Scalable Direct-Iterative Hybrid Solver for Sparse Matrices on Multi-Core and Vector Architectures

Kenji Ono
Kyushu University
Fukuoka, Japan
keno@cc.kyushu-u.ac.jp

Toshihiro Kato
NEC Corporation
Tokyo, Japan
t-katou@ew.jp.nec.com

Satoshi Ohshima
Nagoya University
Nagoya, Japan
ohshima@cc.nagoya-u.ac.jp

Takeshi Nanri
Kyushu University
Fukuoka, Japan
nanri@cc.kyushu-u.ac.jp

ABSTRACT

In the present paper, we propose an efficient direct-iterative hybrid solver for sparse matrices that can derive the scalability of the latest multi-core, many-core, and vector architectures and examine the execution performance of the proposed SLOR-PCR method. We also present an efficient implementation of the PCR algorithm for SIMD and vector architectures so that it is easy to output instructions optimized by the compiler. The proposed hybrid method has high cache reusability, which is favorable for modern low B/F architecture because efficient use of the cache can mitigate the memory bandwidth limitation. The measured performance revealed that the SLOR-PCR solver showed excellent scalability up to 352 cores on the cc-NUMA environment, and the achieved performance was higher than that of the conventional Jacobi and Red-Black ordering method by a factor of 3.6 to 8.3 on the SIMD architecture. In addition, the maximum speedup in computation time was observed to be a factor of 6.3 on the cc-NUMA architecture with 352 cores.

CCS CONCEPTS

• **Computing methodologies** → *Vector / streaming algorithms*; *Shared memory algorithms.*

KEYWORDS

parallel cyclic reduction, cache bandwidth, line successive over-relaxation

ACM Reference Format:
Kenji Ono, Toshihiro Kato, Satoshi Ohshima, and Takeshi Nanri. 2020. Scalable Direct-Iterative Hybrid Solver for Sparse Matrices on Multi-Core and Vector Architectures. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020), January 15–17, 2020, Fukuoka, Japan.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3368474.3368484

HPCAsia2020, January 15–17, 2020, Fukuoka, Japan

ACM ISBN 978-1-4503-7236-7/20/01…$15.00
https://doi.org/10.1145/3368474.3368484

1 INTRODUCTION

In recent years, performance per power consumption has become important, even in the research field of high-performance computing, and multi-core/many-core processing with low-power consumption hardware is advancing. In architectures with many cores in the CPU, it is important to improve thread scalability in order to increase the unit performance. However, the memory wall problem, in which data transfer performance is relatively small compared to the core performance, becomes a significant bottleneck in writing high-performance code. Therefore, in order to improve the performance, it is widely known that using an algorithm that takes into account the data locality and effective reuse of the cache is essential.

In fluid and electromagnetic simulation, it takes a great deal of time to solve linear equations derived from a finite-difference method or other discretization techniques. The matrix of the linear system derived herein is a large-scale sparse matrix and is, in general, solved by iterative methods. Iterative methods yield a high B/F code with a small number of operations per loop and have a memory bound characteristic, and so often remain at an execution performance of several percent of the theoretical performance. Limited room for tuning and difficulty in adapting to recent architectures are also problematic. On the other hand, direct methods are suitable for low B/F architectures. There are very-high-performance modules, such as DGEMM, included in the LAPACK library. These modules are robust but have high memory requirements and are not suitable for solving large sparse matrices.

2 RELATED WORK

Conventionally, a method combining direct methods and iterative methods has been proposed to improve convergence when solving large-scale sparse matrices. Rajamanickam et al. [16] proposed the ShyLU framework for MPI+threads parallelism, which combines the Schur-complement-based iterative part and the existing packaged direct solvers. They reported the performance of their framework and focused on hybrid parallel performance rather than thread scalability. Agullo et al. [1] also proposed a hybrid approach based on the Schur complement with multi-threaded direct solvers in BLAS or LAPACK libraries, and reported its good performance on 1,728 cores for solving the 3D diffusion problem. This hybrid concept is not new and is also found in previous studies [4, 6, 21], which refer to the line successive over-relaxation (SLOR) method

that combines the successive over-relaxation (SOR) method and a tridiagonal system. The formed tridiagonal systems are solved by Gaussian elimination, more precisely by LU decomposition, also known as the Thomas algorithm in the field of computational fluid dynamics (CFD). In other words, the SLOR method includes LU decomposition inside the process of SOR iterations. The convergence rate of the SLOR method that uses line-wise relaxation is faster than that of the conventional point-wise SOR method [6]. Although the most efficient LU decomposition is used for the SLOR method, we have a choice regarding the direct solver. However, since the LU decomposition is a serial algorithm, it is not applicable for modern multi-core and many-core architectures.

In order to solve this problem, parallelizable LU decomposition algorithms based on cyclic reduction (CR) [2, 7], parallel cyclic reduction (PCR) [8], recursive doubling (RD) [19], and partitioning method [3, 10] have been rediscovered for GPGPU computation. Zhang et al. [22] studied the performance of three basic parallel algorithms (CR, PCR, and RD) and their hybrid variants for solving tridiagonal linear systems on a GPU. They reported that their GPU implementation achieved up to a 28x speedup over a sequential LAPACK solver and 12x speedup over a multi-threaded CPU solver.

Cyclic Reduction is two-step algorithm. In the first step, CR divides a set of linear equations into two independent sets of linear equations, the size of each being half of the original size, and extracts parallelism. This procedure is repeated until the sets have only one equation. The second step is backward substitution. In both steps, the parallel granularity decreases during the iteration in each step, which leads to a performance degradation in parallel computation. Parallel Cyclic Reduction is the parallel version of CR algorithm. Although the amount of arithmetic becomes larger than for the CR algorithm, PCR provides high parallelism.

The contribution of the present paper is two fold. First, we propose thread-scalable PCR combined with the SOR method as a direct-iterative hybrid solver. Second, we apply the proposed method to modern multi-core and vector architectures and clarify its properties. The remainder of the present paper is organized as follows. Section 2 introduces the proposed hybrid SLOR-PCR method. Section 3 describes the target architectures and measurements in the present study. Section 4 describes the results of the performance evaluation, and Section 5 summarizes the present study and describes future research.

3 DIRECT-ITERATIVE HYBRID SOLVER

Consider the following three-dimensional (3D) steady diffusion problem: $\nabla^2\phi = \Psi$, where ϕ is some physical variable, and Ψ is its source term. When this equation is discretized by the second-order finite-difference approximation on a structured grid system, a linear system, the coefficient matrix of which is a seven-diagonal sparse matrix (1), is obtained:

$$\begin{aligned}&\phi_{k,i-1,j} + \phi_{k,i+1,j} + \phi_{k,i,j-1} \\ &\quad + \phi_{k,i,j+1} + \phi_{k-1,i,j} + \phi_{k+1,i,j} - 6\,\phi_{k,i,j} = h^2\,\Psi_{k,i,j}\,,\end{aligned} \tag{1}$$

where h is the grid width and indices i, j, k correspond to coordinate axes in the x, y, and z directions. In the actual problem, the grid width changes locally, but, here, the grid width is considered to be equally spaced. Of course, since the proposed algorithm can be applied to the problem of anisotropic spacing, generality is not lost.

