Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Comparing the Performance of Pure MPI vs. OpenMP and MPI

HPC Project Report

Sun Zhongxia, Gao Yurong, Pan Hao

zhosun@student.ethz.ch

yurgao@student.ethz.ch

haopan@student.ethz.ch

Institute for Atmospheric and Climate Science

Department Environmental Systems Science, ETH Zürich

September 9, 2023

Contents

1. Introduction

Large-scale parallel computers today predominantly use a distributed-memory architecture at the system level, yet leverage shared-memory compute nodes as fundamental units. In recent years, we have witnessed the emergence of hybrid architectures (MPI+OpenMP) that blend the features of both shared and distributed memory. Yet, a significant number of parallel applications still employ pure MPI for parallelization, often overlooking the inherent hardware structure.

This prevalent use of pure MPI can be attributed to the relative ease of porting existing MPI applications and libraries to shared-memory systems, coupled with an implicit assumption that MPI libraries would efficiently optimize intranode message passing. Furthermore, the commonality of small to medium-sized shared-memory nodes, typically featuring two to four processors per node, has fueled the belief that hybrid codes rarely outperform their pure MPI counterparts for equivalent tasks.

This report will challenge this belief by comparing the performance of pure MPI implementations with hybrid MPI/OpenMP implementations. We have conducted a series of computational experiments, followed by a meticulous analysis of execution times and other performance metrics. The subsequent sections of this paper will detail our experimental design, results, and analyses, shedding light on the relative advantages and appropriate use cases for different implementations.

2. Method

2.1. Structure of hybrid OpenMP/MPI program

The fundamental premise of a hybrid OpenMP/MPI programming model involves empowering any MPI process to initiate a team of OpenMP threads, similar to the role of the master thread in a standalone OpenMP program. In this research, our approach involves decomposing domains and assigning them to separate MPI processes for independent computation. Concurrently, we utilize OpenMP to enable parallelism within these individual computational domains, as illustrated in Figure 1.

2.2. Experimental design

Prior to launching the MPI processes, it's crucial to specify the maximum number of OpenMP threads for each MPI process, similar to the procedure for a standalone OpenMP program. The interaction between OpenMP threads and MPI processes within a node can take numerous forms, thereby offering substantial opportunities for optimization in most hybrid applications. For a single

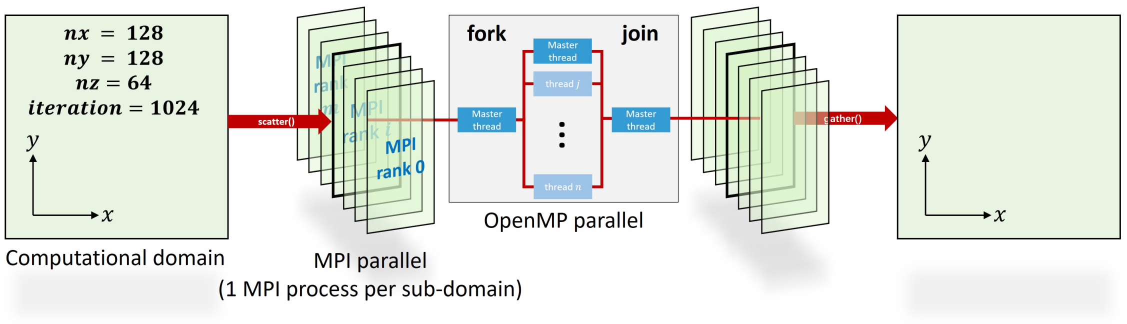

Figure 1. Schematic of hybrid OpenMP/MPI stencil program (z-dimension neglected).

node, we've designed 6 experimental sets, as demonstrated in Table 1. In each of these sets, the product of the number of MPI processes and the maximum number of OpenMP threads equals the number of hardware threads, which is 24 in this case.

Table 1. Experimental design for 1 node: Mapping between MPI processes and OpenMP threads

	MPI Processes	Number of threads (Max)
Case 1	2	12
Case 2	3	8
Case 3	4	6
Case 4	6	4
Case 5	8	3
Case 6	12	2

We've also done some experiments on 2 nodes, again 6 experimental sets are designed, which are demonstrated in Table 2. The number of MPI processes are set to be even numbers to make sure that these MPI processes can be equally distributed into 2 nodes.

Table 2. Experimental design for 2 nodes: Mapping between MPI processes and OpenMP threads

	MPI Processes	Number of threads (Max)
Case 1	2	24
Case 2	4	12
Case 3	6	8
Case 4	8	6
Case 5	12	4
Case 6	24	2

2.2.1. Blocking and Scheduling

Block partitioning is a widely adopted strategy in loop parallelization. With this approach, loop iterations are segmented into multiple blocks or groups, each of which is handled by an individual thread or process. This not only enhances cache utilization but also curtails the necessity for inter-thread or inter-process communication. Although partitioning can be performed across any dimension, such as i, j, k, and so forth, our study primarily concentrates on k-parallel and j-parallel. The respective codes for the parallel areas can be found in Listings A.1-A.3 in the appendix. The i-parallel dimension has been deliberately disregarded due to its substantial overhead associated with thread creation and management.

