

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computational performance of a smoothed particle hydrodynamics simulation for shared-memory parallel computing



Daisuke Nishiura*, Mikito Furuichi, Hide Sakaguchi

Department of Mathematical Science and Advanced Technology, Japan Agency for Marine-Earth Science and Technology, Kanagawa 236-0001, Japan

ARTICLE INFO

Article history:

Received 14 November 2013

Received in revised form

11 March 2015

Accepted 9 April 2015

Available online 18 April 2015

Keywords:

SPH

Particle simulation

OpenMP

CUDA

MIC

GPU

ABSTRACT

The computational performance of a smoothed particle hydrodynamics (SPH) simulation is investigated for three types of current shared-memory parallel computer devices: many integrated core (MIC) processors, graphics processing units (GPUs), and multi-core CPUs. We are especially interested in efficient shared-memory allocation methods for each chipset, because the efficient data access patterns differ between compute unified device architecture (CUDA) programming for GPUs and OpenMP programming for MIC processors and multi-core CPUs. We first introduce several parallel implementation techniques for the SPH code, and then examine these on our target computer architectures to determine the most effective algorithms for each processor unit. In addition, we evaluate the effective computing performance and power efficiency of the SPH simulation on each architecture, as these are critical metrics for overall performance in a multi-device environment. In our benchmark test, the GPU is found to produce the best arithmetic performance as a standalone device unit, and gives the most efficient power consumption. The multi-core CPU obtains the most effective computing performance. The computational speed of the MIC processor on Xeon Phi approached that of two Xeon CPUs. This indicates that using MICs is an attractive choice for existing SPH codes on multi-core CPUs parallelized by OpenMP, as it gains computational acceleration without the need for significant changes to the source code.

© 2015 The Authors. Published by Elsevier B.V.

This is an open access article under the CC BY-NC-ND license (<http://creativecommons.org/licenses/by-nc-nd/4.0/>).

1. Introduction

Large-scale parallel computing is important for numerically reproducing actual measurement results and dynamics of phenomena in various science and engineering areas, such as civil engineering [1], bioengineering [2], pharmaceuticals [3], and earth sciences [4,5]. The computational performance of parallelized software tools plays a critical role in such simulation studies, as these improve the computational accuracy relative to the simulation resolution within a limited computation time. Recent massively parallel computer systems based on shared- and distributed-memory architectures employ various types of arithmetic processors. Current processor designs are known to exhibit totally different computational performance depending on the numerical algorithms and implementation methods employed. Thus, it is important to investigate and compare different numerical algorithms and code tuning techniques for each type of processor.

Currently, parallel computing generally uses either a multi-core central processing unit (CPU), graphics processing unit (GPU), or many integrated core (MIC) processor. Multi-core CPUs have traditionally been used in high-performance computing, whereas GPUs were originally designed for computer graphics with many arithmetic cores [6]. Nowadays, the purpose of GPUs is more generalized, and they are used in many of the parallel computer systems on the TOP 500 list [7]. MIC is a new hardware accelerator used in processors such as Intel's Xeon Phi [8], which consists of up to 61 cores. The MIC architecture is used in the cluster systems of Tianhe-2 and Stampede, which were ranked first and sixth, respectively, in the 2013 TOP 500 list. These recent supercomputers employ distributed/shared hybrid memory systems designed for inter/inner-node hierarchically parallelized applications.

The common progress of current processor designs is the increase in the number of cores using vector operations such as single-instruction-multiple-data (SIMD). In such a situation, the shared-memory parallelization plays a basic but critical role in dealing with

* Corresponding author.

E-mail address: nishiura@jamstec.go.jp (D. Nishiura).

the increasing number of arithmetic cores in an efficient manner. However, parallel computing often cannot be performed efficiently on shared memory owing to memory-access conflicts, whereby parallel threads concurrently write to the same memory address. In addition, for multi-core processors, memory allocation must be carefully considered, because data locality significantly influences the memory access speed. For many-core processors such as GPUs, data alignment should also be appropriately implemented, as this affects the global memory access speed.

Numerical simulation methods used in science and engineering include the finite difference method (FDM), finite element method (FEM), finite volume method (FVM), boundary element method (BEM), and particle simulation method (PSM). Among these, PSM has a benefit of being mesh-free, allowing the computation of large-scale deformations and fractures of a continuum body without expensive remeshing tasks. As a PSM, smoothed particle hydrodynamics (SPH) is often used for a range of applications including wave breaking, tsunami simulations, etc. [9–12] because of its robustness in free-surface fluid dynamics. However, PSM programs must be implemented carefully to avoid write-access conflicts under shared-memory parallelization, especially when calculating a resultant force. In general, the inter-particle interaction force \mathbf{F}_{ij} between particles i and j is calculated once per interacting pair, and, according to the action–reaction law, the calculated force is distributed between the two particles as \mathbf{F}_{ij} and \mathbf{F}_{ji} ($= -\mathbf{F}_{ij}$). Conflicts may arise on the shared memory when the interaction forces are distributed to each particle i and j in parallel, because different parallel threads may concurrently add the force to the same particle.

To address these issues, a number of parallel algorithms for shared memory have been developed [13–19]. One of the simplest methods is to use atomic instructions, which atomically access memory locations to parallelize the reduction operation by adding compiler directives to the program code. In general, however, such instructions are computationally expensive because of memory barriers. The use of private memory space on each thread is proposed for reducing the cost for such memory barriers [19], although this technique is useful only for a small number of threads. Another approach is to calculate the interaction twice [13–15], such that \mathbf{F}_{ij} and \mathbf{F}_{ji} are calculated separately in order to integrate the forces on particles i and j in parallel without using the action–reaction law. Although this method can avoid write-access conflicts, it requires double the arithmetic cost to evaluate the reaction force \mathbf{F}_{ji} . We have proposed parallel algorithms to avoid this problem [20]. Our algorithms use the action–reaction law to evaluate \mathbf{F}_{ji} from \mathbf{F}_{ij} , and parallelize the interaction summation with a reference table to avoid memory access conflicts. Our method was found to show high computational performance on GPUs, but it is not clear whether our algorithms are also efficient on other current processors.

To suggest which processor and implementation is suitable for PSM, we investigate the parallel performance of an SPH program on various many- and multi-core platforms. First, we introduce several parallelization strategies for three major modules of SPH, namely neighbor particle pair list creation, interaction calculation, and updating particle information. The computation time of these modules is then examined to determine the best algorithm for each processor type: CPUs with/without SIMD instructions, MIC, and GPUs. Finally, we discuss the effective performance and power efficiency for the SPH simulation in high-performance computing.

2. Computational procedures for parallelized SPH simulations

2.1. Formulation of SPH simulation

SPH is a mesh-free simulation method that discretizes the field with explicitly tracked reference particles [21]. Each particle has a kernel function characterized by a spatial distance, called the “smoothing length”. In this research, Wendland’s function in three-dimensional space is used as the kernel function W_{ij} with smoothing length h :

$$W_{ij} = \frac{21}{16\pi h^3} \left(1 - \frac{|\mathbf{r}_{ij}|}{2h}\right)^4 \left(\frac{2|\mathbf{r}_{ij}|}{h} + 1\right) \quad 0 \leq |\mathbf{r}_{ij}| \leq 2h, \quad (1)$$

$$\nabla_i W_{ij} = \frac{105}{16\pi h^5} \left(\frac{|\mathbf{r}_{ij}|}{2h} - 1\right)^3 \mathbf{r}_{ij}, \quad (2)$$

where $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ is the relative position between particle i and particle j , with \mathbf{r}_k denoting the position of particle k , and $\nabla_i W_{ij}$ is the gradient of the kernel function.

By discretizing the Navier–Stokes equations of fluid with the kernel function, the momentum equation and the continuity equation are given by

$$\frac{d\mathbf{v}_i}{dt} = - \sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} \right) \nabla_i W_{ij} + \sum_j m_j \left(\frac{\xi}{\rho_i \rho_j} \frac{4\mu_i \mu_j}{(\mu_i + \mu_j)} \frac{\mathbf{v}_{ij} \cdot \mathbf{r}_{ij}}{r_{ij}^2 + \eta^2} \right) \nabla_i W_{ij} \quad (3)$$

and

$$\frac{d\rho_i}{dt} = \sum_j m_j \mathbf{v}_{ij} \cdot \nabla_i W_{ij} \quad (4)$$

respectively, where \mathbf{v}_{ij} ($= \mathbf{v}_i - \mathbf{v}_j$) is the relative velocity between particle i and particle j , and \mathbf{v}_k , P_k , ρ_k , μ_k , and m_k are the velocity, pressure, density, viscosity, and mass of the k th particle, respectively. η is a small parameter used for smoothing out the singularity at $\mathbf{r}_{ij} = 0$, which is set to $0.01h$. ξ is determined to be 4.96333 as per a calibration against the known exact transient solution of the Couette flow [22]. The local pressure in the first term on the right-hand side of Eq. (3) is given by the following equation of state, which is based on Tait’s equation [23]:

$$P_i = \frac{c_0^2 \rho_0}{\gamma} \left[\left(\frac{\rho_i}{\rho_0} \right)^\gamma - 1 \right] \quad (5)$$

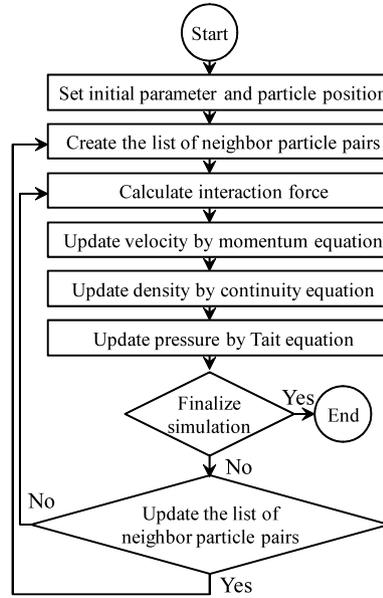


Fig. 1. Flowchart of SPH simulation.

where $\gamma = 7$, ρ_0 is the reference density, and c_0 is the speed of sound at the reference density. This means that fluid is treated as weakly compressible. The second term on the right-hand side of Eq. (3) is the viscous stress force, which is a more sophisticated viscous term obtained by improving the original one by Monaghan [23]. The right-hand side of Eq. (4) is compressibility, which is calculated by the kernel function.