3.1 Line Successive Over-relaxation Method

We can form the m-th SOR iteration process from (1) considering direct solution of a small tridiagonal system in the z direction:

$$\begin{aligned}&-\frac{1}{6}\hat{\phi}^{\,m}_{k-1,i,j} + \hat{\phi}^{\,m}_{k,i,j} - \frac{1}{6}\hat{\phi}^{\,m}_{k+1,i,j} \\ &= \frac{1}{6}\left(\phi^{m-1}_{k,i-1,j} + \phi^{m-1}_{k,i+1,j} + \phi^{m-1}_{k,i,j-1} + \phi^{m-1}_{k,i,j+1} - h^2\,\Psi_{k,i,j}\right),\end{aligned} \tag{2}$$

Here, $\hat{\phi}$ is an updated value obtained by solving a tridiagonal system. The following relaxation process is written to introduce the acceleration factor ω:

$$\begin{aligned}\Delta\phi^{\,m}_{k,i,j} &= \hat{\phi}^{\,m}_{k,i,j} - \phi^{\,m-1}_{k,i,j} \\ \phi^{\,m}_{k,i,j} &= \phi^{\,m-1}_{k,i,j} + \omega\,\Delta\phi^{\,m}_{k,i,j} \quad ,\end{aligned} \tag{3}$$

Since the above procedure is based on the SOR method, the calculation between (3) and the RHS of (2) yields recursion, which prevents both parallelization and vectorization. In order to remedy this situation, various ordering methods have been proposed, including hyper-plane ordering [5] and multi-color ordering [15]. In the present paper, we introduce simple Red-Black ordering in order to retain the parallel granularity. Algorithm 1 shows Red-Black ordering relaxation method.

3.2 Parallel Cyclic Reduction Direct Solver

We consider the following system of n linear equations $A\boldsymbol{x} = \boldsymbol{d}$ from (2), where A is a tridiagonal matrix

$$A = \begin{pmatrix} 1 & c_1 & & & & \\ a_2 & 1 & c_2 & & 0 & \\ & a_3 & 1 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & 0 & & \ddots & 1 & c_{n-1} \\ & & & & a_n & 1 \end{pmatrix}$$

in z direction. Here, the components are $a_k = c_k = -1/6$ and vector $\boldsymbol{d}$ can be rewritten as

$$\begin{aligned}d_k = \frac{1}{6}\Big(&\phi^{m-1}_{k,i-1,j} + \phi^{m-1}_{k,i+1,j} \\ &+\phi^{m-1}_{k,i,j-1} + \phi^{m-1}_{k,i,j+1} - h^2\,\Psi_{k,i,j}\Big) + \beta_k\,,\end{aligned} \tag{4}$$

where the term β_k includes the boundary conditions. Figure 1 shows the computational region to be solved in the z direction. The boundary conditions are given at both ends of the line, i.e., points k=0 and k=n+1. In this case, we apply the Dirichlet boundary condition at both ends, but the following procedure can be also applied to the Neumann condition by setting the coefficient values as $a_0 = c_{n+1} = 0$. The first line of the linear system can be written as $x_1 + c_1x_2 = d_1 - a_1x_0$, and the last line can be written as $a_nx_{n-1} + x_n = d_n - c_nx_{n+1}$. Therefore,

$$\beta_k = \begin{cases} -a_1 x_0 & (k=1) \\ -c_n x_{n+1} & (k=n) \\ 0 & (\text{others}) \end{cases} \quad (5)$$

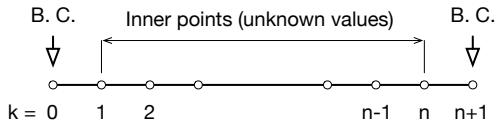

Figure 1: Computational region and index of a tridiagonal system. The boundary conditions are given at points $k = 0$ and $k = n + 1$.

In the PCR, first, a reduction operation is performed by changing an original set of n equations into two sets of n/2 equations. The reduction operation derives the relationship between every other points from the relationship between adjacent points. In other words, as the reduction stage progresses, the relationship with the adjacent points is cut off, the dependency disappears, and parallelization becomes possible. By introducing the number of the p-th reduction as a superscript (p), the reduction process becomes as follows:

$$\begin{aligned} a_k^{(0)} x_{k-1} + x_k + c_k^{(0)} x_{k+1} &= d_k^{(0)} \\ a_k^{(1)} x_{k-2} + x_k + c_k^{(1)} x_{k+2} &= d_k^{(1)} \\ a_k^{(2)} x_{k-4} + x_k + c_k^{(2)} x_{k+4} &= d_k^{(2)} \\ \vdots \qquad\qquad\qquad\quad \vdots &\quad \vdots \\ a_k^{(p)} x_{k-2^p} + x_k + c_k^{(p)} x_{k+2^p} &= d_k^{(p)} \end{aligned}$$

The reduction procedure and the exact form of coefficients $a_k^{(p)}$, $c_k^{(p)}$ and $d_k^{(p)}$ are described as aw_k, cw_k and dw_k, respectively, in Algorithm 2. After N_p steps, where N_p is the smallest integer value to satisfying $2^{N_p} \geq n$, we obtain n sets of independent equation that is not coupled, therefore, the equation can be calculate fully in parallel. First, we start from one set of n coupled equations. At this time, all modified coefficients become $a_k^{(N_p)} = c_k^{(N_p)} = 0$, then the solution $x_k = d_k^{(N_p)}$ is immediately obtained.

In the PCR algorithm, however, the computational cost of the main part becomes $14\,N_p \approx 14\,log_2 n$ per point (see lines 9–20 in Algorithm 2, which requires 14 floating point multiplication, addition, and reciprocal operations). The total computational cost of SLOR-PCR becomes $N_x\,N_y\,Nz\,(14N_p + 11)$, because (4) requires 6 flops (floating point operations) and lines 9–15 in Algorithm 1 requires 5 flops, whereas the total computational cost of Red-Black ordering SOR is $11\,N_x\,N_y\,Nz$ (lines 9–15 in Algorithm 1 and lines 5–7 in Algorithm 2).

3.3 Our Implementation

We improve the performance of the naïve implementation of the PCR Algorithm 2. In our iterative solver implementation, the shape

Algorithm 1: Red-Black ordering SLOR-PCR method.

Input: Nx, Ny, Nz, *Iteration_Max*, ω, ϵ
Output: residual, itr
Data: array real (0 : Nz+1) : d, dw
array real (-1 : Nz+2, -1 : Nx+2, -1:Ny+2) : x

```
1  begin
2    for itr ← 1 to Iteration_Max do
3      residual ⟵ 0;
4      for color ← Red to Black do
5        for j ← 1 to Ny do
6          for i ← 1 to Nx do
7            if (i, j) ∈ color then
8              pcr (color, residual, d, dw)
9              for k ← 1 to Nz do
10               update value by (3);
11               x_p ⟵ x_{k,i,j};
12               Δp ⟵ ω (dw_k − x_p);
13               x_{k,i,j} ⟵ x_p + Δp;
14               residual ⟵ residual + Δp^2;
15             end
16           end
17         end
18       end
19     end
20     if residual < ε then break;
21   end
22 end
```

and loop structure of the 3D array appears as follows in Fortran format:

```
1  real, dimension(-1:Nz+2, -1:Nx+2, -1:Ny+2) :: a
2  do j=1,Ny
3  do i=1,Nx
4  do k=1,Nz
5    a(k,i,j) = ...
6  end do
7  end do
8  end do
```

where Nz is the problem size in the z direction and the system size of (2), and Nx and Ny are the problem sizes in the x and y directions, respectively. We set the z direction as the innermost index in order to calculate PCR efficiently.