There are different ways of distributing the iterations, which are controlled by using the following code: `schedule(type, chunk size)`. The `schedule` clause accepts two parameters. The first one, `type`, specifies the way in which the work is distributed over the threads. The second one, `chunk size`, is an optional parameter specifying the size of the work given to each thread. Three main different options for scheduling are (see schematic in Figure 2):

- `static`: the pieces of work created from the iteration space of the do-loop are distributed over the threads in the team following the order of their thread identification number. By default, the `chunk size` is equal to the number of threads in the team and all pieces are approximately equal in size. If the `chunk size` is specified, the size of the pieces is fixed to that amount. The resulting pieces of work are distributed to the threads in a round-robin fashion.

- `dynamic`: assign the different blocks of work in a dynamic way: as one thread finishes its piece of work, it gets a new one. If `chunk size` is not given, then a size equal to one iteration is considered.

- `guided`: have pieces of work with decreasing sizes. The decreasing law is of exponential nature so that the following pieces of work have half the number of iterations as the previous ones. The `chunk size` specifies the smallest number of iterations grouped into one piece of work. If `chunk size` is not given, then a size equal to one is considered.

We conduct scheduling on the k-parallel hybrid version, and the respective scope for scheduling is the same as Listings A.1-A.3 in the appendix.

2.2.2. Pre-computation Computation on-the-fly

Pre-computation and on-the-fly computation represent distinct approaches to handle intermediate computation results. In pre-computation, intermediate results are stored in advance for use in subsequent computations. Conversely, on-the-fly computation calculates based on the original data precisely when required.

In our specific study of the stencil problem, two Laplacian operators must be applied to the input field. With pre-computation, the initial Laplacian is performed, and the outcome is saved in a temporary field; the second Laplacian is then executed on this temporary field. In contrast, on-the-fly computation merges the two Laplacian operators, conducting the calculation directly on the input field. The respective codes for these methods can be found in Listings A.4 and A.5 in the appendix. Importantly, both codes implement the k-block strategy, enhancing cache utilization.

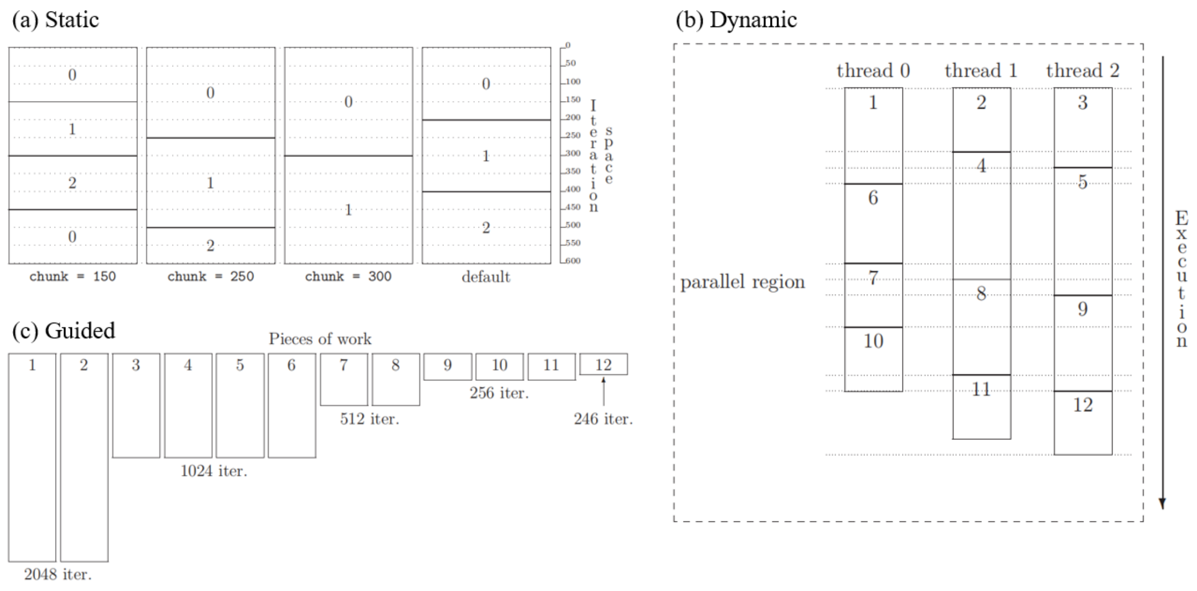

Figure 2. Graphical explaining the working principle of (a) static, (b) dynamic, and (c) guided scheme in scheduling. Adapted from (Hermanns, 2002).

Let n_x, n_y, n_z denote the number of grid points in the x, y, and z directions. We can tally the floating-point calculations and the memory read/write operations for each strategy. For pre-computation, approximately $12n_xn_yn_z$ floating-point computations and $17n_xn_yn_z$ floating-point memory read/write operations are performed, neglecting halo points. For on-the-fly computation, the figures are roughly $25n_xn_yn_z$ and $16n_xn_yn_z$, respectively. By assessing the arithmetic intensity, we determine that this stencil problem is memory-bound. Although memory read/write operations for both strategies appear similar in number, data might be sourced from various cache levels, which is further analyzed in the results section.