An overview of the calculation process of the SPH simulation is given in Fig. 1. First, the initial state is determined for an arbitrary number of particles with uniform displacement. The list of neighbor particle pairs is created to speed up the search of the interacting particle pairs that are placed within the smoothing length. We then calculate the right-hand side of Eq. (3) using the pair list to update the particle velocities. The particle positions are also updated using the updated velocities according to Euler's method. Finally, the density is calculated from the updated velocity according to Eq. (4), and the local pressure is calculated from the updated density using Eq. (5).

2.2. Subroutines of SPH simulation

We consider three major modules in the calculation procedure of our SPH simulation. These are the creation of the neighbor particle pair list, calculation of interactions, and update of particle information. In the following subsections, we introduce subroutines that implement techniques to calculate Eqs. (3)–(5) efficiently with shared-memory parallelization.

2.2.1. Creating the list of neighbor particle pairs

The list of neighbor particle pairs is used to search for interacting particle pairs. The pair list is recreated during the simulation when the cumulative displacement of a particle exceeds $0.5\alpha h$, where h is the smoothing length and α is a constant set to 0.1 in this study. This technique is similar to that proposed by Domínguez et al. [24]; both techniques are based on the Verlet list method [25] but use a different constant control parameter αh to update the pair list.

First, every particle is identified by a particle label (i or j) and belongs to the cell labeled by $cell_label$. Grid cells are defined on a uniform Cartesian grid with a size of $2h + \alpha h$. To assign each particle to a corresponding cell in parallel, we first sort the particle labels using $cell_label$ as a sortkey. The minimum and maximum particle labels in each cell are then recorded in the arrays $ID[1, cell_label]$ and $ID[2, cell_label]$, respectively. This type of sorting is known to be used in efficient algorithms for creating neighboring particle lists in a parallel computing environment [20,19]. This algorithm is shown in Sub_C, where N_p is the total number of particles and IB is the array of cell labels corresponding to the positions of all particles.

In the next step, we create a neighbor particle pair list. From the array ID , we already know which particles belong to the same or adjacent cells. The particle pairs whose displacement is less than $2h + \alpha h$ are listed as neighbor particle pairs. The pair list is created as the one-dimensional arrays $List_i[l] = i$ and $List_j[l] = j$, which indicates that the l th pair consists of the i th and j th particles. The procedure of searching for pairs is operated by outer and inner loops. The outer loop considers the i th particle index, and the inner loop searches the index of the j th particle, which is a neighbor of the i th particle. We label the particle pairs obtained in this way with a sequential index number that is used for efficient memory access. In the simplest method, such labeling tasks are conducted in a sequential manner, even though pair-searching tasks can be parallelized in the outer loop. Because a number of the i th particle's neighbors are unknown before the inner loop calculation, it is not straightforward to parallelize the labeling of the pair list array with a sequential index in the outer loop.

In our method, we count the number of neighbor particles for $j > i$ and $j < i$. These are defined as the arrays $ipair$ and $jpair$, respectively. In addition, the prefix sum ipl of the pair list for the i th particle, defined by

$$ipl[i] = \sum_{n=1}^i ipair[n] \quad ipl[0] = 0, \quad (6)$$

Sub_C: Storing particle labels to corresponding cell data

```

1:  Call sort {particle label is sorted by cell label}
2:  for  $i = 1$  to  $N_p$  do {enable thread parallelism}
3:     $ibl \leftarrow ib[i - 1]$ 
4:     $ibi \leftarrow ib[i]$ 
5:     $ibr \leftarrow ib[i + 1]$ 
6:    if  $ibl < ibi$  then
7:       $ID[1, ibi] \leftarrow i$ 
8:    endif
9:    if  $ibr > ibi$  then
10:      $ID[2, ibi] \leftarrow i$ 
11:   endif
12: end do

```

is calculated to parallelize the pair list labeling task. This preconditioning allows the neighbor particles of i and $j > i$ to be listed in parallel to the arrays of the particle pair list, $List_i[l]$ and $List_j[l]$, because we already know that the pair list label l for particle i ranges from $ipl[i - 1] + 1$ to $ipl[i]$.

We consider seven implementation methods for particle pair creation, named “Sub_L#”, where # is the label of the subroutine. An overview of these subroutines is given in Table 1. Details of the subroutines for calculating the pair list are summarized below.

Sub_L1, Sub_L1s: In these subroutines, two pair list arrays (i.e., $List_i$ and $List_j$) are used to calculate the interactions between particles by parallelizing each index of the particle pair list, as shown in Section 2.2.2. The difference between Sub_L1 and Sub_L1s is in the prefix sum operation; Sub_L1 computes this in parallel, whereas Sub_L1s computes this sequentially.

Sub_L2: This subroutine uses only the $List_j$ pair list. The difference between Sub_L1 and Sub_L2 in calculating inter-particle interactions is explained in Section 2.2.2.

Sub_L3: This stores all neighbor j th particles for each i th particle in the pair list (i.e., $List_j$ of Sub_L3 contains particles with $j > i$ and $j < i$, whereas Sub_L2 only creates the list for $j > i$). This pair list is used for the method that calculates the interactions between each pair twice, first for the i th particle from j , and second for the j th particle from i .

Sub_L4: This subroutine creates the same pair list as Sub_L2, but its inner loop length for searching neighboring particles is only half that of the other subroutines. This is because the $jpair$ and $tempji$ arrays are redundant. This subroutine is used with the interaction calculation of the parallel atomic instructions, as described in Section 2.2.2.

Sub_L2_gpu, Sub_L3_gpu: These subroutines are customized versions of Sub_L2–3 for GPU computing. It is well-known that the memory access pattern greatly influences the performance of GPU calculations. In fact, the global memory of GPU attains maximum efficiency only when the access pattern has the same order as the threads, i.e., coalescent memory access. However, the creation of the pair list $List_j$ on line 51 of Sub_L2–3 achieves non-coalescent memory access by parallel threads, as shown in Fig. 2(b). Thus, Sub_L2–3 cannot attain efficient performance on a GPU. We therefore create the code for Sub_L2_gpu and Sub_L3_gpu so as to allow coalescent memory access on line 53 of the code, as shown in Fig. 2(a). Note that this improved memory access requires a larger memory space than Sub_L2–3, as seen from a comparison of Fig. 2(a) and (b).

Another notable feature of Sub_L1, Sub_L2, and Sub_L2_gpu is the reference table array Ref . This is created at the end of the subroutine, and references the index of the reaction force array from $jpair$. The reference table is used to sum the reaction forces, as described in the next subsection.

For further optimization of codes Sub_L#, the methods of simplifying neighbor search and dividing the domain into smaller cells proposed by Domínguez et al. [19] can be applied to reduce the cost for searching neighbor particles. In this work, however, we did not implement these methods for simplicity.

2.2.2. Calculation of interactions between particle pairs

The interaction forces of our SPH formulation (Eq. (3)) involve the pressure gradient and laminar viscous stress. We first calculate the interactions for each l th pair $F[l]$ in parallel using the pair list of Section 2.2.1. The interaction forces $F[l]$ exerted on the i th particle are then gathered as the force term $Force[i]$. The density calculation (Eq. (4)) proceeds in the same manner as that for the interaction force. In our study, we employ seven subroutines, named “Sub_L#”. These differ in their choice of pair list from Section 2.2.1 and summation algorithm for the interactions, as summarized in Table 2. Details of these subroutines are explained below.

Sub_L1: This subroutine uses the pair list from Sub_L1 to compute $F[l]$ in parallel. By utilizing the reference table $Ref[l]$, the interaction forces exerted on particle i are then combined as

$$Force[i] = \sum_{l=ipl[i-1]+1}^{ipl[i]} F[l] - \sum_{l=jpl[i-1]+1}^{jpl[i]} F[Ref[l]] \quad (7)$$

where $jpl[i]$ is the prefix sum of $jpair[i]$, and $jpair[i]$ is the number of neighbor j th particles ($j < i$) for the i th particle. The first and second terms on the right-hand side are the summation of interactions from j th particles with $j > i$ and $j < i$, respectively. To calculate the density from Eq. (4), the relative velocities are combined in a similar way to Eq. (7) as

$$Rvel[i] = \sum_{l=ipl[i-1]+1}^{ipl[i]} V[l] + \sum_{l=jpl[i-1]+1}^{jpl[i]} V[Ref[l]] \quad (8)$$

where $V = m_j \mathbf{v}_{ij} \nabla_j W_{ij}$ is defined between particle i and its neighbor particle j .

Table 1
Summary of subroutines for particle pair list creation.

