl. Symposium on Cluster, Cloud and Grid Computing CCGRID, 2013
[9] Balazs Gerofi, Masamichi Takagi, Atsushi Hori, Guo Nakamura, Tomoki Shirasawa and Yutaka Ishikawa, On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel, IEEE International Parallel and Distributed Processing Symposium IPDPS, 2016
[10] Gerofi, B, Riesen, R., Takagi, M., Boku, T., Nakajima, K., Ishikawa, Y., Wisniewski, R.W., Performance and Scalability of Lightweight Multi-kernel Based Operating Systems, IEEE Proceedings of IPDPS 2018, 116-125, 2018
[11] Saad, Y., Iterative Methods for Sparse Linear Systems (2nd Edition), SIAM, 2003
[12] Smith, B., P. Bjostad, and W. Gropp, Domain Decomposition, Parallel Multilevel Methods for Elliptic Partial Differential Equations, Cambridge Press, 1996
[13] Demmel, J., Hoemmen, M., Mohiyuddin, M., Yelick, K., Avoiding communication in sparse matrix computations, IEEE Proceedings of 22nd International Symposium on Parallel and Distributed Processing, 2008 (IPDPS 2008), 2008
[14] Nakajima, K.and Ogita, T., Evaluations of Stability of Parallel Conjugate Gradient Methods based on Pipelined Algorithmss, IPSJ SIG Technical Reports 2018-HPC-167-26 (in Japanese), 2018
[15] Chronopoulos, A.T. and Gear, C.W., s-Step iterative methods for symmetric linear systems. Journal of Computational and Applied Mathematics. v25 i2. 153-168
[16] Hanawa, T., Nakajima, K., Ohshima, S., Hoshino, T., Ida, A., Performance Evaluation of Pipelined CG Method, IPSJ SIG Technical Report, Vol.2016-HPC-157, No.6, 2016 (in Japanese)
[17] Supercomputing Research Division, Information Technology Center, The University of Tokyo (ITC/U.Tokyo): http://www.cc.u-tokyo.ac.jp/
[18] Gropp, W., Update on libraries for blue waters, https://wgropp.cs.illinois.edu/bib/talks/tdata/2011/Stream-nbcg.pdf
[19] Horikoshi, M., Nakajima, K., Gerofi, B., Ishikawa, Y., Parallel Preconditioned Itertative Solvers on Oakforest-PACS, 28th Seminar of MEPA (Algorithms for Matrix / Eigenvalue Problems and their Applications, JSIAM (Japan Society for Industrial and Applied Mathematics)), 2019
[20] STREAM Benchmark: https://www.cs.virginia.edu/stream/
[21] Amit Ruhela, Hari Subramoni, Sourav Chakraborty, Mohammadreza Bayatpour, Pouya Kousha and Dhabaleswar K.(DK) Panda, Efficient design for MPI asynchronous progress without dedicated resources, Parallel Computing 85,13-26, 2019
[22] Horikoshi, M., Meadows L., Elken T., Sivakumar P., Mascarenhas, M., Erwin J., Durnov D., Sannikov A., Hanawa T., Boku, T., Scaling collectives on large clusters using Intel(R) architecture processors and fabric, ACM Proceedings of IXPUG HPC Asia 2018, 2018
[23] Hoefler T., Kambadur P., Graham R.L., Shipman G., Lumsdaine A. (2007) A Case for Standard Non-blocking Collective Operations. In Recent Advances in Parallel Virtual Machine and Message Passing Interface. EuroPVM/MPI 2007. Lecture Notes in Computer Science, vol 4757. Springer, Berlin, Heidelberg
[24] Alexandre Denis, Julien Jaeger, Hugo Taboada, Progress Thread Placement for Overlapping MPI Non- Blocking Collectives using Simultaneous Multi-Threading. COLOC: 2nd workshop on data locality, in conjuction with Euro-Par 2018, 2018
[25] Sebastian Ohlmann, Fabio Baruffa, Markus Rampp, Overlapping communication and computation using the Intel MPI library's asynchronous progress control, IXPUG Meeting 2020, https://www.ixpug.org/resources/overlapping-communication-and-computation-using-the-intel-mpi-library-s-asynchronous-progress-control

A APPENDIX

A.1 Table of measurement data in 2019

Table 7: Results on Linux (measured in 2019). Alg. 3i and 4i used asynchronous progress threads while others did not use. From 128 to 4,096 nodes.

	Alg.1 [sec]	Alg.2 [sec]	Alg.3 [sec]	Alg.3i [sec]	Alg.4 [sec]	Alg.4i [sec]
128	1.39E+01	1.37E+01	1.42E+01	1.38E+01	1.42E+01	1.34E+01
256	9.21E+00	9.01E+00	9.34E+00	9.01E+00	9.39E+00	8.47E+00
512	6.37E+00	6.10E+00	6.24E+00	5.95E+00	6.40E+00	5.33E+00
1024	5.00E+00	4.69E+00	4.75E+00	4.49E+00	4.91E+00	3.93E+00
2048	4.51E+00	4.16E+00	4.22E+00	3.89E+0	4.58E+00	3.41E+00
4096	4.50E+00	4.03E+00	4.91E+00	4.14E+00	4.55E+00	3.29E+00

A.2 Tables of measurement data in 2021

Table 8: Results on Linux (measured in 2021). Alg. 3i and 4i used asynchronous progress threads while others did not use. From 128 to 4,096 nodes.

	Alg.1 [sec]	Alg.2 [sec]	Alg.3 [sec]	Alg.3i [sec]	Alg.4 [sec]	Alg.4i [sec]
128	1.39E+01	1.38E+01	1.43E+01	1.42E+01	1.43E+01	1.35E+01
256	9.15E+00	9.06E+00	9.35E+00	9.58E+00	9.47E+00	8.92E+00
512	6.49E+00	6.24E+00	6.43E+00	6.78E+00	6.56E+00	6.08E+00
1024	4.91E+00	4.68E+00	4.80E+00	5.50E+00	4.88E+00	4.79E+00
2048	5.09E+00	4.74E+00	4.82E+00	6.88E+00	5.05E+00	6.27E+00
4096	3.60E+00	3.34E+00	3.36E+00	6.33E+00	3.54E+00	5.89E+00

Table 9: Results on McKernel (measured in 2021). Alg. 3i and 4i used asynchronous progress threads while others did not use. From 128 to 4,096 nodes.

	Alg.1 [sec]	Alg.2 [sec]	Alg.3 [sec]	Alg.3i [sec]	Alg.4 [sec]	Alg.4i [sec]
128	1.38E+01	1.36E+01	1.40E+01	1.40E+01	1.40E+01	1.37E+01
256	9.01E+00	8.81E+00	9.10E+00	9.32E+00	9.20E+00	8.95E+00
512	5.96E+00	5.77E+00	5.96E+00	6.24E+00	5.99E+00	5.85E+00
1024	4.45E+00	4.29E+00	4.44E+00	4.78E+00	4.52E+00	4.43E+00
2048	3.91E+00	3.68E+00	3.66E+00	4.00E+00	3.82E+00	4.13E+00
4096	3.56E+00	3.16E+00	3.14E+00	3.99E+00	3.36E+00	3.99E+00