2.3. OpenMP parallel region

As the computational domain is evenly divided among different MPI processes, each MPI process carries approximately equal workload. Under this circumstance, our task is to identify OpenMP parallel regions that can maximize acceleration. We conducted CrayPat-lite Performance Statistics on the original stencil2d-mpi.F90 and found that the loops at line 168, line 178 and the two Laplace operations, accounted for 75% of the total program run time. Accordingly, we performed OpenMP parallelization on these parts of the code (see kparallel example in Appendix: Listing A.1, Listing A.2, Listing A.3).

3. Results

3.1. Blocking

In this section, we perform k-parallel and j-parallel in the OpenMP parallel region mentioned in Section 2.3 and compare the results with pure MPI, as shown in Figure 3.

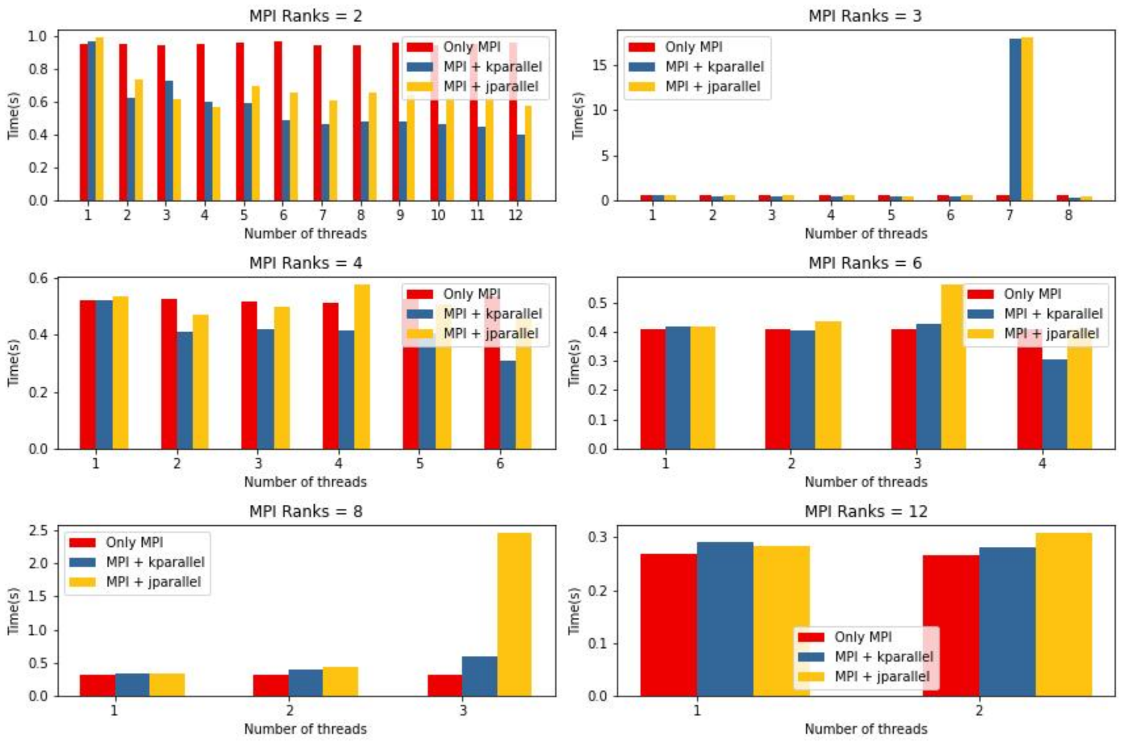

Figure 3. Comparison of "Only MPI" with "OpenMP k-parallel and j-parallel" using varying numbers of MPI ranks and threads. The performance is measured based on execution time.

Figure 3 demonstrates a significant decrease in runtime with the increasing number of MPI processes. This suggests that in environments with limited computational resources, favoring a rise in the number of MPI processes could lead to enhanced performance in hybrid parallel programs.

It is also worth noting that irrespective of the number of MPI processes, parallelization using k-loop consistently outperforms the j-parallel counterpart. The primary reason for this could be the greater number of iterations in k-loop, effectively amortizing the overhead associated with thread creation and management.

An anomaly is observed when the MPI ranks are set to 3 and the number of threads to 7; the runtime increases exponentially, becoming tens of times longer compared to other thread counts. This dramatic slowdown could potentially be attributed to an imbalance in the workload distribution across MPI processes and threads.

Reflecting on the inconsistent trends in the speedup benefits of k-parallel and j-parallel operations, as the number of MPI ranks remains fixed and the number of threads fluctuates - it is notable that during scenarios of limited MPI processes, escalating the thread count can significantly truncate the runtime. However, this effect diminishes and can even invert as the MPI process tally grows. On top of that, we chose to employ the median runtime across the full thread spectrum for each MPI rank as a comparative measure between the performances of pure MPI and MPI/OpenMP hybrid parallelization across different MPI ranks. This median runtime serves as a comprehensive encapsulation of the overall performance of MPI/OpenMP hybrid parallelization under specific MPI rank conditions, as depicted in Figure 4.