Subroutine name: Sub_L#	Major aspects
Sub_L1s	<ul style="list-style-type: none"> • Pair list is constructed by two arrays of $List_i$ and $List_j$ that contain particle i and particle $j > i$ • Pair list is labeled using sequential prefix sum • Reference table array of Ref is created to sum up interactions
Sub_L1	<ul style="list-style-type: none"> • Pair list is constructed by two arrays of $List_i$ and $List_j$ that contain particle i and particle $j > i$ • Pair list is labeled using parallel prefix sum • Reference table array of Ref is created to sum up interactions
Sub_L2	<ul style="list-style-type: none"> • Pair list is constructed by one array of $List_j$ which contains particle $j > i$ • Pair list is labeled using parallel prefix sum • Reference table array of Ref is created to sum up interactions
Sub_L2_gpu	<ul style="list-style-type: none"> • Pair list is constructed by one array of $List_j$ that contains particle $j > i$ • Pair list is labeled using parallel prefix sum • Reference table array of Ref is created to sum up interactions • Pair list data is aligned to allow coalescent memory access on GPU
Sub_L3	<ul style="list-style-type: none"> • Pair list is constructed by one array of $List_j$ that contains all neighbor particles • Pair list is labeled using parallel prefix sum
Sub_L3_gpu	<ul style="list-style-type: none"> • Pair list is constructed by one array of $List_j$ that contains all neighbor particles • Pair list is labeled using parallel prefix sum • Pair list data is aligned to allow coalescent memory access on GPU
Sub_L4	<ul style="list-style-type: none"> • Pair list is constructed by one array of $List_j$ that contains particle $j > i$ • Pair list is labeled using parallel prefix sum • The number of searched cells is half that of other subroutines

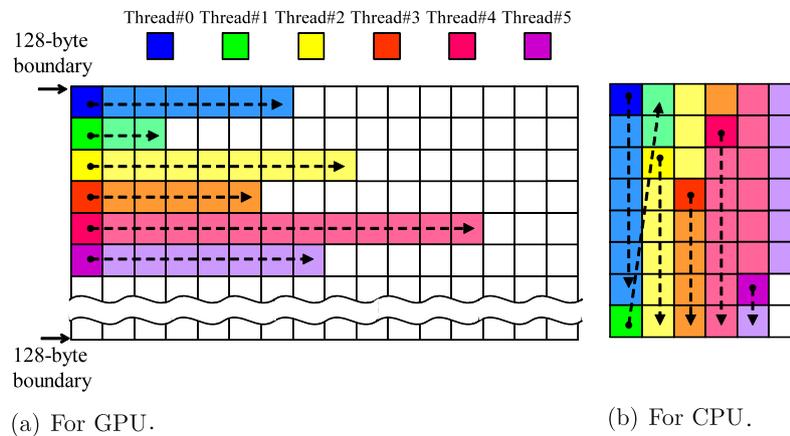


Fig. 2. Memory allocation of the arrays of particle pair list and interaction forces.

Sub_I2: This subroutine uses the pair list from Sub_L2 to calculate the interactions of $F[l]$ and $V[l]$. Different from Sub_I1, the first term on the right-hand side of Eqs. (7) and (8) can be additively calculated in parallel within the inner loop of the interactions calculation. Therefore, we need only calculate the second term on the right-hand side of Eqs. (7) and (8) to obtain the resultant interaction after the calculation loop for $F[l]$ and $V[l]$.

Sub_I3: This calculates the interactions twice using the pair list from Sub_L3, which does not require any reference table for the summation of interactions as the second term on the right-hand side of Eqs. (7) and (8) is not necessary. Note that, in the first term on the right-hand side of Eqs. (7) and (8), $ipl[i]$ is the prefix sum for the number of all neighbor j th particles with $j > i$ and $j < i$ for the i th particle.

Sub_I4: This subroutine uses the list from Sub_L4, and simply utilizes an atomic instruction “!\$OMP ATOMIC” for the summation of interactions without using the reference table.

Sub_I1_gpu, Sub_I2_gpu, and Sub_I3_gpu: These are customized subroutines for GPU computing. Sub_I1_gpu, Sub_I2_gpu, and Sub_I3_gpu allow coalescent memory access to calculate interactions using the pair lists from Sub_L1, Sub_L2_gpu, and Sub_L3_gpu, respectively. In calculating Eqs. (7) and (8), Sub_I1_gpu and Sub_I2_gpu allow read-access with the coalesced memory for arrays F and V and for the reference table array Ref , respectively. In addition, Sub_I2_gpu and Sub_I3_gpu can operate the read-access for the pair list array with the coalesced memory. One of the known techniques for efficient GPU implementation is grouping particle information (e.g., coordinates and density) by particle index on the memory. If four single-precision values are grouped, then the memory latency cost to access the global memory for particle interaction calculations can be reduced by one-fourth [19]. However, we did not implement this technique in our work; we argue that the grouping scheme will not change our performance analysis drastically because in this study we

Sub_L#: Creation of neighbor particle pair list

```

1:  #define Sub_L# {subroutine name, Sub_L1, Sub_L2_gpu, etc.}
2:  for  $i = 1$  to  $N_p$  do {enable thread parallelism}
3:     $lij \leftarrow 0$ 
4:     $lji \leftarrow 0$ 
5:     $ibi \leftarrow ib[i]$ 
6:  #if defined Sub_L1, Sub_L2, Sub_L3, Sub_L2_gpu, or Sub_L3_gpu
7:    for  $ix = 1$  to 27 do {search for 27 neighbor cells}
8:  #elif defined Sub_L4
9:    for  $ix = 1$  to 14 do {search for 14 neighbor cells}
10: #end if
11:    $icel = ic[ix, ibi]$  {neighbor cell index containing cell  $ibi$ }
12:   for  $j = ID[1, icel]$  to  $ID[2, icel]$  do
13:     if distance between particle  $i$  and particle  $j$  is less than  $2h + \alpha h$  then
14:   #if defined Sub_L1, Sub_L2, or Sub_L2_gpu
15:     if  $i < j$  then
16:        $lij \leftarrow lij + 1$ 
17:        $tempij[lij, i] \leftarrow j$ 
18:     else if  $i > j$  then
19:        $lji \leftarrow lji + 1$ 
20:        $tempji[lji, i] \leftarrow j$ 
21:     end if
22:   #elif defined Sub_L3, or Sub_L3_gpu
23:      $lij \leftarrow lij + 1$ 
24:      $tempij[lij, i] \leftarrow j$ 
25:   #elif defined Sub_L4
26:     if  $i < j$  then
27:        $lij \leftarrow lij + 1$ 
28:        $tempij[lij, i] \leftarrow j$ 
29:     end if
30:   #end if
31:   end if
32:   end do
33:   end do
34:    $ipair[i] \leftarrow lij$ 
35: #if defined Sub_L1, Sub_L2, or Sub_L2_gpu
36:    $jpair[i] \leftarrow lji$ 
37: #end if
38: end do
39: #if defined Sub_L1s
40:   Call sequential_prefix_sum {sequential prefix sum for  $ipair$  and  $jpair$ }
41: #elif
42:   Call parallel_prefix_sum {parallel prefix sum for  $ipair$  and  $jpair$ }
43: #end if
44:   for  $i = 1$  to  $N_p$  do {enable thread parallelism}
45:     for  $k = 1$  to  $ipair[i]$  do
46:        $l \leftarrow ip[l[i - 1] + k]$ 
47:   #if defined Sub_L1
48:      $List_i[l] \leftarrow i$ 
49:   #end if
50: #if defined Sub_L1, Sub_L2, Sub_L3, or Sub_L4
51:    $List_j[l] \leftarrow tempij[k, i]$ 
52: #elif defined Sub_L2_gpu, or Sub_L3_gpu
53:    $List_j[i + N16_p \times (k - 1)] \leftarrow tempij[k, i]$  {where  $N16_p = \text{INT}((N_p + 15)/16) \times 16$ }
54: #end if
55:   end do
56: end do
57: #if defined Sub_L1, Sub_L2, or Sub_L2_gpu
58:   Call reference_table {create reference table  $Ref$  using  $jpair$  and  $tempji$ }
59: #end if

```

are dealing with double-precision values. In current GPU architecture, 16 bytes is the upper limit for the word size for coalescent memory access. Therefore, we can group only two values for reducing the memory latency cost for the double-precision values.

Table 2
Summary of subroutines for interaction calculation.

