Vol.2016-HPC-153 No.15
2016/3/2

Prototype Implementation and Its Fundamental Performance Evaluation of a Manycore-Aware OhHelp'ed PIC Simulation Code

Hiroshi Nakashima[1,a)] Keisuke Kikura[2,b)] Yohei Miyake[2,c)]

Abstract: We are now developing a manycore-aware implementation of PIC simulation code based on EMSES, a highly parallel production-level space-plasma simulator with the load-balancing library OhHelp. As a step of the development, we implemented its prototype to confirm the effectiveness and efficiency of our manycore-aware techniques such as cell coloring, on-the-fly particle sorting, and position-aware inter-process particle transfer. In this report we summarize these issues and then present results of fundamental performance evaluation with various artificial settings including those causing severe load imbalance among processes.

1. Introduction

We are now developing a manycore-aware implementation of PIC (Particle-In-Cell) simulation code based on EMSES[1], a highly parallel production-level space-plasma simulator with the load-balancing library OhHelp[4], targeting processors of many tens (or hundreds) of ordinary cores with a wide SIMD mechanism such as Intel Xeon Phi. As the very first step of our development, we have implemented a proof-of-concept single-node version[3] with cell-coloring, strict particle-binning and field-array scalarization for full exploitation of many cores and SIMD mechanism as summarized in Section 2, to have good single-processor performance of the KNC (Knights Corner) version of Xeon Phi which significantly surpasses that of ordinary Xeon Haswell.

However, this single-node implementation is obviously far from our goal because a production level PIC simulator must be capable of many billions of particles which a single node cannot accommodate. Therefore, we have proceeded to the next step for a prototype multi-node implementation with OhHelp as discussed in this report. We consider the implementation as a prototype because, though its kernels for time-evolutional particle and electromagnetic field simulation are very realistic, we omit various features required for a production level simulator, such as initial particle/field settings, (occasional) Poisson equation solving, non-periodic boundary conditions, snapshot output, and so on. However, since the key performance issues of manycore-aware PIC simulators lie in the particle management mechanism, we believe our prototyping is the essential step toward our ultimate goal.

[1] Kyoto University, Yoshida Hon-machi, Sakyo, Kyoto, 606-8501, Japan
[2] Kobe University, 1-1 Rokkodai-cho, Nada, Kobe, 657-8501, Japan
[a)] h.nakashima@media.kyoto-u.ac.jp
[b)] kikkun28@stu.kobe-u.ac.jp
[c)] y-miyake@eagle.kobe-u.ac.jp

In fact, as discussed in Section 3, we found a few critical implementation issues on the management of both intra- and inter-process particle motions. That is, we found that an optimization to reduce the number of inter-process particle transfers causes an artificial depletion of particle bins to make our strict binning unexpectedly costly. Another finding is that OhHelp's load-balancing mechanism sometimes allocate a small but congested particle mass to a process in which a severe intra-process imbalance occurs to add a significantly large cost to the binning. Our work is of course not only to point out difficulties but also to find reasonably efficient solutions for them as shown in the section.

The efficiency of our solutions and of the whole of our prototype implementation are evaluated in Section 4 with various artificial settings of particle distribution and motion. These evaluations also revealed a performance bottleneck due to poor single-core performance of the KNC version of Xeon Phi by which inter-node MPI communication performance is limited and the optimal per-processor number of processes is shifted higher than expected and desired.

The evaluation also prompted us to improve our implementation as summarized in the concluding Section 5, in which we show our to-do list toward the goal as well.

2. Basic Single-Node Implementation

In this section, we summarize manycore-aware issues of the single-node PIC implementation we presented in [3]. The PIC method is to simulate the motion of plasma particles being ions and electrons by modeling them as a huge set of *super-particles* moving in a large scale discretized electromagnetic field. In a PIC simulation, the time evolution of the system is tracked by repeating the following three operations for each discrete time step.

Particle-push is to accelerate each particle by electric and Lorentz force laws referring to the electric and magnetic

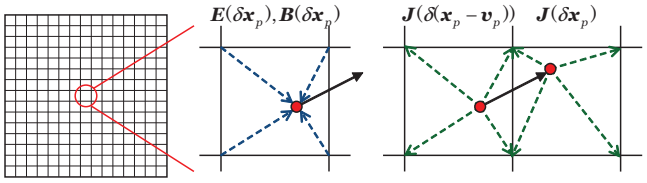

Fig. 1 Particle-push and current-scatter.

fields E and B surrounding the particle. More specifically, with the first-order shape function, the acceleration acting on a particle p is determined by its position x_p, velocity v_p, mass m_p, charge q_p and the electromagnetic field vectors E and B defined at the set of grid points δx_p being the eight vertices of the cubic cell in which the particle resides, as shown in Figure 1. Then the particle moves to a new position x_p according to its updated velocity v_p.