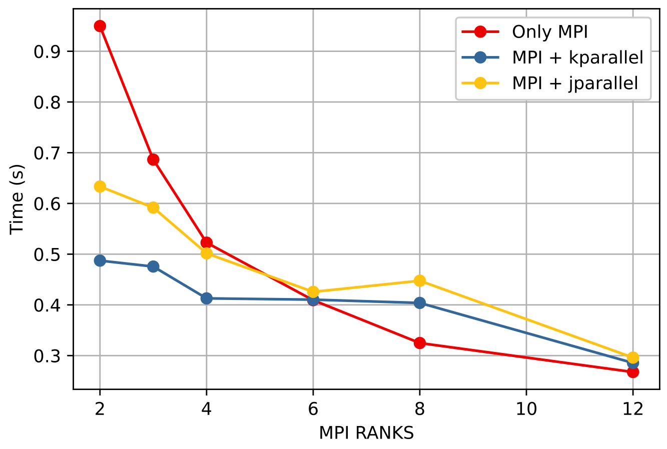

Figure 4. Comparison of pure MPI versus MPI/OpenMP k-parallel and j-parallel implementations across varying MPI rank counts. The evaluation metric is the median execution time computed across all thread counts.

From Figure 4, it can be seen that while increasing the number of MPI ranks can reduce the runtime for all schemes, OpenMP can play a significant role when there are few MPI processes (up to twice as fast as pure MPI). This may be because the communication overhead between threads is much smaller than that between processes. When there are many processor cores but only a few MPI processes, OpenMP can better utilize these cores to perform parallel computations within MPI tasks. Despite fluctuations in computer performance, these results can be stably repeated, indicating that for the research problem at hand, it is not always best to prioritize increasing MPI processes. Hybrid parallelism significantly outperforms pure MPI when the number of MPI processes is low (MPI ranks < 6), whereas pure MPI is more optimal than hybrid parallelism when the number of MPI processes is high (MPI ranks > 6).

In summary, OpenMP can be highly effective when there are a limited number of available ranks for creating MPI processes. However, when a greater number of ranks are available for MPI processes, a pure MPI approach tends to outperform the hybrid approach of combining OpenMP with MPI.

3.2. Scheduling

In this section, we introduce scheduling to the k-parallel hybrid version, and compare their run time with only MPI and k-parallel hybrid version.

Figure 5 shows that when the number of MPI ranks is small and with more threads, scheduling reduces run time compared to the MPI-only version. However, as scheduling decreases and the number of threads increases, scheduling reduction becomes unstable and even increases run time. The reason is the same as the blocking we discussed above. It's an exception of one thread because parallel has no role in this case.

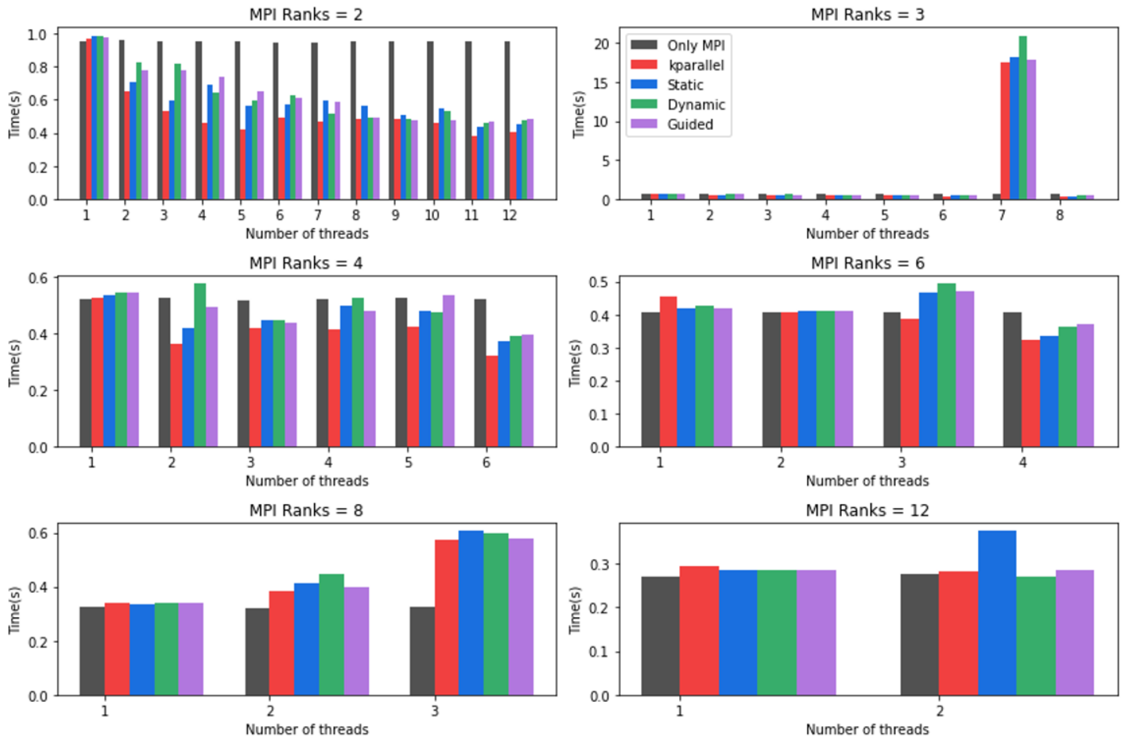

Figure 5. The runtime comparison includes the following versions: MPI-only version (Black bars), MPI version with k-parallel (Red bars), MPI version with k-parallel and static schedule (Blue bars), MPI version with k-parallel and dynamic schedule (Green bars), and MPI version with k-parallel and guided schedule (Purple bars). The comparison involves variations in the number of ranks and the number of threads per rank. Subplots (a)-(f) show the results for 2, 3, 4, 6, 8, and 12 ranks, respectively. X-axis in each subplot is the number of threads. The chunk size in each scheduling scheme was set as default.