Subroutine name: Sub_I#	Major aspects
Sub_I1	<ul style="list-style-type: none"> Interaction calculation is parallelized in the pair list index by utilizing $List_i$ and $List_j$ Summation of interactions are parallelized by utilizing Ref
Sub_I1_gpu	<ul style="list-style-type: none"> Interaction calculation is parallelized in the pair list index by utilizing $List_i$ and $List_j$ Summation of interactions are parallelized by utilizing Ref Temporal arrays of interactions (F and V) and Ref are read with coalescent memory access on GPU
Sub_I2	<ul style="list-style-type: none"> Interaction calculation is parallelized in the particle index by utilizing $List_j$ Summation of interactions are parallelized by utilizing Ref
Sub_I2_gpu	<ul style="list-style-type: none"> Interaction calculation is parallelized in the particle index by utilizing $List_j$ Summation of interactions are parallelized by utilizing Ref $List_j$ and Ref are read with coalescent memory access on GPU
Sub_I3	<ul style="list-style-type: none"> Interaction calculation is parallelized in the particle index by utilizing $List_j$ Summation of interactions is parallelized by double calculation without action–reaction law
Sub_I3_gpu	<ul style="list-style-type: none"> Interaction calculation is parallelized in the particle index by utilizing $List_j$ Summation of interactions is parallelized by double calculation without action–reaction law $List_j$ is read with coalescent memory access on GPU
Sub_I4	<ul style="list-style-type: none"> Interaction calculation is parallelized in the particle index by utilizing $List_j$ Summation of interactions is parallelized using atomic-add instruction

Sub_I1, Sub_I1_gpu: Calculation of interactions for Sub_I1 and Sub_I1_gpu

```

1:  #define Sub_I1 or Sub_I1_gpu
2:    Force  $\leftarrow$  0 or Rvel  $\leftarrow$  0
   --- Calculation of interactions between a particle pair ---
3:    for  $l = 1$  to  $N_l$  do {enable thread parallelism for pair list index}
4:       $i \leftarrow List_i[l]$ 
5:       $j \leftarrow List_j[l]$ 
6:       $FV \leftarrow 0$ 
7:      if distance between particle  $i$  and particle  $j$  is less than  $2h$  then
8:         $FV \leftarrow$  {forces or relative velocities}
9:      end if
10:   #if defined Sub_I1
11:      $F[l] \leftarrow FV$  or  $V[l] \leftarrow FV$ 
12:   #elif defined Sub_I1_gpu
13:      $l16 \leftarrow i + N16_p \times (l - ipl[i - 1])$ 
14:      $F[l16] \leftarrow FV$  or  $V[l16] \leftarrow FV$ 
15:   #end if
16:   end do
   --- Summation of interactions ---
17:   for  $i = 1$  to  $N_p$  do {enable thread parallelism}
18:     for  $k = 1$  to  $ipair[i]$  do
19:     #if defined Sub_I1
20:        $l \leftarrow ipl[i - 1] + k$ 
21:     #elif defined Sub_I1_gpu
22:        $l \leftarrow i + N16_p \times k$ 
23:     #end if
24:      $Force[i] \leftarrow Force[i] + F[l]$  or  $Rvel[i] \leftarrow Rvel[i] + V[l]$ 
25:     end do
26:     for  $k = 1$  to  $jpair[i]$  do
27:        $l \leftarrow jpl[i - 1] + k$ 
28:        $Force[i] \leftarrow Force[i] - F[Ref[l]]$  or  $Rvel[i] \leftarrow Rvel[i] + V[Ref[l]]$ 
29:     end do
30:   end do

```

2.2.3. Update of particle information

Using Eqs. (3) and (7), the translational motion of particle i can be written as

$$\frac{d\mathbf{v}_i}{dt} = Force[i]. \quad (9)$$

In addition, using Eqs. (4) and (8), the equation for the density of particle i is given by

$$\frac{d\rho_i}{dt} = Rvel[i]. \quad (10)$$

Sub_I#: Calculation of interactions, except for Sub_I1 and Sub_I1_gpu

```

1:  #define Sub_I# {subroutine name, Sub_I2, Sub_I2_gpu, etc.}
2:    Force  $\leftarrow$  0 or Rvel  $\leftarrow$  0
   --- Calculation of interactions between particle pair ---
3:    for  $i = 1$  to  $N_p$  do {enable thread parallelism for particle index}
4:      AFV  $\leftarrow$  0
5:      for  $l = ip[l[i - 1] + 1$  to  $ipl[i]$  do
6:        #if defined Sub_I2, Sub_I3, or Sub_I4
7:           $j \leftarrow List_j[l]$ 
8:        #elif defined Sub_I2_gpu or Sub_I3_gpu
9:           $l16 \leftarrow i + N16_p \times (l - ip[l[i - 1] + 1])$ 
10:          $j = List_{j1}[l16]$ 
11:        #end if
12:         FV  $\leftarrow$  0
13:         if distance between particle  $i$  and particle  $j$  is less than  $2h$  then
14:           FV  $\leftarrow$  {forces or relative velocities}
15:         end if
16:        #if defined Sub_I2
17:          $F[l] \leftarrow FV$  or  $V[l] \leftarrow FV$ 
18:          $AFV \leftarrow AFV + FV$ 
19:        #elif defined Sub_I2_gpu
20:          $F[l16] \leftarrow FV$  or  $V[l16] \leftarrow FV$ 
21:          $AFV \leftarrow AFV + FV$ 
22:        #elif defined Sub_I3 or Sub_I3_gpu
23:          $Force[i] \leftarrow Force[i] + FV$  or  $Rvel[i] \leftarrow Rvel[i] + FV$ 
24:        #elif defined Sub_I4
25:        !SOMP ATOMIC
26:          $Force[j] \leftarrow Force[j] - FV$  or  $Rvel[j] \leftarrow Rvel[j] + FV$ 
27:          $AFV \leftarrow AFV + FV$ 
28:        #end if
29:        end do
30:        #if defined Sub_I2 or Sub_I2_gpu
31:          $Force[i] \leftarrow AFV$  or  $Rvel[i] \leftarrow AFV$ 
32:        #elif defined Sub_I4
33:        !SOMP ATOMIC
34:          $Force[i] \leftarrow Force[i] + AFV$  or  $Rvel[i] \leftarrow Rvel[i] + AFV$ 
35:        #end if
36:        end do
   --- Summation of interactions ---
37:        #if defined Sub_I2 or Sub_I2_gpu
38:         for  $i = 1$  to  $N_p$  do {enable thread parallelism}
39:           for  $k = 1$  to  $jpair[i]$  do
40:             #if defined Sub_I2
41:                $l \leftarrow jpl[i - 1] + k$ 
42:             #elif defined Sub_I2_gpu
43:                $l \leftarrow i + N16_p \times k$ 
44:             #end if
45:              $Force[i] \leftarrow Force[i] - F[Ref[l]]$  or  $Rvel[i] \leftarrow Rvel[i] + V[Ref[l]]$ 
46:           end do
47:         end do
48:        #end if

```

These equations are integrated by a first-order Euler method. Using the updated density, the local pressure is updated according to Eq. (5).

These procedures for updating particle information are commonly parallelized by the particle index for all computers used in this research.

2.3. Summary of the SPH codes

The key issue of optimizing SPH simulation code is the calculation of interactions between particles. Our strategy is to create the interacting pair particle list using Sub_L# (Section 2.2.1) as a preconditioning process, as these perform the huge loop operations of interaction calculation in Sub_I# (Section 2.2.2) with efficient memory access.

To investigate the choice of algorithm for each computer system, we implement five SPH codes parallelized by OpenMP (Codes 1–5), and three codes parallelized by CUDA (Codes 2_gpu, 3_gpu, and 5_gpu) with the subroutines of Section 2.2. The relationship between the codes (Code #) and subroutines (Sub_#) is summarized in Table 3.

Table 3
Program codes tested on each computer systems.

Code name	Major aspects	Subroutines		Computer systems			
		Pair list creation	Interaction calculation	MIC	2CPU	FX10	GPU
Code 1	Interaction calculation is parallelized by pair list index and sequential prefix sum is used	Sub_L1s	Sub_I1	○	○	○	N/A ^b
Code 2	Using parallelized prefix sum in Code 1	Sub_L1	Sub_I1	○	○	○	○
Code 2_gpu	Using coalescent memory access in Code 2	Sub_L1	Sub_I1_gpu	N/A ^a	N/A ^a	N/A ^a	○
Code 3	Interaction calculation is parallelized by particle index in Code 2	Sub_L2	Sub_I2	○	○	○	○
Code 3_gpu	Using coalescent memory access in Code 3	Sub_L2_gpu	Sub_I2_gpu	N/A ^a	N/A ^a	N/A ^a	○
Code 4	Using atomic function for interaction summation in Code 3	Sub_L4	Sub_I4	○	○	○	N/A ^c
Code 5	Interaction is doubly calculated in Code 3	Sub_L3	Sub_I3	○	○	○	○
Code 5_gpu	Using coalescent memory access in Code 5	Sub_L3_gpu	Sub_I3_gpu	N/A ^a	N/A ^a	N/A ^a	○

^a Memory alignment in coalescent access for GPU is not effective for CPU and MIC in terms of cache hit ratio.

^b Non-parallelized prefix sum operation is not effective on GPU.

^c Performance of atomic function is significantly low on GPU.

Table 4
Salient architecture of the shared-memory parallel computers employed.

System	GPU	FX10	2CPU	MIC
OS	Windows	Linux	Linux	Linux
Compiler	Microsoft C++	Fujitsu Fortran	Intel Fortran 13	Intel Fortran 13
Parallelization	CUDA 5.0	OpenMP	OpenMP	OpenMP
Processor				
Product Line	GK110	SPARC64	Xeon	XeonPhi
Product Model	Titan	IXfx	E5-2680	SE10P
Number of Cores	2688 (14 SMX)	16	8	61
Peak performance [GFlops]	1300	236.5	172.8	1073
L1 Data cache	48 KB/SMX	32 KB/Core	32 KB/Core	32 KB/Core
Read only cache	48 KB/SMX	–	–	–
L2 cache	1536 KB	12 MB	2048 KB	31 MB
L3 cache	–	–	20 MB	–
Power	250	110	130	300
Memory				
Granularity [GB]	6	32	32	8
Band width [GB/s]	288.4	85	51.2	352

Code 1 and Code 2 calculate the interaction for each particle pair by looping over the index of the particle pair list. The interactions of each pair are integrated for each particle using the reference table. The difference between Codes 1 and 2 is that the prefix sum is calculated sequentially in Code 1 and in parallel in Code 2. Instead of the pair list index, Codes 3–5 use the particle index as the loop counter for the interaction calculation. Code 3 uses the reference table in the integration process of interactions, but Code 4 uses the atomic instruction. Code 5 calculates the interaction for each particle pair twice to integrate the interaction. Codes 2_gpu, 3_gpu, and 5_gpu are GPU versions of Codes 2, 3, and 5, respectively, to allow coalescent memory access.