Figure 2 illustrates the reduction process of PCR. As the reduction step progresses, the distance between reference points becomes larger. In the p-th step, the distance becomes 2^{p-1}. Therefore, when calculating the value at point k, the reference points are 2^{p-1} apart. Since the values at three points k and $k \pm 2^{p-1}$ are required for the calculation in the reduction process, in order to generate high-performance code, we need to let the compiler know that the loop can be compiled not by stride or gather memory access but rather by three streams of consecutive memory access. As a result, we

Algorithm 2: Procedure of Naïve PCR calculation.

```
   Input: color, residual, d, dw
   Output: residual, dw
   Data: array real (0 : Nz+1) :   a, c, d, aw, cw, dw
1  begin
2      for k ← 1 to Nz do
3          set coefficients a_k and c_k;
4      end
5      for k ← 1 to Nz do
6          calculate source of tridiagonal system by (4);
7      end
8      set boundary condition by (5);
9      for p ← 1 to N_p do
10         s ⟵ 2^(p−1);
11         for k ← 1 to Nz do
12             e ⟵ 1.0/(1.0 − a_k c_(k−s) − c_k a_(k+s));
13             aw_k ⟵ −e a_k a_(k−s);
14             cw_k ⟵ −e c_k c_(k+s);
15             dw_k ⟵ e (d_k − a_k d_(k−s) − c_k d_(k+s));
16         end
17         for k ← 1 to Nz do
18             a_k ⟵ aw_k;   c_k ⟵ cw_k;   d_k ⟵ dw_k;
19         end
20     end
21 end
```

can expect 9 SIMD streams (a_k, c_k, d_k, $a_{k\pm 2^{p-1}}$, $c_{k\pm^{p-1}}$, $d_{k\pm^{p-1}}$ of lines 15–18 in Algorithm 3).

This requires modification of the source code so that the compiler can interpret our intention. The naïve implementation of Algorithm 2 requires an if-branch or min/max operation such as $max(k-2^{p-1}, 0)$ for the index calculation of array a, c, d to avoid access outside of the array. However, this implementation may cause stride or gather instructions, which cannot use the full potential of a SIMD or vector unit, and hence can cause low performance. In order to make issue more efficient SIMD instructions, we modify the size of the array from $a\,(0 : Nz+1)$ to $a\,(-2^{N_p-2} : Nz+1+2^{N_p-2})$ in order to avoid complicated index calculation and enhance compiler optimization. Note that this enlarged array implementation requires zero initialization at every PCR calculation (see line 3 in Algorithm 3).

One of the most important points in the PCR algorithm is to use a one-dimensional array. The assumed array length Nz ranges from 512 to 2,048 in practice, which is 4kB to 16kB in double precision, small enough for the L2 or last level cache (LLC) size. Moreover, in the main part of the PCR calculation, it is expected that the fastest L1 cache can be used. As described above, the main part requires 9 SIMD streams from three arrays for loading data. Assuming $Nz = 1,024$, 24kB is required, which is sufficiently smaller than the L1 cache size (32kB) of a typical modern core. Thus, it is expected that the one-dimensional array data that resides inside the cache reduces the memory pressure, which causes the PCR code to run fast.

Although the PCR algorithm provides good parallel granularity, we should pay attention to the accuracy. When a reduction operation is repeated, the absolute value of the non-diagonal element a and c becomes exponentially smaller than that of the diagonal element, causing a degradation in calculation accuracy. In order to alleviate this problem, we introduced a technique that stops the reduction operation at 2^{N_p-1} steps and directly inverts the 2×2 matrix of the final step [22]. This improvement is also effective in performance because the inversion calculation of the 2×2 small matrix gives high arithmetic counts and high L1 cache use, which is similar to the calculation of the main part of the PCR. The improved procedure is shown in Algorithm 3. After 2^{N_p-1} steps calculation, in the PCR algorithm, the $Nz/2$ sets of equations are obtained. More specifically, the element of index k and the element apart from k by 2^{N_p-1}, i.e. $k + 2^{N_p-1}$, form 2×2 matrices to be inverted. This procedure is described at lines 24 – 34 in Algorithm 3.

Figures 3 and 4 show the results for the naïve implementation and our implementation on Skylake-SP and SX-Aurora respectively (See the following section for the specifications of the computer). Our implementation could significantly improve the performance for both SIMD and vector architectures by a factor of 2.3.

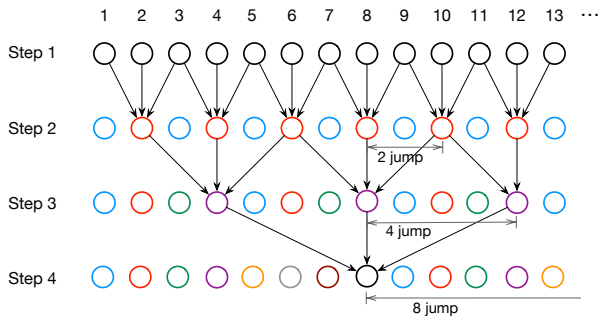

Figure 2: Reduction steps in PCR algorithm. Colors indicate different sets of equations in each step.

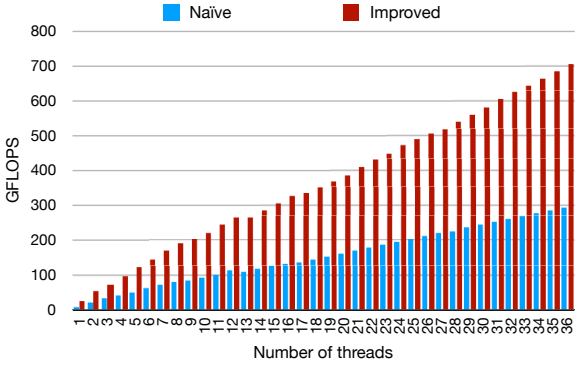

Figure 3: Performances of naïve and modified SLOR-PCR implementations on Skylake-SP. The problem size is 128×128×512.

Algorithm 3: Procedure of improved PCR calculation.