When compared to the k-parallel version, scheduling slows down the computing speed in most cases, with simple k-parallel having the shortest computing time. This is because weather and climate codes are usually very well structured, with similar workloads in iterations (excluding boundary layer calculations in the vertical direction). In this case, all three scheduling schemes introduce some additional overhead, such as synchronization between threads and load balancing. These costs can outweigh the benefits, resulting in slower computations. While the simple k-parallel version involves only parallel data splitting without complex scheduling overhead, and thus has the shortest computation time in most cases. This makes the simple k-parallel version a more efficient choice. However, for some other complex, parallel tasks with dynamically changing workloads or dependencies, scheduling may reduce computing time more significantly.

In addition, the related run time of three different scheduling schemes is unstable. This instability could potentially be attributed to different cache effects caused by different rankings and threads, overlapping the race conditions caused by dynamic scheduling and guided scheduling.

We tried to tune the chunk size for each scheduling scheme, under fixing MPI ranks and threads, in order to get a stable optimal parameter. But we found the results are still unstable. The run time for each chunk size is changing during the repeat of the experiment (see Figure 6). Due to the difference in the magnitude of run time for each chunk size being small, the performance fluctuation is possibly caused by the change in hardware heat.

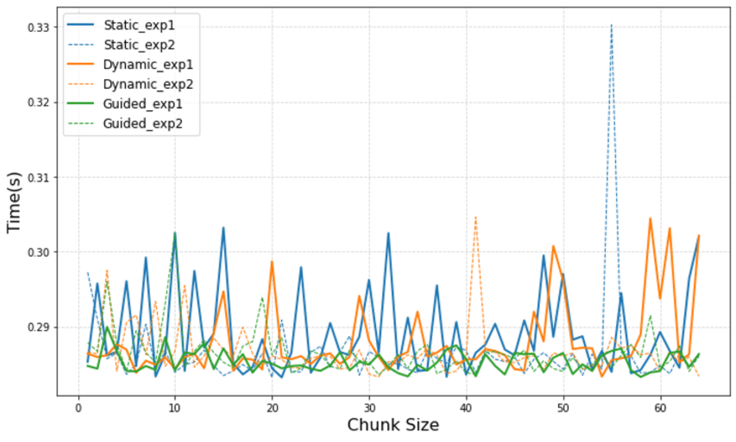

Figure 6. Comparison of the repeated experiment running time of three different scheduling schemes under different chank sizes (Taking ranks=12, threads=2 as an example).

In summary, with fewer MPI ranks, the scheduling performs well compared to the MPI-only version but still lags behind the k-parallel version. When there are more MPI ranks, the MPI-only version takes the lead. The performance of different scheduling schemes appears to be unstable.

3.3. Pre-computation vs. Computation on-the-fly

For pre-computation, we can choose whether or not to fuse the k-loops, leading to two variants: the no-block and the k-block. In the on-the-fly computation, the two Laplacians are merged, so it is inherently fused. Our experiments ran on 2 nodes, with each node containing 12 physical cores. Every core is equipped with a 32KB L1 cache and a 256KB L2 cache. We tested various domain sizes: $64 \times 64 \times 64$, $128 \times 128 \times 64$, $512 \times 512 \times 64$, and $1024 \times 1024 \times 64$. $(n_x \times n_y \times n_z)$ The outcomes can be observed in Fig. 7, Fig. 8, Fig. 9, and Fig. 10, respectively.

From our findings, when working with a smaller domain size and a larger number of MPI ranks, the k-block version of the pre-computation code and the on-the-fly computation code exhibit little difference in speed. However, as the domain size grows and the MPI ranks decrease, the on-the-fly computation becomes notably faster. This discrepancy can be attributed to the L2 cache size. For more substantial domains and fewer MPI ranks, the decomposed field in each rank becomes so large that the temporary field cannot fit within the L2 cache. Consequently, during the second and third j-loops, the temporary field must be retrieved from the L3 cache, which operates slower than the L2 cache.