3. Shared-memory parallel computers

The performance of the double-precision parallel SPH simulations is measured on a multi-core CPU, MIC coprocessor, and a GPU. Table 4 shows a summary of the computer systems used for the performance test. In the following subsections, we describe some details of each computer system.

3.1. Multi-core CPUs

Recently, the computational performance of CPUs has increased with the number of arithmetic cores on the chip set. However, the performance of the core itself has not changed much for several years. Therefore, an efficient highly parallel programming technique is critical to obtain good computational performance. We parallelize the codes using OpenMP on a multi-core CPU system. We use the Intel Xeon E5-2680 and Fujitsu SPARC64 IXfx multi-core CPUs. These CPU cores have an out-of-order architecture that allows later instructions to be processed before an earlier instruction is completed. In the Xeon CPU system, there are two processors with 8 cores per CPU on a single node. In total, 16 OpenMP threads are available per node. Each core in the CPU shares the 20 MB L3 cache through a bidirectional ring bus. Data access between CPUs in the nodes is accelerated by the QuickPath Interconnect (QPI) technology, which improves the data locality on local memory. In addition, each core has 32 KB L1 cache and 256 KB L2 cache. Thus, the Xeon processor has a three-level cache hierarchy. Each CPU accesses the 16 GB local memory through four DDR3 channels with a bandwidth of 51.2 GB/s. The Fujitsu FX10 node consists of one SPARC64 IXfx processor with 16 cores. These parallelize the computational operations with 16 OpenMP threads. The processor is a successor of the SPARC64 VIIIfx for the K computer in RIKEN [26]. Each core of the processor constructs a two-level cache hierarchy: 32 KB L1 cache on each core and 20 MB L2 shared cache among all cores on the chip. The processor accesses the 32 GB local memory with a bandwidth of 85 GB/s. Another notable feature is the SIMD capability of SPARC64, which conducts multiple (vector) data operations simultaneously with a single instruction. Because the SIMD vectorization technique is known to improve the computational

performance considerably, we use this acceleration by adding directives to all program codes used on the FX10 system. Note that these SIMD instructions are disabled for integer arithmetic, whose operations are frequently performed in the pair list creation procedure.

3.2. Many integrated core (MIC) coprocessors

MIC is now available as add-in PCI-Express cards. Intel's Xeon Phi MIC coprocessors have up to 61 arithmetic in-order cores. Each core supports up to four hardware threads, which can be useful for hiding the latency of data access; while one of these threads is waiting on data, others can continue executing. This latency is inherent in in-order core architectures, where multi-thread instructions must wait for previous ones to receive all operands. Each core has 32 KB L1 cache and 512 KB L2 cache. The L2 cache on each core is linked through the bidirectional ring bus network. Thus, the 31 MB L2 cache is totally shared between cores in MIC. In comparison to a CPU, the computational performance of MIC is somewhat sensitive to cache access, because the cost of cache coherency increases with the number of cores. MIC also has 8 GB of local memory that can be accessed at 352 GB/s. Another notable feature is the SIMD capability of the 512-bit vector processing units, which performs best under sequential data access.

There are two modes of running programs on MIC, native mode and offload mode. In native mode, the program runs only on the coprocessor, using the MIC as a standalone computational unit. Recompiling the existing program code is only necessary in native mode. Offload mode uses the MIC as an external accelerator, allowing jobs to be offloaded from the host CPU to the coprocessor on demand. To use this mode, some directive operations must be added to the existing CPU program code. In our performance test, we use the same test codes on MIC as for the multi-core CPUs, thus utilizing the native mode.

3.3. Graphics processing units (GPUs)

The Nvidia Geforce GTX Titan GPU consists of 14 streaming multiprocessor (SMX) units, with 192 single-precision (SP) cores and 64 double-precision (DP) units per SMX unit. This gives a total of 2688 SP cores and 896 DP units. The GPU has 6 GB local memory that can be accessed at 288.4 GB/s. There is also a 48 KB read-only cache on each SMX, in addition to 48 KB of L1 cache. To obtain efficient memory access for coalescent memory, our GPU program code is optimized to use aligned arrays with a 128-byte boundary. In the performance test, we mainly use Codes 2_gpu, 3_gpu, and 5_gpu with the aligned arrays instead of Codes 2, 3, and 5, as shown in Table 3. These codes are programmed using CUDA 5.0.

4. Performance evaluation

In our benchmark SPH test, the computational system size was $13.0 \text{ m} \times 0.5 \text{ m} \times 0.5 \text{ m}$ (depth \times width \times height), and the number of particles was 1,687,016. Of these particles, 518,816 were used as walls (i.e., a fixed boundary). Such wall particles do not require velocity and position calculations, but are included in the density and interaction calculations. The smoothing length, density, viscosity, and speed of sound were set to 0.01 m, 1000 kg/m³, 0.001 Pa s, and 1500 m/s, respectively. Initially, the boundary particles were positioned as the sidewalls of the system, and fluid particles were placed in a rectangle of height and width 0.3 m and depth 12.98 m. These particles were placed using a simple cubic lattice, where the lattice size is equal to the smoothing length. For simplicity, our performance test dealt with static solutions, in which particles do not move during the computation, although information such as the particle velocity, position, and density are updated. The neighbor particle pair list was created for every time step, and the elapsed time was measured. We estimated the performance for practical use by updating the pair list creation at certain intervals, as described in Section 4.2. The computation time per step was averaged over 100 iterations, with a discrete time step of 1×10^{-6} s.

4.1. Overview of computational cost with shared-memory parallelization

To discuss the performance of each program code on each parallel processor, we show the averaged computation times per time step for the creation of the particle pair list (T_L) by Sub_L#, the calculation of interactions between particles (T_I) by Sub_I#, and the update of particle information (T_U) in Fig. 3. The performance test was run using all cores on each processor. On multi-core CPU systems, the number of parallel threads is equal to the number of cores. To ameliorate the performance degradation caused by data access latency, the tests on the MIC and GPU systems were performed using 4 threads per core and 256 threads per streaming multiprocessor (SM), respectively.

4.1.1. Multi-core CPU

Fig. 3(a) and (b) shows the performance on two Xeon processors (2CPU) and the SPARC64 processor (FX10). We first compare the time for Code 1 and Code 2, which use subroutines Sub_L1s and Sub_L1, respectively. Sub_L1 is parallelized for the prefix sum, whereas Sub_L1s is not. We found that T_L for Code 1 is smaller than that for Code 2. This implies that parallelizing the prefix sum does not improve the computational performance on multi-core CPUs, because the 16 cores of the multi-core CPU are not sufficient to overcome the parallelization overhead. The parallel prefix sum performs $O(N_p \log_2 N_p)$ additive operations, whereas a sequential prefix sum performs only $O(N_p)$ [27]. Thus, the overhead of parallelization is $\log_2 N_p$. In this case, $\log_2 N_p$ is 20.7 for $N_p = 1,687,016$. This indicates that at least 21 cores would be required to speed up the prefix sum operation using parallelization. In terms of calculating interactions, both Code 1 and Code 2 are parallelized in the single do-loop, and show the same T_I .

For Code 3, T_L is smaller than that of Code 2, because Code 3 creates only $List_j$ instead of both $List_i$ and $List_j$. However, the T_I value of Code 3 is greater than that of Code 2. The interactions in Code 3 are calculated by a double do-loop, with the particle index as the outer loop and the pair list index as the inner loop. Although we can parallelize the outer loop, the parallel acceleration is degraded by the imbalance of the inner loop length (i.e., the number of interacting particles).

Code 4 uses atomic instructions to calculate the summation of interactions. This produced a larger T_I than that of Codes 1–3. Atomic instructions require synchronization of the thread barrier, which degrades the parallel performance. In particular, the performance of the interaction calculation is very slow on the FX10, because SIMD is unavailable with the thread barrier of atomic instructions. SIMD plays a