Current-scatter is to calculate current density J summing up the small currents caused by the motion of charged particles. From the viewpoint of a particle p, this calculation is to add the contribution of its motion to J at $\delta(x_p - v_p)$ and δx_p corresponding to the cell in which the particle reside before and after the motion respectively, according to the first-order shape function again, as shown in Figure 1.

Field-solve is to update E and B with J using the leapfrog method to solve Maxwell's equations.

The fundamental problem in PIC implementation on manycore processors lies in particle-push and current-scatter in which E, B and J at the grid points δx_p are accessed for each x_p. That is, if particles are ranked in a one-dimensional array (or a set of them) in the order of processing but randomly in their positions, three-dimensional field-arrays for E, B and J are accessed randomly to make SIMD-vectorization of these hot-spot kernels ineffective, hard or even impossible. More importantly, even if particles are sorted according to their resident cells (e.g., in lexicographical order of positions of cells) so that field-array components accessed for contiguously ranked particles are common, SIMD-vectorization will not work well unless our compiler recognizes the fact of sorted conformation of particles and the commonality of filed-array accesses.

To solve the problem, we devised two efficient and combined implementation techniques; namely *strict particle binning* to rank all particles in a cell in a portion of a structure-of-array (SOA) type set of one-dimensional arrays for vector components of particle positions and velocities; and *field-array scalarization* to cache all components of field-arrays commonly accessed by particles in the cell into a set of local scalar variables explicitly showing the compiler that they are invariant in the loop scanning particles in the cell. More specifically, particles are stored in the set of six one-dimensional arrays for positional and velocity vector components, while cells are represented by an array-of-structure (AOS) type three-dimensional arrays to keep the location and size of each particle bin in the particle arrays. As shown in Figure 2, particle bins are separated by a certain amount of gaps in order to reduce the possibility of mutual collision of them, or *particle overflow* in other words, due to the inter-cell transfer of particles.

With the conformation of cells and particles above, particle-push and current-scatter are implemented by the following four loops to scan particles in a cell, first three of which are enclosed by a triple nest of cell scanning loops while the last one is in another nest of cell scanning loops.

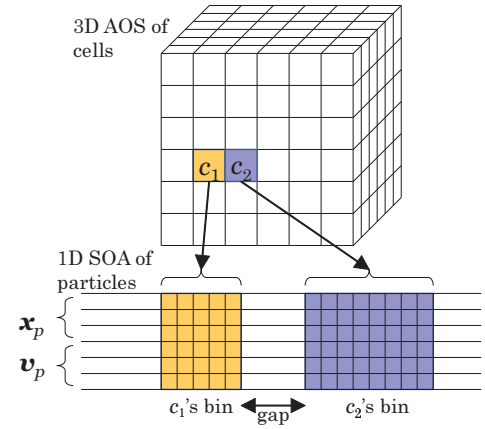

Fig. 2 Cells and particle bins.

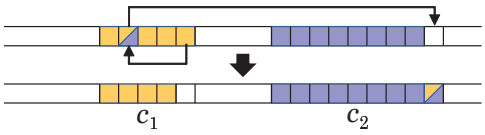

Fig. 3 Moving a particle from a bin to another.

Loop-1 is for the first half of particle-push to update v_p for all p in a cell c referring to $48 = 2 \times 8 \times 3$ scalarized components of E and B around c and common to all p. Since the arrays for x_p and v_p are accessed sequentially and the updates of v_p are independent of each other, this loop is perfectly SIMD-vectorized.

Loop-2 is for the first half of current-scatter to update $12 = 4 \times 3$[*1] scalarized J's components around the cell *before* the motion of the particles in it, and then to update x_p for the second half of particle-push remembering the cell to which p moves by an integer code $-13 \le d_p \le 13$ corresponding to $\{-1, 0, 1\}^3$. Since the arrays for x_p and v_p are accessed sequentially again, this loop is perfectly SIMD-vectorized thanks to vectorized accumulation of scalarized $J's$ components. After the completion of the loop, the scalarized J's components are written back to the array of J.