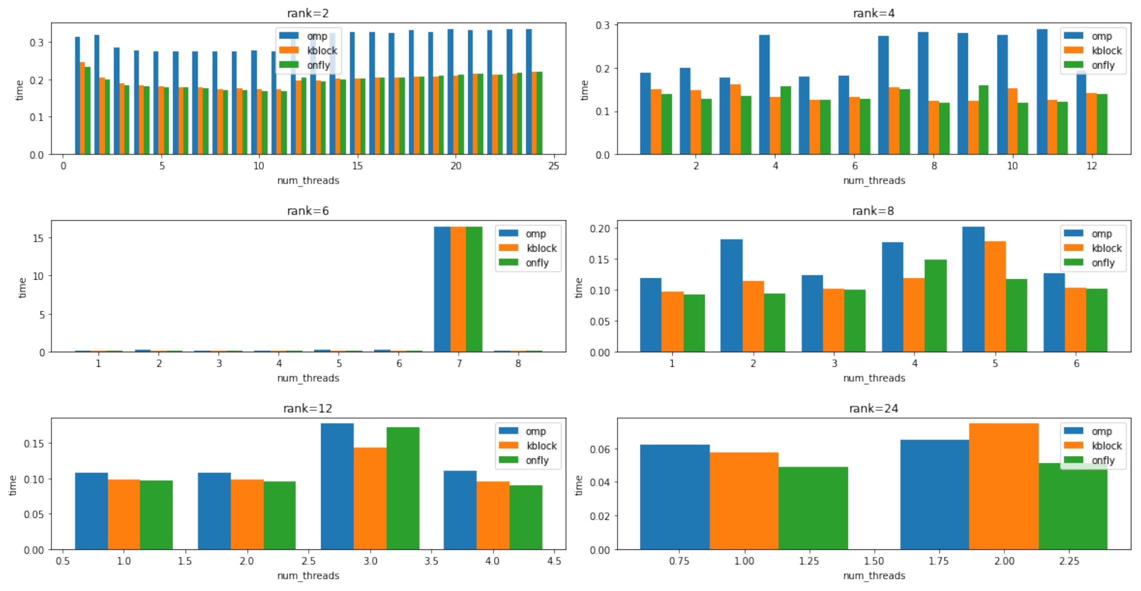

Figure 7. The runtime for different number of MPI ranks, number of OMP threads and strategies, the domain size is $64 \times 64 \times 64$

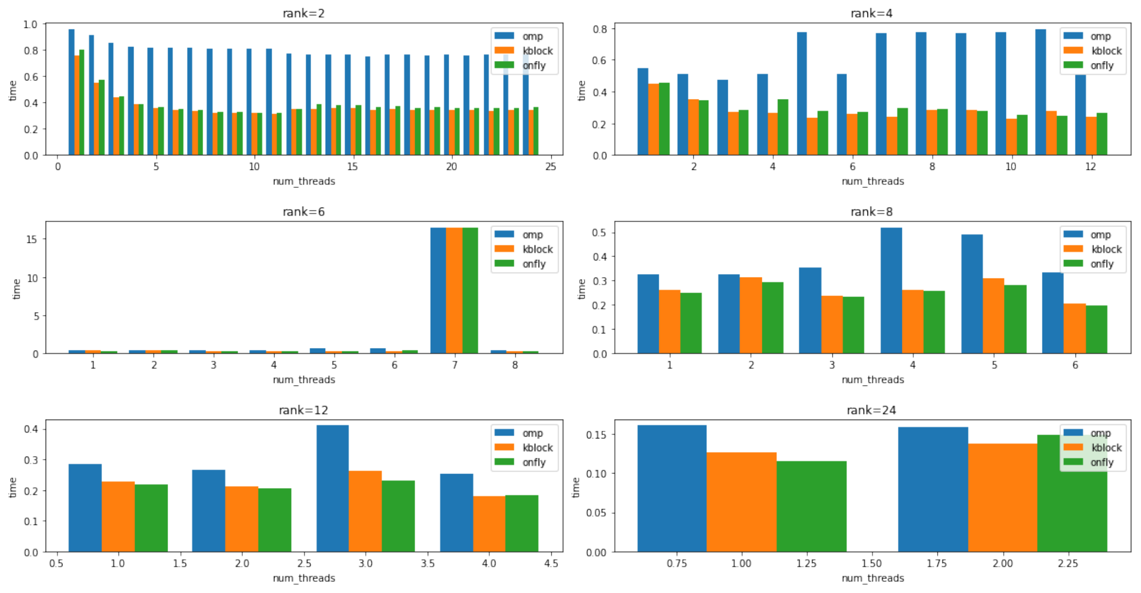

Figure 8. The runtime for different number of MPI ranks, number of OMP threads and strategies, the domain size is $128 \times 128 \times 64$

For a domain size of $128 \times 128 \times 64$, and $rank = 2$, each rank's temporary field size is $128 \times 128 \times 4\text{Byte}/2 = 32\text{kB}$, fitting within the L2 cache. However, with a domain size of $512 \times 512 \times 64$ and 4 MPI ranks, the field size becomes $512 \times 512 \times 4\text{Byte}/4 = 256\text{kB}$, which exactly equals to the size of L2 cache. This explains why $rank = 4$ is a threshold between whether or not on-the-fly computation is faster. For a $1024 \times 1024 \times 64$ domain, the threshold exists between $rank = 12$ and $rank = 24$. In these instances, each rank's temporary field sizes are $1024 \times 1024 \times 4\text{Byte}/12 = 341\text{kB}$ and $1024 \times 1024 \times 4\text{Byte}/24 = 171\text{kB}$, respectively. The former exceeds the L2 cache size, while the latter remains within its limits.