Input: color, residual, d, dw
Output: residual, dw
Data: array real $(0:\mathrm{Nz}+1)$: aw, cw, dw
array real $(-2^{N_p-2}:\mathrm{Nz}+1+2^{N_p-2})$: a, c, d

begin
  for $k \longleftarrow -2^{N_p-2}$ **to** $Nz+1+2^{N_p-2}$ **do**
    $a_k \longleftarrow 0;\quad c_k \longleftarrow 0;\quad d_k \longleftarrow 0;$
  end
  for $k \longleftarrow 1$ **to** Nz **do**
    set coefficients a_k and c_k;
  end
  for $k \longleftarrow 1$ **to** Nz **do**
    calculate source of tridiagonal system by (4);
  end
  set boundary condition by (5);
  for $p \longleftarrow 1$ **to** $N_p - 1$ **do**
    $s \longleftarrow 2^{p-1}$;
    for $k \longleftarrow 1$ **to** Nz **do**
      $e \longleftarrow 1.0/(1.0 - a_k\, c_{k-s} - c_k\, a_{k+s});$
      $aw_k \longleftarrow -e\, a_k\, a_{k-s};$
      $cw_k \longleftarrow -e\, c_k\, c_{k+s};$
      $dw_k \longleftarrow e\,(d_k - a_k\, d_{k-s} - c_k\, d_{k+s});$
    end
    for $k \longleftarrow 1$ **to** Nz **do**
      $a_k \longleftarrow aw_k;\quad c_k \longleftarrow cw_k;\quad d_k \longleftarrow dw_k;$
    end
  end
  for $k \longleftarrow 1$ **to** 2^{N_p-1} **do**
    $cc1 \longleftarrow c_k;$
    $aa2 \longleftarrow a_{k+2^{N_p-1}};$
    $f1 \longleftarrow d_k;$
    $f2 \longleftarrow d_{k+2^{N_p-1}};$
    $jj \longleftarrow 1.0/(1.0 - aa2 * cc1);$
    $dd1 \longleftarrow (f1 - cc1 * f2) * jj;$
    $dd2 \longleftarrow (f2 - aa2 * f1) * jj;$
    $dw_k \longleftarrow dd1;$
    $dw_{k+2^{N_p-1}} \longleftarrow dd2;$
  end
end

Figure 5 illustrates the breakdown of time cost of Algorithm 3. The most time-consuming part is the calculation of PCR. Second, since 3D array ϕ requires memory access to MMU, the calculation time for the source term is long.

4 TARGET ARCHITECTURES, ENVIRONMENT AND MEASUREMENTS

The proposed SLOR-PCR solver was implemented on three architectures of a shared memory machine. The computation environment and software are shown in Table 1. Intel Xeon Gold 6140 (hereafter, we refer as Skylake-SP) is a standard two-socket processor, and

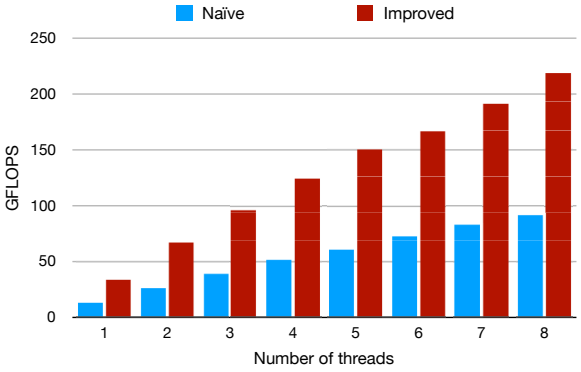

Figure 4: Performance of naïve and modified SLOR-PCR implementations on SX-Aurora. The problem size is 128×128×512.

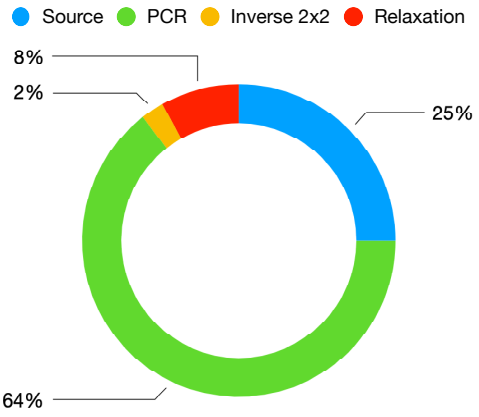

Figure 5: Time breakdown of SLOR-PCR algorithm.

each processor has 18 cores. After the Skylake architecture, since Intel has introduced control of power consumption, the operating frequency varies depending on the operating temperature [9]. Here, as a primary cache, L2 has 1MB locally for each core and a transfer rate of 64 Bytes/cycle.

The SGI UV 300 computer system is based on a cache-coherent non-uniform memory access (cc-NUMA) architecture and serves a 12-TB memory space in a single system. As shown in Figure 6, the chassis of the UV 300 consists of four-socket Intel Xeon (refer as UV 300 or Broadwell-EP) CPUs, and each CPU has 22 cores and two memory sockets. The UV 300 system has four chassis, which gives 352 cores in total. Two HARP communication chips used for the chassis provide high-speed connection between CPUs inside the chassis as well as connection to the external chassis using the NUMAlink 7 interconnect architecture [18].

The NEC SX-Aurora TSUBASA (refer as SX-Aurora) is a new vector architecture with an LLC. The SX-Aurora consists of a vector host (VH), which is an x86 based system, and a vector engine (VE). The VE has eight cores with vector processing units with high

Table 1: Specifications of evaluation systems.

	Intel Xeon Skylake-SP	SGI UV300 Broadwell-EP	SX-Aurora A100-1
Architecture	Gold 6140	E7-8880 v4	Type 10C
Socket (CPUs)	2	4 × 4 Chassis	1
Cores per CPU	18	22	8
Frequency (GHz)	2.3	2.2	1.4[1]
Peak[2] (GFLOPS)	5,298[3]	24,780	4,300
MMU size (GB)	384	12,000	24
MMU BW (GB/s)	255.9	1,360	750
L1 cache[4] (KB)	32	32	–
L2 cache[4] (KB)	1,024	256	–
L3 cache (MB)	24.75	55	16[6]
L3 BW (GB/s)	147[5]	–	332[6]
OS	RHEL 7.4	RHEL 7.3	CentOS 7.4
C++ compiler	Intel 18.0	Intel 18.0	nc++ 2.3.0
Fortran compiler	Intel 18.0	Intel 18.0	nfort 2.3.0

memory bandwidth. The theoretical bandwidth between the LLC and a core is 332 GB/s (2,600 GB/s in aggregate).

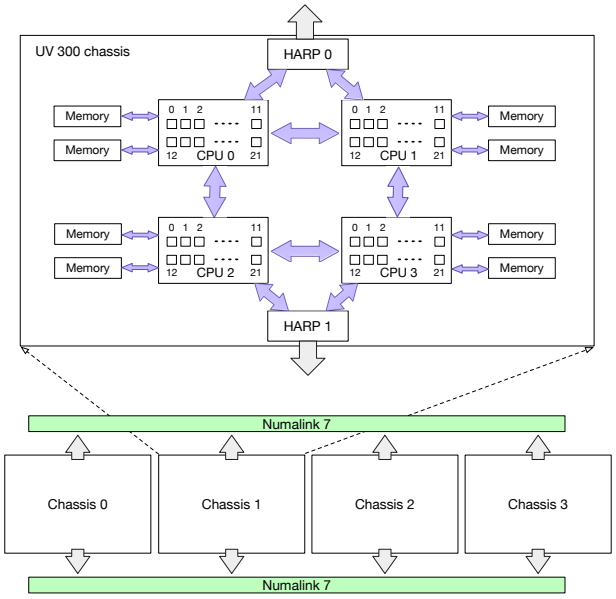

Figure 6: Functional block diagram of the SGI UV 300 system chassis. Four chassis are connected by Numalink7 via HARP modules [18].

[1] Vector engine (VE).
[2] In the case of single precision. In the case of double precision, the score becomes half.
[3] In the case of AVX-2 mode and base frequency 2.3GHz. The operating frequency depends on the temperature of the running CPU in the range of 2.3–3.0 GHz.
[4] Per core.
[5] In the case of a base frequency of 2.3 GHz. The operating frequency varies in the range of 2.3–3.0 GHz for the L2 cache.
[6] Last level cache.