Loop-3 is to move each particle p whose moving code d_p is not for $(0, 0, 0)$ from its original bin to another to keep the strict binning by this *on-the-fly* sorting. As shown in Figure 3, a particle moved from a cell (c_1) to another (c_2) is placed at the tail of the destination bin (for c_2), while the vacancy in the source bin (for c_1) is filled by the particle at the tail of the bin itself. This *sorting* is efficient because its temporal cost is proportional to the number of particles moving across cell boundaries rather than that of whole simulated particles thanks to the fact that particles in a bin may be ordered arbitrarily, as far as we can find rooms for moving particles in the gaps between bins.

[*1] Not $24 = 8 \times 3$ because in our formulation a velocity vector component, say x-component, contributes only to x-components of J's vectors at vertices of a cell surface.

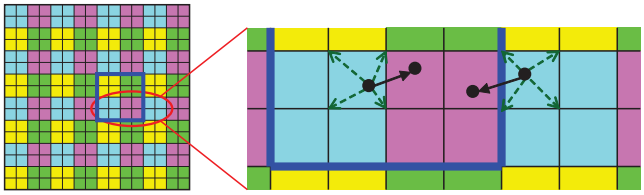

Fig. 4 Cell coloring for conflict-free current-scatter and particle move.

Loop-4 is for the second half of current-scatter to update scalarized $\boldsymbol{J}$'s components around the cell *after* the motion of the particles staying in or moving into it. As well as the Loop-2, this loop is perfectly SIMD-vectorized and is followed by the write-back of the scalarized $\boldsymbol{J}$'s components.

Note that in ordinary implementations these four loops are fused into one loop to minimize the number of accesses to particle components, but we split them by the following reasons. Loop-2 is split from Loop-1 to reduce register pressure caused by 48 components of $\boldsymbol{E}$ and $\boldsymbol{B}$ in total which occupy at least $48/w$ vector registers of w double-precision floating-points (DPFP) for each even with an ISA capable of register operand broadcast and/or broadcast instructions[*2], and by 12 components of $\boldsymbol{J}$ which need 12 vector registers for vectorized accumulation. Loop-3 is split from Loop-2 because the operation to move particles across cell boundaries is hardly SIMD-vectorizable and thus fusing these two inhibits the vectorization of Loop-2. Finally, Loop-4 is split from others because, in order to SIMD-vectorize the loop with scalarization, all particles must be in the bins corresponding to their position after their move.

In addition, in order to have many threads, up to 240 in Xeon Phi KNC, work on each of four loops above in parallel, we apply *coloring* to cell blocks. As shown in Figure 4, we assign a cell block being a set of cells in the shape of cuboid to a thread and then decompose the block D-dimensionally ($D = 2$ in the figure) to have 2^D sub-blocks giving 2^D colors to each of them. Then the cell scanning loop nest surrounding Loop-1, -2 and -3 and that surrounding Loop-4 are also *colored* so that at a time all threads work on their own cells of a particular color and particles in them with a barrier synchronization on color switching. Since in current-scatter we update J's components at the vertices of a particular cell, it is obvious that updates made by a thread cannot collide with those by other threads, if a sub-block has at least two cells along each of decomposed axis. This conflict-free property of the updates in Loop-2 and Loop-4 is also seen in moving particles across cell boundaries in Loop-3, because a particle may travel to one of the cells adjacent to the cell in which the particle resided to make the update of the destination bin conflict-free too.

3. Multi-Node Implementation

3.1 Baseline Implementation

Our multi-node implementation is based on EMSES[1] in which the load-balancing library OhHelp[4] is used to decompose the simulated space domain regularly but to assign up to two decomposed subdomains, *primary* and *secondary* ones, to a process so that processes whose primary subdomains have particles more than per-subdomain average, or *helpand* processes in short, are helped by other *helper* processes. One important feature of EMSES is that each process has *halo* cells surrounding its primary and secondary subdomains to keep particles *logically* going out from the subdomains from *physically* transferred to other processes. That is, EMSES does not perform inter-process particle transfer until some processes find their particles go out from their halo cells, drastically reducing the frequency of inter-process transfers and load-balance managements taken by OhHelp.

By this optimization with halo cells and OhHelp's inherent mechanism to keep good load-balancing, EMSES shows excellent scalability. For example, our performance evaluation with 128-node/4096-core Cray XE6 given in [1] shows 70 % and 66 % parallel efficiency, just 1-2 % less than the 1-node/32-core efficiency, for the settings with perfectly uniform distribution of particles and with extremely congested one in which all particles are in 1/4096 of the whole space domain, respectively.