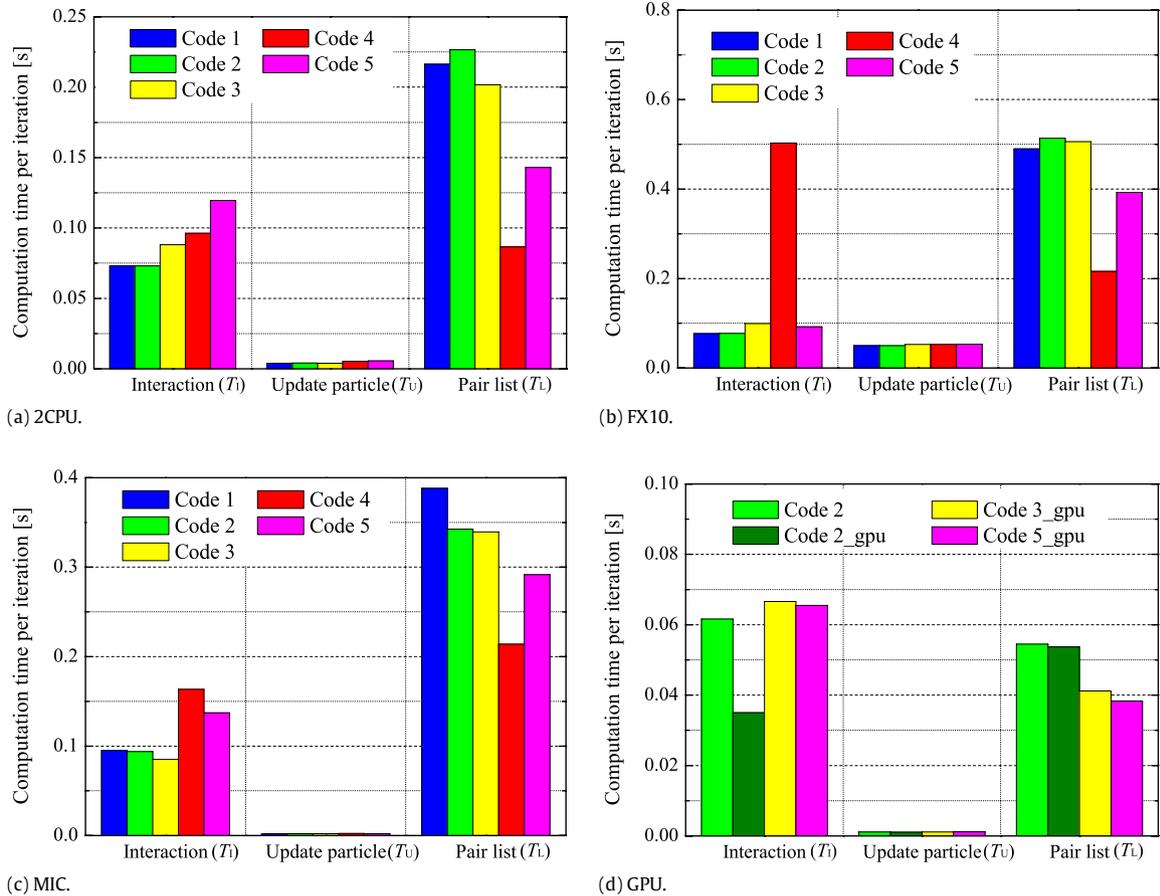


Fig. 3. Computation time per time-marching iteration of each program code for the operations of interaction calculation, time integration, and particle pair list creation on each shared-memory parallel computer.

critical role in the efficient performance of FX10. On the other hand, there are benefits to atomic instructions, as only half the number of cells must be searched for neighboring particles and the creation of a reference table is unnecessary. Therefore, Code 4 results in the best T_l value among all of the codes.

In Code 5, the arithmetic cost of calculating interactions is twice that of the other codes, because interactions are doubly calculated. Therefore, the T_i value of Code 5 is the worst among all codes on 2CPU. However, Code 5 shows good T_i on FX10, because SIMD can be used for the calculation of interactions. The SIMD acceleration on FX10 makes the computation time comparable to that using the reference table, even though the arithmetic cost of the interaction calculation is twice that of forming the reference table. However, as Code 5 does not create a reference table, its T_l value is smaller than that of Codes 1–3 on both 2CPU and FX10.

We should note that the T_l of FX10 is more than twice that of 2CPU for all codes. This is because FX10 cannot efficiently perform integer arithmetic operations in the pair list creation tasks, as these cannot be operated under SIMD.

4.1.2. MIC coprocessor

Fig. 3(c) shows the MIC performance. When we compare the T_l values of Code 1 and Code 2, the improvement obtained by parallelizing the prefix sum operation can be clearly observed. This result supports the assertion that 60 cores (240 threads) are sufficient to speed up the prefix sum in comparison to multi-core CPUs.

The T_i of Code 3 is slightly better than that of Code 2. Unlike the multi-core CPUs, the effect of the cost imbalance is not seen in the MIC coprocessor. In Code 3, the interactions are calculated for the i th particle in the outer loop, and then the data of the i th particle are temporally stored to register memory and reused during the inner loop calculation. This data access pattern can reduce the read cost from the global and cache memory. On the MIC, this acceleration can hide the overhead of the cost imbalance loop.

The T_l result for Code 4 is significantly worse than that of Code 3. The cost of atomic instructions is large in comparison to the 2CPU system. To understand this result, we investigated the effect of atomic instructions on parallel performance. Fig. 4(a) plots the speed-up of T_l in Code 3 and Code 4 against the number of active threads on MIC. The performance of Code 4, as well as that of Code 3, improves linearly until around 60 threads, because MIC has 61 cores. Increasing from 60 to 240 threads, the performance speeds up asymptotically, because the latency of data access is hidden by the hardware threads on in-order cores. This means that these codes obtain the full benefit of the number of MIC cores. However, the effect of atomic operations on the performance is not clear, because this result contains the effect of both atomic and other operations. Hence, to estimate only the overhead caused by thread barriers from atomic instructions, we calculated the parallel fraction S for the differential in computation time between Code 3 and Code 4. According to Amdahl's law, the definition of S is

$$S = \frac{T_n - T_m}{(1 - 1/m)T_n - (1 - 1/n)T_m} \quad (11)$$

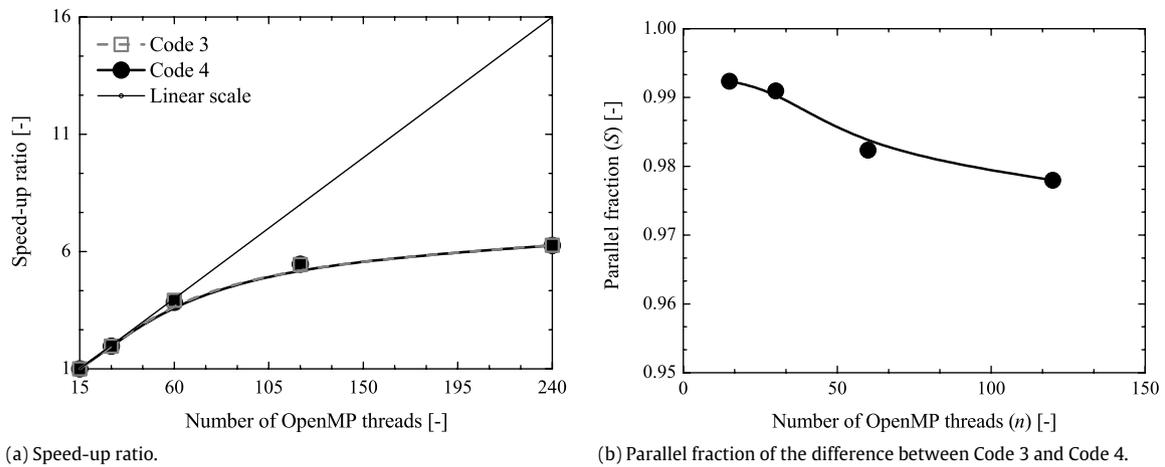


Fig. 4. Parallel performance for interaction calculation of Code 3 and Code 4 on MIC.

where T_n and T_m are the different computation times of Code 3 and Code 4 using n threads and m threads, respectively. S describes the fraction of code that can be parallelized. In other words, the computational cost of the remaining fraction $(1 - S)$ cannot be reduced by parallelization. The parallel fraction for each value of n for $m = 240$ is shown in Fig. 4(b). The parallel fraction is found to decrease as n increases. From this result, it was confirmed that the increase in overhead due to atomic instructions significantly degrades the performance of Code 4.

The performance of Code 5 compared to Code 3 is the same as that for multi-core CPUs, described in Section 4.1.1.

4.1.3. GPUs

Fig. 3(d) illustrates the performance of each code on the GPU. We first note that the computational performance of the interaction calculation in Code 2_gpu is significantly improved from that in Code 2 because of the coalescent address treatment. As a result, Code 2_gpu shows the best performance in terms of calculating interactions.

The T_1 result of Code 3_gpu is worse than that of Code 2_gpu. This performance degradation is caused by the imbalance of the inner loop length, as for the multi-core CPUs. In addition, the decrease in the parallelized loop length of Code 3_gpu, which is shorter than that of Code 2_gpu, could cause the parallel efficiency to be degraded in many-core computing. On the other hand, the interaction calculation in Code 5_gpu is slightly faster than that in Code 3_gpu, for the same reason as that for FX10. Thus, we found that the double calculation of interactions is effective in SIMD architectures such as GPU and FX10. Note that the order relation of T_1 between Code 3_gpu and Code 5_gpu is equal to that between Code 3 and Code 5 in the other processors.

4.1.4. Performance evaluation for updating particle information

In terms of updating the particle information, the computational cost is the worst on the FX10. The cache hierarchy of FX10 is two-level, rather than the three-level system of 2CPU, and the total cache per core is the lowest among the multi-CPU and MIC. On the other hand, the memory bandwidth per flop (B/F) of FX10 is larger than those of 2CPU and MIC. The large B/F value of FX10 can efficiently supply data to the processor from local memory when the data overflows from the small cache. However, the access speed of local memory is less than that of cache memory, even if the B/F of local memory is large. To update particle information, data such as position, velocity, and force are required. Thus, we believe the particle data used in the arithmetic operations overflows from the small cache on FX10. In comparison with the 2CPU and MIC systems, the amount of overflow data from the cache is larger because the cache size is smaller. Hence, the computational performance of FX10 is lower than that of the other systems, regardless of the large B/F value. On the contrary, the computational performance of the GPU is the best of all the processors considered here, because the hiding of memory access latency is perfectly implemented by the reading and writing from/to coalescent memory addresses aligned on the 128-byte boundary.