4.1 Timing Measurement

We use our own portable performance monitor library PMlib [14] to measure run-time performance. PMlib provides an avenue for reporting the arithmetic/application workload that is manually counted from a source code, as well as the actually executed system workload. PMlib also provides detailed utilization reports of processor-specific hardware including the categorized SIMD instruction statistics, the layered cache hit/miss rate, and the effective memory bandwidth, which are captured via hardware performance counters (HWPC) through the PAPI interface [20]. Using PMlib, users can conduct a synthetic analysis of application performance, and obtain useful feedback for further optimized execution of applications [12]. We used HWPCk through PAPI to count floating operation for Skylake-SP and UV 300 systems.

We also use the vendor provided trace tool "`ftrace`" command for SX-Aurora. The `ftrace` command displays performance information from sampled information at run time, such as vector-unit-related information, vector load hit ratio at LLC, execution time, instruction count, FLOP count, MFLOPS, L1 cache miss time, and memory size used.

5 PERFORMANCE EVALUATION

5.1 Kernel Code

In order to measure the performance of the proposed method, we use a kernel code from a practical application to solve a 3D steady diffusion equation on a rectangle computational domain. The steady diffusion equation is discretized by the finite-difference method on an orthogonal grid system and is transformed into a linear system with a seven-diagonal matrix, as shown in (1), where $\Psi = 0$. The corresponding exact solution is:

$$\phi(x,\ y,\ z) = \frac{1}{\sin(\sqrt{2}\pi)} \sin(\pi x)\sin(\pi y) \left\{ \sinh(\sqrt{2}\pi z) - \sinh\left(\sqrt{2}\pi(z-1)\right) \right\},$$

with boundary condition

$$\phi(x,\ y,\ z) = \begin{cases} \alpha \sin(\pi x)\sin(\pi y) & \text{on} \quad (x,\ y,\ 0) \\ \sin(\pi x)\sin(\pi y) & \text{on} \quad (x,\ y,\ 1) \\ 0 & \text{on} \quad \text{other boundaries} \end{cases}, \quad (6)$$

where α is a factor as an index of ease of solving a problem. In other words, α gives the magnitude of the gradient of the variable ϕ. A large value of α indicates that the problem is difficult to solve. Normally, α is set to one, and this parameter is used for the discussion in Section 5.5.2.

The kernel code [13] is written in C++, and the main calculation module is coded in Fortran90. Various iterative solvers, such as Jacobi, SOR, Red-Black SOR, and BiCGSTAB in addition to the present SLOR-PCR solvers, are implemented in this application. The problem sizes we choose range from $32\times32\times64$ to $256\times256\times4,096$, depending on the memory capacity.

5.2 Skylake-SP

In this first Skylake-SP case, we choose a problem size of $128\times128\times512$ so that the size of a variable in single precision becomes greater

than the L2 size of 18MB. First, we investigate the memory bandwidth[1], as shown in Figure 7 using the stream benchmark [11]. The measured performance is almost 70% of the theoretical bandwidth.

We compared the performance of the proposed SLOR-PCR solver and the conventional Jacobi and Red-Black SOR solvers in single precision. Figure 8 shows that SLOR-PCR has excellent scalability. On the other hand, the conventional iterative solvers show memory bound behavior similar to the performance curve of compact affinity in Fig. 7.

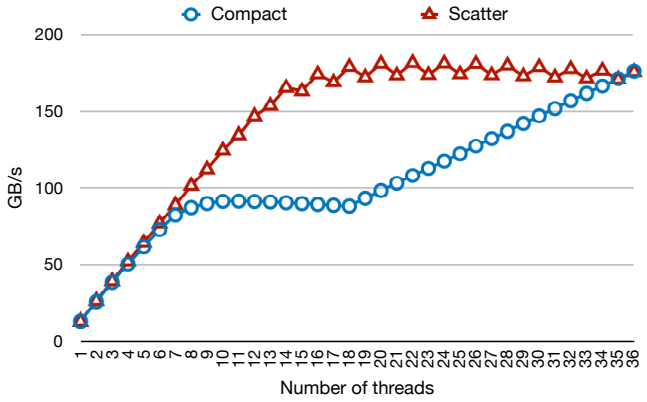

Figure 7: Memory bandwidth of Skylake-SP. The performance is compared for different affinities of compact and scatter.

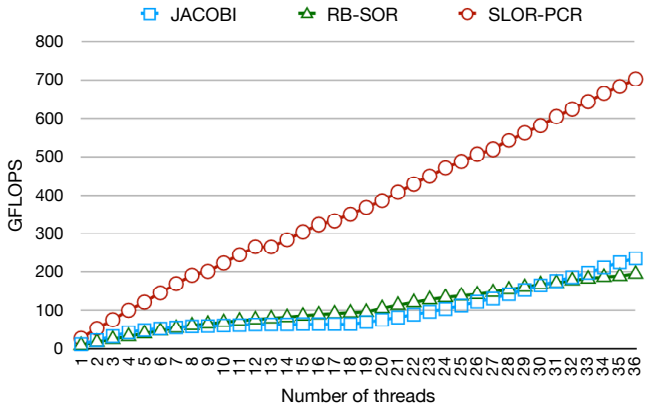

Figure 8: Solver performance in single precision on Skylake-SP. The affinity is set as compact.

Figure 9 reveals the utilization of the L1 and L2 caches for 36 cores based on the information from PMlib. Here, the L1 cache hit rate is defined as the number of times that data is present in the L1 cache or LFB (Line Fill Buffer), so that processing is possible between the L1 cache and registers, divided by the total number of data access events resulting from Load/Store instructions. The L2 cache hit rate is the number of times that data is present in the L2 cache, and processing is possible between the L2 cache and the registers, but the L1 cache is not being hit, again divided by the total number of data access events. The SLOR-PCR solver exhibits high utilization of the L1 cache and low utilization of the L2 cache. This tells us that the SLOR-PCR solver can effectively reuse the data in the L1 cache. On the other hand, RB-SOR exhibits relatively low utilization of the L1 cache but high utilization of the L2 cache. In addition, RB-SOR exhibits unbalanced L2 utilization because RB-SOR divides the innermost loop into two groups, which causes a load imbalance and the L1 cache misses. As designed, the SLOR-PCR solver designates high L1 cache usage by introducing a one-dimensional array in the cache, which turns out to have a significant impact on performance gain.

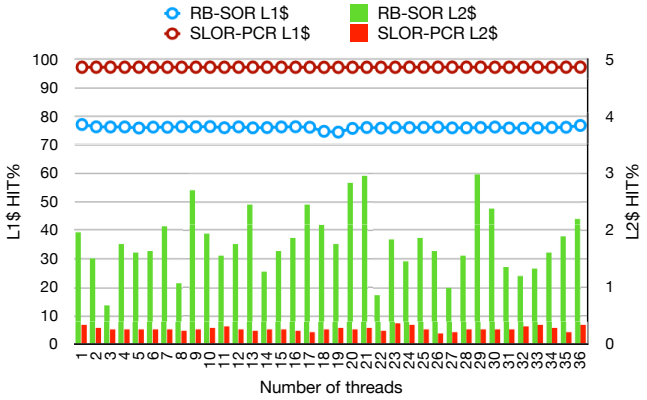

Figure 9: Cache hit rate for 36 cores on Skylake-SP.