3.2 Intra-Process Particle Distribution

An important implementation issue of our OhHelp'ed multi-node implementation is that OhHelp is unaware of particle position when it takes care of inter-process particle transfer[*3]. More precisely, OhHelp's level-2 interface[2] assumes the following properties.

(1) Particles are stored in a one-dimensional AOS-type array, while we use SOA-type arrays for six components.
(2) Particles are packed in the array, while our arrays consist of a set of particle bins separated by inter-bin gaps.
(3) Particles which a process receives may be stored in any locations of the array[*4], while we need to store them in corresponding bins.
(4) Particles which a process sends and their destination processes may be chosen arbitrarily[*5].

Since the first three properties are logically inconsistent with our implementation, we have to abandon the easy way to rely on OhHelp's particle transfer mechanism. Therefore, we use level-1 interface[*6] of OhHelp by which we are informed of the number of particles to be sent and received to/from other specific processes to keep good load-balancing, while we may (or must) design our own particle transfer mechanism. In summary, we pick particles to be sent, send/receive identifiers of cells to which the particles belong to, send/receive each of components of particles, and then stores received components into corresponding bins consulting the received cell identifiers.

On the other hand, we may assume the fourth property in our own mechanism to choose particles to be sent and their destination processes, because any choices are logically correct. Suppose we have a set of particles P belonging to a subdomain n which the process n takes care of as its primary subdomain while another helper process m does as its secondary one. In this case,

[*2] Xeon Phi KNC has a limited operand broadcast functionality to allow four scalar DPFPs are in a vector register with a duplication.

[*3] OhHelp has position-aware extensions for particle collision and SPH, but they are incompatible with our aim.
[*4] At the tail of the array, in fact.
[*5] By a simple FCFS-like match-making with ordered set of particles and that of recipients.
[*6] Together with level-3 interface for inter-process communication of field arrays.

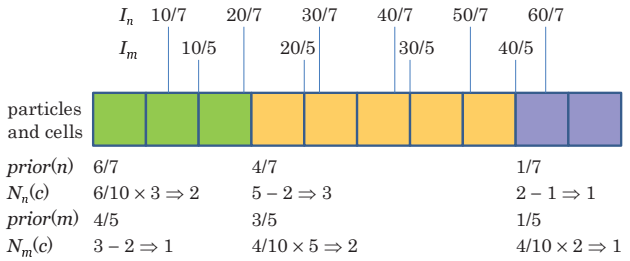

Fig. 5 Intra-process distribution of 10 particles in 3 cells for two processes n and m such that $|P_n| = 6$ and $|P_m| = 4$.

OhHelp tells us the cardinality of two partitions of P, namely $|P_n|$ and $|P_m|$ where $P_n \cup P_m = P$ and $P_n \cap P_m = \emptyset$, while any conformation of P_n and P_m is logically correct. For example, we may let P_n and P_m have particles whose indices in the SOA-type arrays are small and large, respectively.

However, such a simple partitioning of P must bring us severe performance problems because it is inconsistent with our intra-process decomposition of the subdomain n. That is, if $|P_n| \approx |P_m|$ and particles in P are uniformly distributed in n or a *slab* nearby its surface, the partitioning gives P_n and P_m particles residing lower and upper half of n in some sense resulting in a thread-level load imbalance because only a half of threads work on the received particles. In addition, since the subdomain n is somewhat congested, it is expected that cells accommodating particles P are also congested resulting in that the processes n and m have unusually congested cells whose bins are easily overflown.

Therefore, we take more care about the partitioning so that P_n and P_m have particles whose spatial distribution is similar to P as much as possible. This intra-process uniform distribution is accomplished as follows. Let $P = \{p_0, \ldots, p_{|P|-1}\}$ whose order corresponds to the spatial distribution of particles in P, e.g., the lexicographical order of cells accommodating the particles in the subdomain n. Also let I_k where $k \in \{n, m\}$ be the set of conceptual indices of rational numbers, $I_k = \{(i \cdot |P|)/(|P_k| + 1) \,|\, 1 \le i \le |P_k|\}$. Then we sort a set $I = \{(i, k) \,|\, i \in I_k,\ k \in \{n, m\}\}$ to have ascending sequence $I = \{(i_0, k_0), \ldots, (i_{|P|-1}, k_{|P|-1})\}$ by which we let $P_k = \{p_j \,|\, (j, k) \in I\}$, as shown in the upper half of Figure 5.