4.2. Best SPH code for shared-memory parallel computing

In Section 4.1, no single code gave the best performance for both the pair list creation (T_L) and interaction calculation (T_I). The cost balance of our SPH simulation depends on a target problem characterized by the update frequency of the particle pair list. To determine the best SPH code, we examine the effect of the pair list update frequency on the overall computation time. We then plot the relationship between the update frequency f and the computation time per iteration T_{all} for each code, as shown in Fig. 5. T_{all} is defined as

$$T_{\text{all}} = T_I + T_U + T_L f \quad (12)$$

where f is defined by the number of updates per iteration, and T_I , T_U , and T_L are obtained from the results shown in Fig. 3. For 2CPU, the computation times of Code 1 and Code 4 are smallest in the range $0 < f < 0.189$ and $0.189 < f < 1$, respectively. For FX10, the performance of Code 1 is also the best when $0 < f < 0.186$. However, different from the 2CPU result, Code 5 has the best performance when $0.186 < f < 1$. For MIC, the computation time of Code 4 is the smallest in the range $0.626 < f < 1$, and that for Code 3 is the smallest outside this range. For the GPU, Code 2_gpu shows the best performance for all values of f .

To compare performance between the processors, the performance for the best code on each processor is summarized in Fig. 6. This figure illustrates the smaller computation time given by the two best codes at $f = 1$ and $f = 0.01$. When the update frequency f is large, the computation time is found to have the order “Code 2_gpu on GPU” < “Code 4 on 2CPU” < “Code 4 on MIC” < “Code 5 on FX10”. When

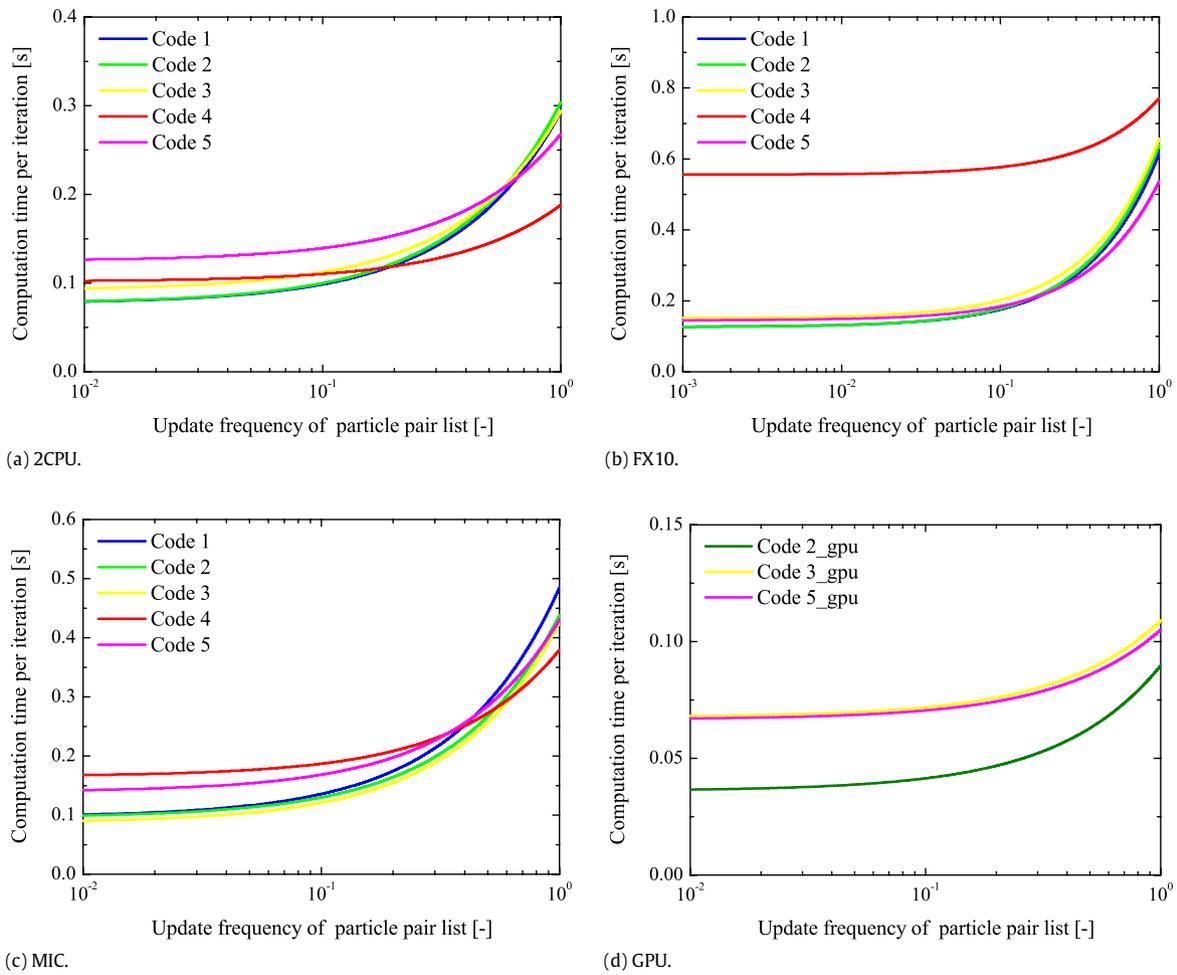


Fig. 5. Comparison of the effects of update frequency of particle pair list creation on computation time per time-marching iteration between program codes on each shared-memory parallel computer.

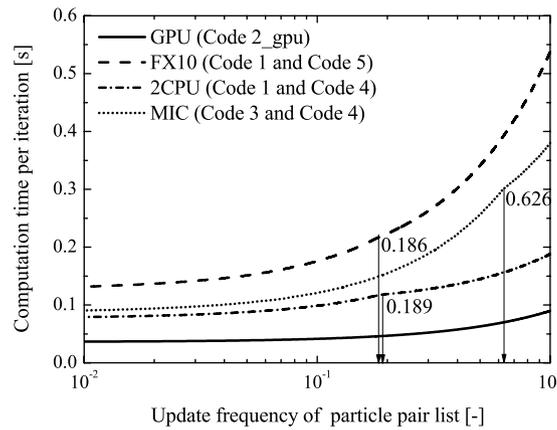


Fig. 6. Relationship between the update frequency of particle pair list creation and computation time per time-marching iteration when using the best performance code on each shared-memory parallel computer.

f is small, however, the order is “Code 2_gpu on GPU” < “Code 1 on 2CPU” < “Code 3 on MIC” < “Code 1 on FX10”. It should be noted that MIC and 2CPU exhibit similar computation times. From these results, if we are targeting computational speed on a single-node system, we recommend using Code 2_gpu on the GPU. However, if we consider overall performance, such as effective performance and power efficiency, this recommendation does not stand, as discussed in the next subsection.

4.3. Best processor for SPH simulations

To determine the most suitable processor for SPH simulations, we measured the effective performance and power efficiency. Fig. 7(a) shows the computational speed divided by the processor’s theoretical peak performance for the code used in Fig. 6, which we define as

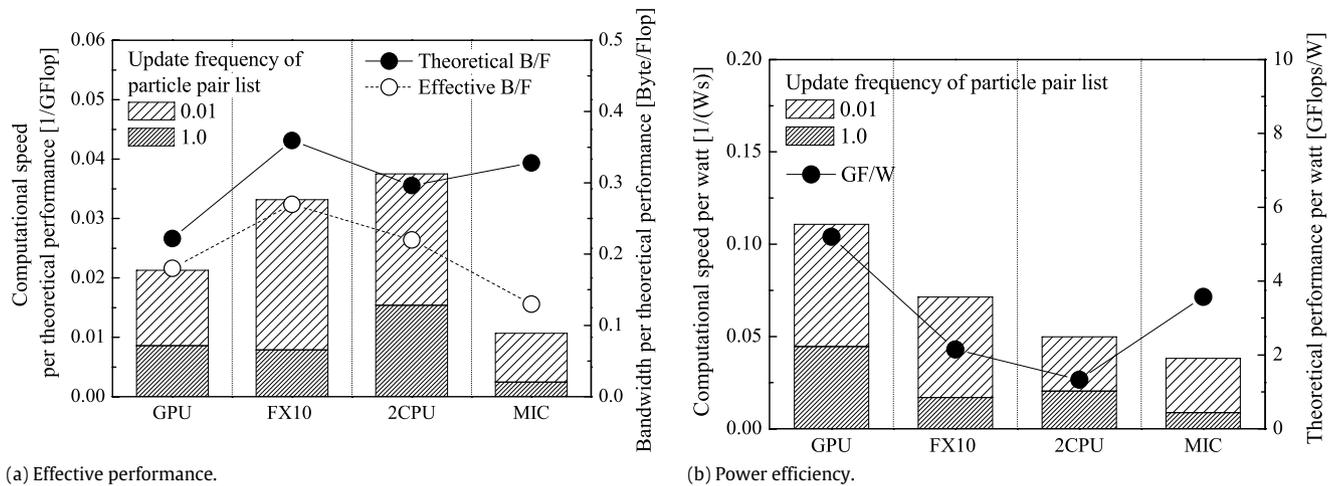


Fig. 7. Comparisons of effective performance and power efficiency between the shared-memory parallel computers when using the best performance code for each of the computers.

the “effective performance”. The theoretical memory bandwidth for the theoretical peak performance (B/F) is also shown as a solid line in the figure. Generally, high B/F results in high effective performance, because the fast data supply reduces the idle time of arithmetic units. The effective performance correlates well with B/F, except for the MIC. The effective performance of MIC is the worst, even though its B/F is as high as that of multi-core processors. We believe there are two reasons for this performance loss on MIC:

One reason concerns the performance of the data supply. In our SPH implementation, random access to read particle data is inevitable, resulting in frequent local memory access due to cache miss. We believe that such local memory access may largely degrade the computational performance because the memory bandwidth of MIC in practical use (i.e., the effective bandwidth) is not as good as shown by the theoretical performance. In order to estimate the effective memory bandwidth, we implemented the STREAM benchmark [28]. The triad test shows that the effective bandwidth of MIC is 40% of the theoretical one. This is considerably smaller than the effective bandwidth of other processors, which are more than 70% of the theoretical one. Fig. 7(a) shows a plot of the effective B/F based on the obtained effective memory bandwidth as a dashed line. The figure shows a high correlation between the effective B/F and effective performance.