5.3 UV 300

In the second case, we investigate the proposed solver in the cc-NUMA environment. The memory performance of the UV 300 exhibits quite different behavior from the Skylake-SP case, as shown in Figure 10, due to the cc-NUMA mechanism and very high memory performance of approximately 88% of the theoretical peak bandwidth.

First, the scalability to the problem size was investigated. Figure 11 shows the strong scaling performance for which the problem size is varied from 32×32×512 to 384×384×512. In this case, data for problems with sizes of less than or equal to 256 remains inside the L3 cache. Since OpenMP directive is applied for the loop in the x and y directions, in the cases of 32 and 64, it can be clearly seen that the parallel granularity is insufficient. The case of 128 shows relatively good performance. The remaining two cases demonstrate excellent scalability. A performance jump occurs at the boundaries of the CPU and/or chassis due to remote memory access of cc-NUMA.

Figure 12 illustrates the measured results when the size in the z direction is varied from $192 \times 192 \times 64$ to $192 \times 192 \times 2{,}048$ in order to examine the effect of the vector length of the innermost loop. In this case, problems with sizes of less than or equal to 1,024 remains inside the L3 cache, but for a size of 2,048, it does not. We can see two different types of behaviors, i.e., the cases of 64 and 128 do not scale well, whereas the cases of 512 and 1,024 scale well thanks to a sufficient vector length. The problem size of 256 is moderate.

[1]Compile options : `icc -xHOST -O3 -ipo -no-prec-div -fp-model fast=2 -qopenmp stream.c`

Although the case of 2,048 scales, the performance is low. Since PCR calculation requires 6 streams in lines between 14 and 22 in Algorithm 3, the size when Nz=2,048 is out of the L1 cache size.

From the above results, we choose the problem size of 192 × 192 × 512 as a moderate problem size for the evaluation on UV 300. Figure 13 shows the results of the iterative solvers. Similar to the case on Skylake-SP, the performance of the PCR solver scales, whereas that of Jacobi and Red-Black ordering SOR is limited by the memory bandwidth.

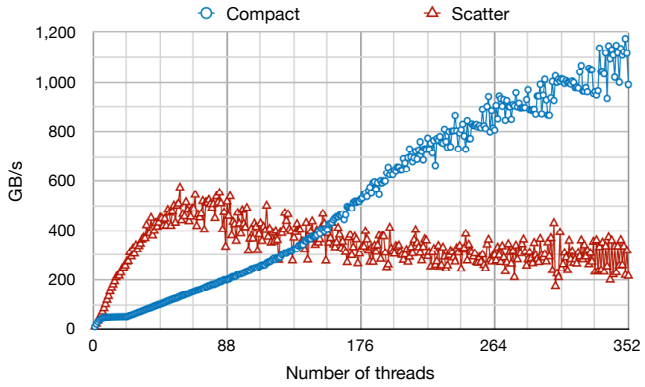

Figure 10: Measured memory bandwidth on UV 300.

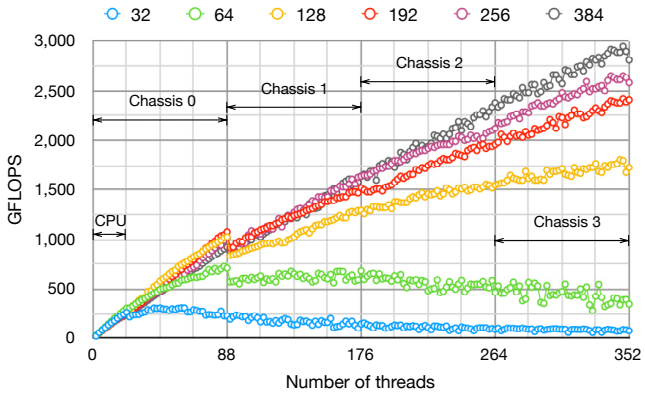

Figure 11: Strong scaling performance of the SLOR-PCR solver for a problem size in the x and y directions on UV 300. For example, 32 indicates 32 × 32 × 512. The affinity is set as compact.

5.4 NEC SX-Aurora TSUBASA

The last case is the performance on the vector architecture. Strong scaling performance was investigated as shown in Figure 14. The problem sizes of 32 and 48 clearly lack granularity of parallelization. We choose the problem size as 192 × 192 × 2,048 to allow a sufficient vector length in the z direction.

Figure 15 shows the measured memory bandwidth and the performance of the solvers. The SX-Aurora provides a high memory performance of over 95% for the theoretical peak.

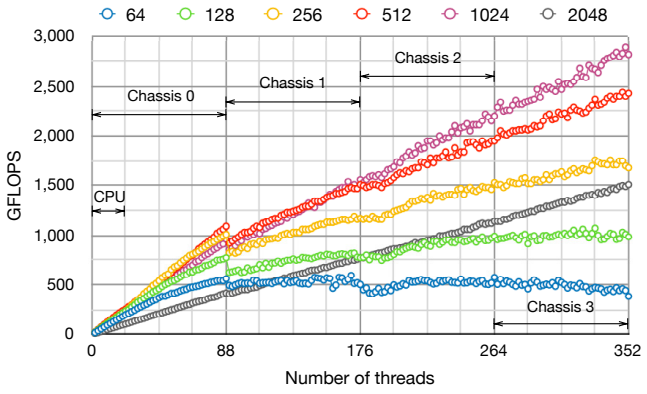

Figure 12: Strong scaling performance of the SLOR-PCR solver for a problem size in the z direction on UV 300. For example, 64 indicates 192 × 192 × 64. The affinity is set as compact.

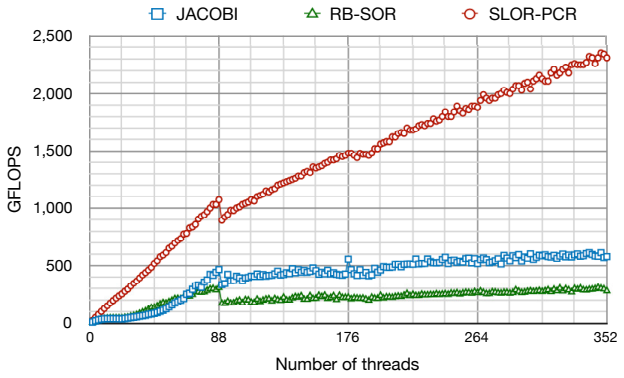

Figure 13: Performance of iterative solvers on UV 300. The problem size is 192 × 192 × 512, and the affinity is set as compact.

For this vector computer case, the problem size was examined, and a size of 192 × 192 × 2, 048 was selected. Although this size is out of the size of the LLC, the high memory bandwidth enables us to select a larger size than the Skylake-SP or Broadwell-EP architectures. The observed performance of the SLOR-PCR solver is dominant compared to the other solvers shown in Fig. 15. Figure 16 shows the hit ratio of the LLC for the vector load instruction. The hit ratio of Jacobi and Red-Black ordering SOR is less than under 50%, while SLOR-PCR retmains over 93% for all threads. This means that the PCR algorithm intensively uses the LLC and is no longer memory bound but rather LLC bound.