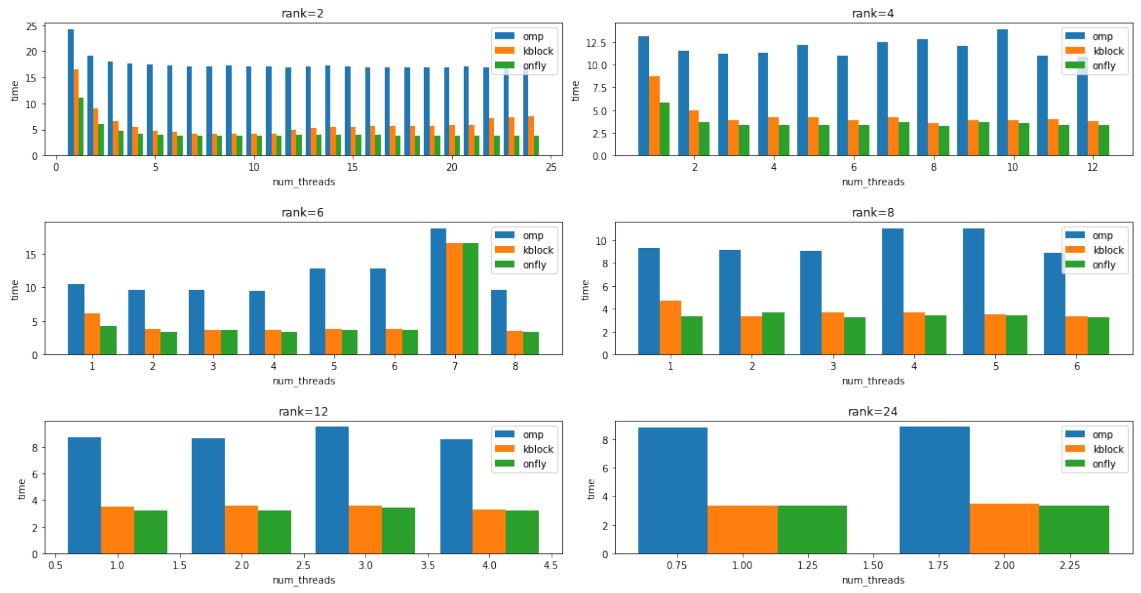

Figure 9. The runtime for different number of MPI ranks, number of OMP threads and strategies, the domain size is $512 \times 512 \times 64$

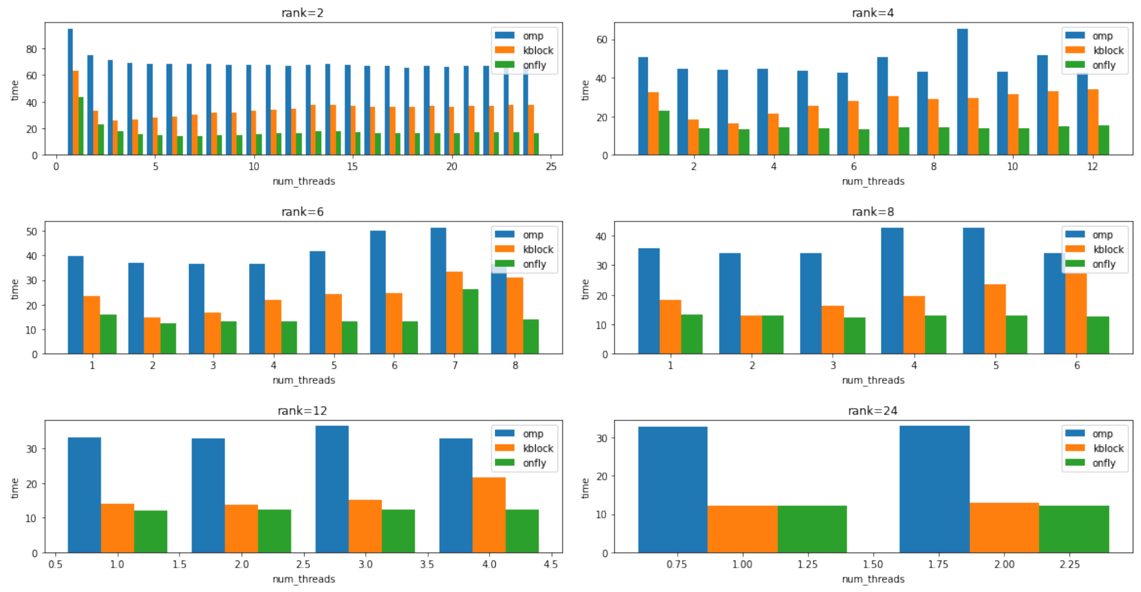

Figure 10. The runtime for different number of MPI ranks, number of OMP threads and strategies, the domain size is $1024 \times 1024 \times 64$

In summary, for memory-bound computations, on-the-fly computation has an edge over pre-computation when each rank's domain size surpasses the L2 cache size. This advantage arises because on-the-fly computation doesn't need to cache the temporary field, though it demands more floating-point calculations. Conversely, we can infer that in compute-bound scenarios, pre-computation is likely to outperform on-the-fly computation.