The other reason concerns the low effective flop performance on MIC because SIMD instructions are disabled for our loop calculation with an indirect access. To overcome this problem, we require more advanced optimization techniques.

Fig. 7(b) shows the computational speed divided by the processor’s peak watt for the code used in Fig. 6, which we define as the “power efficiency”. The theoretical peak performance per watt (GF/W) of each processor is also shown as a reference. Similar to the effective performance in Fig. 7(a), the power efficiency is proportional to GF/W for GPU, FX10, and 2CPU. The power efficiency of MIC is the lowest, although the GF/W of MIC is the second highest after GPU. This means that MIC cannot efficiently utilize its power supply, because the effective performance of MIC is lower than that of the other processors.

As a result, if we compare parallel computer systems with the same theoretical peak performance, the fastest computational speed for our SPH simulation is obtained on the multi-core CPU system. If we compare between parallel computer systems with the same peak power consumption, the fastest computational speed is obtained on the GPU system. Thus, the multi-node system of multi-core CPUs and of GPUs are suitable for high-performance computations of SPH if other multi-node systems have the same peak performance or same peak power consumption, respectively.

5. Conclusion

We evaluated the performance of SPH simulation codes on several current shared-memory parallel computer architectures. On multi-core CPUs and MIC, the performance of pair list creation was the best when atomic instructions were used for the summation of interaction forces. For the GPU, the performance of pair list creation was the best when interactions were doubly calculated without using the action–reaction law. On the other hand, the interaction calculation with atomic instructions was not performed efficiently on any of the processors. Instead, the interaction calculation using a reference table was most efficient on all processors. The overall elapsed time for the SPH simulation was the smallest on the GPU. The second best time was given by the system of two Xeon CPUs and of MIC. Moreover, the effective performance and power efficiency of the GPU and multi-core CPUs were found to be more effective than those of MIC.

In summary, the GPU appears to be the best architecture in terms of overall highest computational speed, as long as the writing or rewriting of well-developed parallelized code with CUDA is not an issue. On the other hand, MIC has the attractive feature that the program code can be easily parallelized using a general programming language, attaining a computational speed approaching that of two CPUs. This consistency is also useful for conducting simulations using both co- and host-processors for heterogeneous parallel computing.

We hope that this performance evaluation for SPH simulation algorithms and processor architectures can help to develop more efficient parallel computation.

Acknowledgments

This work is supported by Japan Science and Technology Agency (JST) under the Core Research of Evolutional Science and Technology (CREST) project “ppOpen-HPC: Open Source Infrastructure for Development and Execution of Large-Scale Scientific Applications on Post-Peta-Scale Supercomputers with Automatic Tuning”. We thank Georg Stadler for his assistance in using the Stampede supercomputer at

Texas University for performing calculations using MIC. We also thank the reviewers for their valuable comments, which greatly helped improve the manuscript.

References

- [1] C.S. Campbell, P.W. Cleary, M. Hopkins, Large-scale landslide simulations: Global deformation, velocities and basal friction, *J. Geophys. Res.: Solid Earth* 100 (B5) (1995) 8267–8283.
- [2] A. Hosoi, T. Washio, J.-I. Okada, Y. Kadooka, K. Nakajima, T. Hisada, A multi-scale heart simulation on massively parallel computers, in: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society, New Orleans, LA, USA, 2010, pp. 1–11.
- [3] Y. Mochizuki, K. Yamashita, K. Fukuzawa, K. Takematsu, H. Watanabe, N. Taguchi, Y. Okiyama, M. Tsuboi, T. Nakano, S. Tanaka, Large-scale FMO-MP3 calculations on the surface proteins of influenza virus, hemagglutinin (HA) and neuraminidase (NA), *Chem. Phys. Lett.* 493 (4) (2010) 346–352.
- [4] W. Ohfuchi, H. Sasaki, Y. Masumoto, H. Nakamura, Virtual atmospheric and oceanic circulation in the earth simulator, *Bull. Am. Meteorol. Soc.* 88 (6) (2007) 861–866.
- [5] M. Furuichi, D. Nishiura, Robust coupled fluid-particle simulation scheme in stokes-flow regime: Toward the geodynamic simulation including granular media, *Geochemistry, Geophysics, Geosystems* 15 (7) (2014) 2865–2882.
- [6] Graphics processing unit, (2013). http://en.wikipedia.org/wiki/Graphics_processing_unit.
- [7] Top 500 supercomputer sites, (2013). <http://www.top500.org/lists/2013/06/>.
- [8] Intel xeon phi coprocessors, (2013). <http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html>.
- [9] B.D. Rogers, R.A. Dalrymple, SPH modeling of tsunami waves, in: P.L.-F. Liu, H. Yeh, C. Synolakis (Eds.), in: *Advances in Coastal and Ocean Engineering*, vol. 10, World Scientific, New York, 2008, pp. 75–100.
- [10] J. Xie, I. Nistor, T. Murty, A corrected 3-d SPH method for breaking tsunami wave modelling, *Nat. Hazards* 60 (1) (2012) 81–100.
- [11] P. St-Germain, I. Nistor, R. Townsend, T. Shibayama, Smoothed-particle hydrodynamics numerical modeling of structures impacted by tsunami bores, *J. Waterway, Port, Coastal, Ocean Eng.* 140 (1) (2013) 66–81.
- [12] M.H. Dao, H. Xu, E.S. Chan, P. Tklich, Modelling of tsunami-like wave run-up, breaking and impact on a vertical wall by SPH method, *Nat. Hazards Earth Syst. Sci.* 13 (12) (2013) 3457–3467.
- [13] J. Yang, Y. Wang, Y. Chen, GPU accelerated molecular dynamics simulation of thermal conductivities, *J. Comput. Phys.* 221 (2) (2007) 799–804.
- [14] J. Anderson, C. Lorenz, A. Travesset, General purpose molecular dynamics simulations fully implemented on graphics processing units, *J. Comput. Phys.* 227 (10) (2008) 5342–5359.
- [15] W. Liu, B. Schmidt, G. Voss, W. Müller-Wittig, Accelerating molecular dynamics simulations using graphics processing units with CUDA, *Comput. Phys. Comm.* 179 (9) (2008) 634–641.
- [16] D. Rapaport, Multi-million particle molecular dynamics, I. design considerations for vector processing, *Comput. Phys. Comm.* 62 (2) (1991) 198–216.
- [17] A. Héroult, G. Bilotta, R.A. Dalrymple, SPH on GPU with CUDA, *J. Hydraul. Res.* 48 (1) (2010) 74–79.
- [18] B. Carrión Schäfer, S.F. Quigley, A.H. Chan, Acceleration of the discrete element method (DEM) on a reconfigurable co-processor, *Comput. Struct.* 82 (20) (2004) 1707–1718.
- [19] J.M. Domínguez, A.J. Crespo, M. Gómez-Gesteira, Optimization strategies for CPU and GPU implementations of a smoothed particle hydrodynamics method, *Comput. Phys. Comm.* 184 (3) (2013) 617–627.
- [20] D. Nishiura, H. Sakaguchi, Parallel-vector algorithms for particle simulations on shared-memory multiprocessors, *J. Comput. Phys.* 230 (5) (2011) 1923–1938.
- [21] J.J. Monaghan, Smoothed particle hydrodynamics, *Annu. Rev. Astron. Astrophys.* 30 (1992) 543–574.
- [22] P.W. Cleary, New implementation of viscosity: Tests with Couette flows, SPH Technical Note #8, CSIRO Division of Mathematics and Statistics, Tech. Report DMS – C 96/32 (1996).
- [23] J.J. Monaghan, Simulating free surface flows with SPH, *J. Comput. Phys.* 110 (2) (1994) 399–406.
- [24] J.M. Domínguez, A.J.C. Crespo, M. Gómez-Gesteira, J.C. Marongiu, Neighbour lists in smoothed particle hydrodynamics, *Internat. J. Numer. Methods Fluids* 67 (12) (2011) 2026–2042.
- [25] L. Verlet, Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules, *Phys. Rev.* 159 (1967) 98–103.
- [26] K computer, (2013). <http://www.aics.riken.jp/en/kcomputer>.
- [27] W.D. Hillis, G.L. Steele Jr, Data parallel algorithms, *Commun. ACM* 29 (12) (1986) 1170–1183.
- [28] The stream benchmark: Computer memory bandwidth, (2013). <http://www.streambench.org/>.