5.5 Discussion

5.5.1 Peak Performance. We summarize the achieved performance for three architectures in Table 2. The conventional iterative methods, such as Jacobi and Red-Black ordering SOR, ranged from 1.1%

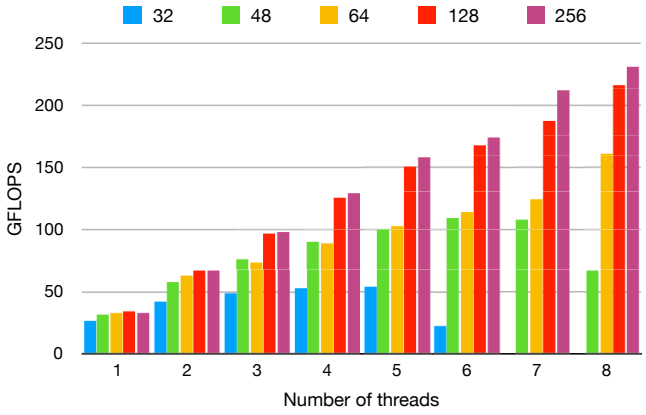

Figure 14: Performance of SLOR-PCR solver for different problem sizes on the SX-Aurora. The problem size of 32 indicates $32 \times 32 \times 512$.

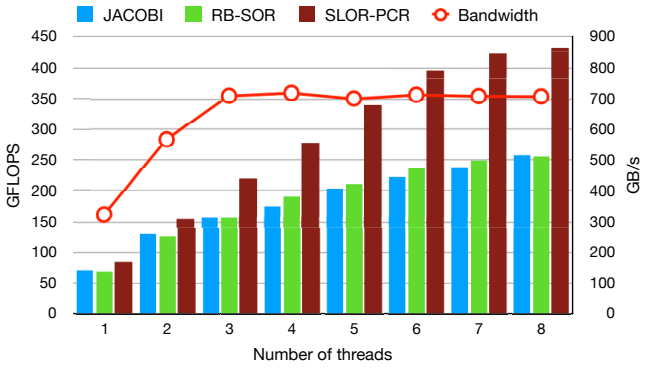

Figure 15: Performance of iterative solvers and memory bandwidth on SX-Aurora. The problem size is $192 \times 192 \times 2,048$.

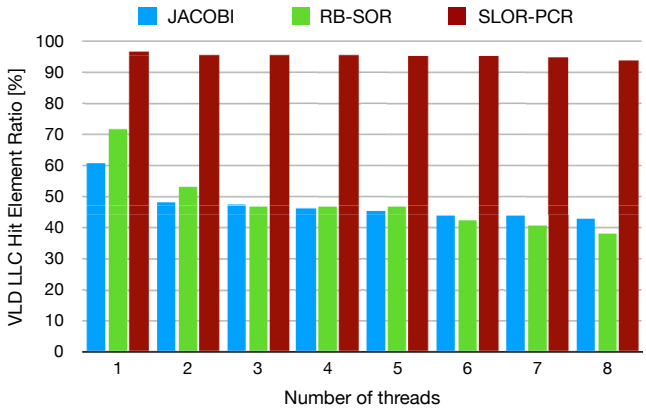

Figure 16: Vector load hit rate of last level cache on SX-Aurora. The problem size is $192 \times 192 \times 2,048$.

to 3.4% of the peak performance on multi-core NUMA architectures, whereas SX-Aurora achieved a relatively high 6% of the peak performance thanks to the high memory bandwidth and efficient vector units.

Table 2: Achieved performance of three solvers on different architectures. Numeric value in parentheses indicate the ratio of measured values to the peak performance.

Architecture	Skylake-SP	Broadwell-EP	SX-Aurora
Jacobi	235 (4.4%)	578 (2.3%)	259 (6.0%)
RB-SOR	194 (3.7%)	278 (1.1%)	256 (5.9%)
SLOR-PCR	704 (13.3%)	2,310 (9.3%)	432 (10.0%)
Speedup	3.6x	8.3x	1.7x

Among the iterative solvers, the SLOR-PCR method is superior to the conventional solvers and achieved 9.3% to 13/3% of the peak performance on all architectures evaluated herein. Based on the results in Sections 5.2, 5.3 and 5.4, the reason for the performance improvement is the effective use of the cache, as expected based on the design and implementation of the SLOR-PCR algorithm.

Comparison of SLOR-PCR and RB-SOR, which are base solvers of the proposed method, shows a speedup by a factor of 1.7 to 8.3 for the evaluated architectures.

5.5.2 Execution time. In this section, we consider the calculation time. Thus far, we have determined that SLOR-PCR demonstrates higher performance than other methods but requires a high computational cost at the same time. The computation time is estimated by the following basic relationship:

$$\text{Time} \propto \frac{\text{Iteration} \times \text{Total flop counts per iteration}}{\text{GFLOPS}}. \quad (7)$$

Let us examine in detail the behavior for 352 cores on the UV 300 system. In Table 3, "Flop counts per point" is the floating operation count in a loop. Although we explained that the flop count of Red-Black ordering SOR is 11 in Section 3.2, Red-Black ordering SOR requires a flop count of 18 because the numeric values in the table include additional utility operations. "Relative cal. cost" is calculated as "Flop counts per point" times "Number of iterations", where $N_p = 9$, and is normalized by the value for the Red-Black SOR. "Relative performance" is the relative GFLOPS value based on Red-Black SOR among the three iterative methods. Finally, "Relative time cost" was calculated based on these two values. The calculated value of "Relative time cost" was approximately the same as the ratio of the measured time. As a result, there is almost no difference between the relationship (7) and the measured values. Consequently, in this case, the execution time of SLOR-PCR is approximately half that of RB-SOR.

Next, we examine the relationship between the matrix to be solved and its computation time for three solvers. Figure 17 shows the computation time to solve linear systems that have different properties, i.e., the ease of solving a matrix. In Fig. 17, the horizontal axis indicates an index that expresses the ease of solving a given matrix, and this index is related to the coefficient α in (6). As a physical meaning, α controls the magnitude of the gradient value $\nabla\phi$ in the z direction. Of course, the high gradient makes solving

Table 3: Execution times for iterative solvers on UV 300 for 352 cores. The problem size is 192 × 192 × 512. In this case, N_p=9 and α=1.0.

Iterative method	Jacobi	RB-SOR	SLOR-PCR
Flop counts per point	18	18	14 (N_p-1)+21
Number of iterations	1,884	1,372	669
Time [s]	1.146	1.744	0.852
GFLOPS	578	278	2,310
Relative cal. cost	1.37	1.00	3.61
Relative performance	2.08	1.00	8.31
Relative time cost	0.65	1.00	0.43

the matrix difficult. The computation time increases as the value of α increases and the difference in the execution time between the SLOR-PCR method with high convergence and other methods with relatively low convergence becomes large. Thus, if we apply SLOR-PCR to a problem that is difficult to solve, then the SLOR-PCR method makes the computation time shorter. SLOR-PCR has the shortest calculation time among the evaluated solvers and has a speedup of a factor of 6.2 for the index of 100.0. In contrast, if we apply SLOR-PCR to a problem that is easy to solve, the reduction of the computation time is small. Although specific examples are not shown due to space limitations, the calculation time for SLOR-PCR may be long depending on the index of the ease of solving. In such a situation, it is expected that auto-tuning technology [17], which automatically chooses the most appropriate iterative method and switches to this method during calculation, will play an important role for in high-performance computing.