A. Appendix

Listing A.1: OpenMP parallel region 1: Laplace

```fortran
200 subroutine laplacian( field , lap , num_halo , extend )
201     implicit none
202 
203     ! argument
204     real (kind=wp), intent(in) :: field(:, :, :)
205     real (kind=wp), intent(inout) :: lap(:, :, :)
206     integer , intent(in) :: num_halo , extend
207 
208     ! local
209     integer :: i, j, k
210 
211     !$omp parallel do default(none) shared(lap, k, j, i, num_halo, field, extend)
212     do k = lbound(field,3), ubound(field,3)
213     do j = lbound(field,2) + num_halo - extend, ubound(field,2) - num_halo + extend
214     do i = lbound(field,1) + num_halo - extend, ubound(field,1) - num_halo + extend
215         lap(i, j, k) = -4._wp * field(i, j, k)      &
216             + field(i - 1, j, k) + field(i + 1, j, k)  &
217             + field(i, j - 1, k) + field(i, j + 1, k)
218     end do
219     end do
220     end do
221     !$omp end parallel do
222 
223 end subroutine laplacian
```

Listing A.2: OpenMP parallel region 2: loops at line 168

```fortran
164 ! do forward in time step
165 !$omp parallel do default(none) shared(k, j, i, num_halo, out_field, in_field, alpha, tmp2_field)
166 do k = lbound(out_field,3), ubound(out_field,3)
167 do j = lbound(out_field,2) + num_halo, ubound(out_field,2) - num_halo
168 do i = lbound(out_field,1) + num_halo, ubound(out_field,1) - num_halo
169     out_field(i, j, k) = in_field(i, j, k) - alpha * tmp2_field(i, j, k)
170 end do
```

```
171 end do
172 end do
173 !$omp end parallel do
```

Listing A.3: OpenMP parallel region 3: loops at line 178

```
175 ! copy out to in in caes this is not the last iteration
176 if ( iter /= num_iter ) then
177     !$omp parallel do default(none) shared(k, j, i, num_halo, out_field, in_field)
178     do k = lbound(in_field,3), ubound(in_field,3)
179     do j = lbound(in_field,2) + num_halo, ubound(in_field,2) - num_halo
180     do i = lbound(in_field,1) + num_halo, ubound(in_field,1) - num_halo
181         in_field(i, j, k) = out_field(i, j, k)
182     end do
183     end do
184     end do
185     !$omp end parallel do
186 end if
```

Listing A.4: Code for pre-computation

```
!$omp parallel do private(k, j, i, tmp1_field, tmp2_field)
do k = lbound(out_field,3), ubound(out_field,3)
    do j = lbound(in_field,2) + num_halo - 1, ubound(in_field,2) - num_halo + 1
    do i = lbound(in_field,1) + num_halo - 1, ubound(in_field,1) - num_halo + 1
        tmp1_field(i, j) = -4._wp * in_field(i, j, k) + in_field(i-1, j, k) &
        + in_field(i+1, j ,k) + in_field(i, j-1, k) + in_field(i, j+1, k)
    end do
    end do

    do j = lbound(tmp1_field,2) + num_halo, ubound(tmp1_field,2) - num_halo
    do i = lbound(tmp1_field,1) + num_halo, ubound(tmp1_field,1) - num_halo
        tmp2_field(i, j) = -4._wp * tmp1_field(i, j) + tmp1_field(i-1, j) + &
        tmp1_field(i+1, j) + tmp1_field(i, j-1) + tmp1_field(i, j+1)
    end do
    end do

```

```fortran
    do j = lbound(out_field,2) + num_halo, ubound(out_field,2) - num_halo
    do i = lbound(out_field,1) + num_halo, ubound(out_field,1) - num_halo
        out_field(i, j, k) = in_field(i, j, k) - alpha * tmp2_field(i, j)
    end do
    end do

! copy out to in in caes this is not the last iteration
if ( iter /= num_iter ) then
    do j = lbound(in_field,2) + num_halo, ubound(in_field,2) - num_halo
    do i = lbound(in_field,1) + num_halo, ubound(in_field,1) - num_halo
        in_field(i, j, k) = out_field(i, j, k)
    end do
    end do
end if
end do
!$omp end parallel do
```

Listing A.5: Code for on-the-fly computation

```fortran
!$omp parallel do private(k, j, i)
do k = lbound(out_field,3), ubound(out_field,3)
do j = lbound(out_field,2) + num_halo, ubound(out_field,2) - num_halo
do i = lbound(out_field,1) + num_halo, ubound(out_field,1) - num_halo
    out_field(i, j, k) = &
        +  a1 * in_field(i  , j-2, k) &
        +  a2 * in_field(i-1, j-1, k) &
        +  a8 * in_field(i  , j-1, k) &
        +  a2 * in_field(i+1, j-1, k) &
        +  a1 * in_field(i-2, j  , k) &
        +  a8 * in_field(i-1, j  , k) &
        + a20 * in_field(i  , j  , k) &
        +  a8 * in_field(i+1, j  , k) &
        +  a1 * in_field(i+2, j  , k) &
        +  a2 * in_field(i-1, j+1, k) &
        +  a8 * in_field(i  , j+1, k) &
        +  a2 * in_field(i+1, j+1, k) &
        +  a1 * in_field(i  , j+2, k)
end do
end do

! copy out to in in caes this is not the last iteration
```

```
if ( iter /= num_iter ) then
    do j = lbound(in_field,2) + num_halo, ubound(in_field,2) - num_halo
    do i = lbound(in_field,1) + num_halo, ubound(in_field,1) - num_halo
        in_field(i, j, k) = out_field(i, j, k)
    end do
    end do
end if
end do
!$omp end parallel do
```

B. Bibliography

Hermanns, M. (2002). Parallel Programming in Fortran 95 using OpenMP.

URL: *https://www.openmp.org/wp-content/uploads/*$F95_{O}penMPv1_{v}2.pdf$