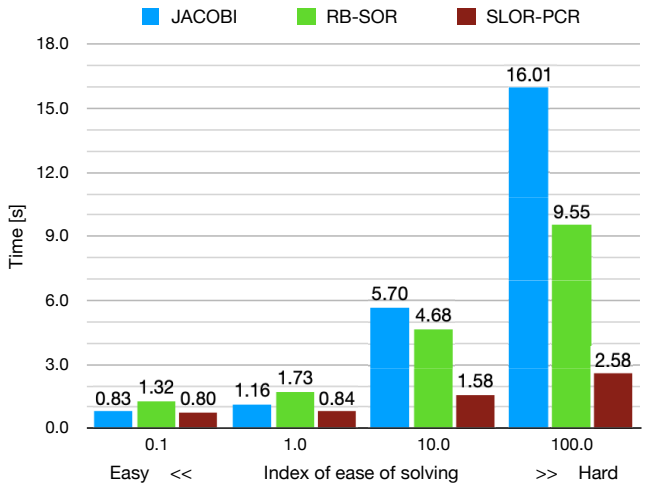

Figure 17: Execution times among solvers for different problem settings on UV 300. The problem size is 192 × 192 × 512.

6 CONCLUSION

We proposed a fully parallelizable direct-iterative hybrid method and its effective implementation to exploit the L1 cache. The proposed SLOR-PCR algorithm combines the Red-Black ordering SOR method as an iterative solver and the parallel cyclic reduction method as a direct solver. We also demonstrated an efficient implementation of the PCR algorithm for the SIMD and vector architectures, which facilitates outputting instructions optimized by the compiler. The SLOR-PCR method was examined on three modern CPUs and the NUMA, cc-NUMA, and vector architectures. The measured performance of SLOR-PCR shows excellent scalability for these three architectures, thanks to intensive utilization of the cache. The achieved performance of SLOR-PCR is higher than the conventional vectorizable Jacobi and Red-Black ordering SOR method by a factor of 3.6 to 8.3 on the SIMD architecture. Concerning the computation time, the execution time varies depending on the ease of solving a problem. Since the convergence rate of the SLOR-PCR method is superior to other methods, the more difficult the problem is and the larger the number of iterations, the shorter the calculation time for the SLOR-PCR method, as compared to the conventional methods, and the greater the advantage of using the SLOR-PCR method. The maximum speedup in computation time was observed to be a factor of 6.3 on the UV 300 cc-NUMA architecture with 352 cores.

In the future, we intend to replace the inversion of the 2 × 2 matrix with the inversion of 4 × 4 matrix, which can result in a low B/F arithmetic, and when the number of steps in the PCR procedure is reduced, the stride of the memory access becomes smaller, and thus the performance can be improved. We also considering applying the proposed SLOR-PCR to GPU, and the mixed-precision calculation to improve the performance.

ACKNOWLEDGMENTS

The present study was supported in part by MEXT as a social and scientific priority issue (Development of Innovative Design and Production Processes that Lead the Way for the Manufacturing Industry in the Near Future) to be tackled using the post-K computer (No. hp190197). computation was carried out using the computer resources offered under the category of General Project by Research Institute for Information Technology, Kyushu University.

REFERENCES

[1] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, and Jean Roman. 2011. Parallel hierarchical hybrid linear solvers for emerging computing platforms. *Comptes Rendus Mécanique* 339, 2 (2011), 96 – 103. https://doi.org/10.1016/j.crme.2010.11.005 High Performance Computing.
[2] B. L. Buzbee, G. H. Golub, and C. W. Nielson. 1970. On direct methods for solving Poissons equations. *SIAM J. Numer. Anal.* 7, 4 (1970), 627 – 656.
[3] L. Chang, J. A. Stratton, H. Kim, and W. W. Hwu. 2012. A scalable, numerically stable, high-performance tridiagonal solver using GPUs. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.12
[4] Clive. A. J. Fletcher. 1991. *Computational Techniques for Fluid Dynamics 1, Second Edition.* Springer Berlin Heidelberg.
[5] Seiji Fujino, Masatake Mori, and Toshiki Takeuchi. 1991. Performance of hyperplane ordering on vector computers. *J. Comput. Appl. Math.* 38, 1 (1991), 125 – 136. https://doi.org/10.1016/0377-0427(91)90165-G
[6] Charles Hirsch (Ed.). 1988. *Numerical Computation of Internal &Amp; External Flows: Fundamentals of Numerical Discretization.* John Wiley & Sons, Inc., New York, NY, USA. 474 pages.
[7] R. W. Hockney. 1965. A fast direct solution of Poissons equation using Fourier analysis. *J. ACM* 12, 1 (1965), 95 – 113.
[8] R. W. Hockney and C. R. Jesshope. 1981. . Adam Hilger, Bristol.
[9] Intel 2019. *Intel Xeon Processor Scalable Family, Specification Update.* Intel. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf.

[10] Silvia M. MÃijller and Dieter Scheerer. 1991. A method to parallelize tridiagonal solvers. *Parallel Comput.* 17, 2 (1991), 181 – 188. https://doi.org/10.1016/S0167-8191(05)80104-8

[11] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[12] K. Mikami, K. Ono, and J. Nonaka. 2018. Performance Evaluation and Visualization of Scientific Applications Using PMlib. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*. 243–249. https://doi.org/10.1109/CANDARW.2018.00053

[13] Kenji Ono. 2019. Testing platform for iterative solvers. Retrieved August 20, 2019 from https://github.com/kenoogl/CubeZ

[14] Kenji Ono and Kazunori Mikami. 2019. Performance monitor library. Retrieved August 20, 2019 from https://github.com/avr-aics-riken/PMlib

[15] James M. Ortega. 1988. *Introduction to Parallel and Vector Solution of Linear Systems*. Springer, US. https://doi.org/10.1007/978-1-4899-2112-3

[16] S. Rajamanickam, E. G. Boman, and M. A. Heroux. 2012. ShyLU: A Hybrid-Hybrid Solver for Multicore Platforms. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 631–643. https://doi.org/10.1109/IPDPS.2012.64

[17] Takao Sakurai, Takahiro Katagiri, Hisayasu Kuroda, Ken Naono, Mitsuyoshi Igai, and Satoshi Ohshima. 2013. A Sparse Matrix Library with Automatic Selection of Iterative Solvers and Preconditioners. *Procedia Computer Science* 18 (2013), 1332 – 1341. https://doi.org/10.1016/j.procs.2013.05.300 2013 International Conference on Computational Science.

[18] SGI 2016. *SGI UV 300 System User Guide*. SGI. Document Number 007-6351-003.

[19] H. S. Stone. 1973. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. ACM* 20, 1 (1973), 27 – 38.

[20] University of Tennessee 2019. PAPI, Performance API. Retrieved August 20, 2019 from http://icl.cs.utk.edu/papi/software/index.html

[21] Richard S. Varga. 2000. *Matrix Iterative Analysis, Second Edition*. Springer, Berlin Heidelberg.

[22] Yao Zhang, Jonathan Cohen, and John D. Owens. 2010. Fast Tridiagonal Solvers on the GPU. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1693453.1693472