For the real implementation, however, we devised the following more efficient algorithm by which almost equivalent distribution is accomplished without the sorting as exemplified in the lower half of Figure 5.

(1) Let $prior(k) = |P_k|/(|P_k| + 1)$ and $share(k) = |P_k|/|P|$ for $k \in \{n, m\}$.
(2) Visit all cells c such that $\pi(c) \subseteq P$, where $\pi(c)$ is the set of particles accommodated by c.
(3) At each visit of c, assign $N_k(c) = round(share(k) \cdot |\pi(c)|)$ particles to $k = \arg\max\{prior(k) \,|\, k \in \{n, m\}\}$ and $N_l(c) = |\pi(c)| - N_k(c)$ particles to $l = \arg\min\{prior(l) \,|\, l \in \{n, m\}\}$. Then let $prior(k) \leftarrow prior(k) - (N_k(c)/(|P_k| + 1))$ for $k \in \{n, m\}$.
(4) Periodically update $share(k)$ so that it approximates $(|P_k| - \sum N_k(c))/(|P| - \sum(N_n(c) + N_m(c)))$ where the summation is for all c which we have already visited[*7].

Note that the real implementation is of course capable with any number processes rather than two, and the prioritized assignment is efficiently implemented with a heap structure for $prior(\cdot)$. Also note that the periodical update in (4) is to avoid frequent floating-point divisions which we would have to perform in each visit of cells, while the update of $prior(\cdot)$ does not require a division because $N_k(c)/(|P_k| + 1)$ is approximated by $N_k(c) \cdot (1/(|P_k| + 1))$ with a sufficient accuracy.

[*7] This operation is not shown in Figure 5.

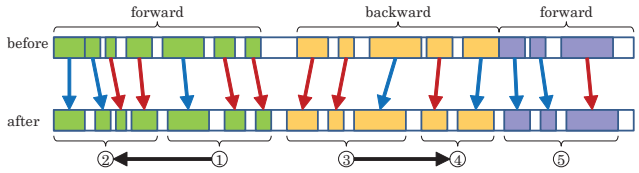

Fig. 6 Shifting three sequences of particle bins, forward for two and backward for one, by two (red and blue) threads.

3.3 Particle Overflow

3.3.1 Particle Bin Shift

On-the-fly sorting performed by Loop-3 shown in Section 2 may face *particle overflow* when the destination bin of a particle does not have a gap between the bin following it. If it happens, fundamentally we have to rearrange all bins, by resizing them not only for enlarging the overflown bin but also to give all bins sufficiently large gaps between them according to the number of particles in them, and by an *in-place shift* of them according to the resizing result.

Though the in-place bin-shift is an $O(N)$ procedure whose temporal cost is proportional to the number of particles which a process accommodates, we need to make the cost as small as possible because at least its memory access cost is comparable with particle-push and current-scatter. Therefore, we need to make the shifting procedure multithreaded and SIMD-vectorizable. Our bin-shift has the following two steps. First we find a sequence of bins whose *moving directions* are *coherent*, i.e., *forward* toward larger indices or *backward* toward smaller indices as shown in Figure 6. Then second, for each sequence of bins having a coherent direction, we perform multithreaded shift assigning bins in the sequence to threads so that each thread has as many bins as possible with a constraint that the total particle count in the bins does not exceed the size of a thread-local buffer through which particles are shifted. That is; (a) we scan bins in a forward (resp. backward) sequence descendingly (resp. ascendingly) assigning them to threads; (b) let all threads copy particles in their bins into their buffers; (c) after a barrier synchronization, let all threads make a copy again but this time from buffers to bins in shifted locations; and repeat (a)–(c) until all bins in the sequence are processed. Note that the copy from/to bins to/from buffers is obviously SIMD-vectorizable, and shows a good access locality with buffers of a size able to be accommodated in, e.g., secondary caches because the shift amount is expected not so large.

3.3.2 Particle Bin Resizing

Prior to shifting bins, we have to resize particle bins enlarging and shrinking inter-bin gaps to make the possibility that the overflow happens again and soon as small as possible. We determine the bin size of a cell c, namely $B(c) = |\pi(c)| + g(c)$ where $g(c)$ is the size of the gap following the particles in $\pi(c)$, by two factors.

One is the fixed constant gap g_f to make $g(c) \geq g_f$ regardless $|\pi(c)|$[*8]. The other gap amount $g_v(c)$ is determined by $|\pi(c)|$ to let $g(c) = g_v(c) + g_f$ exploiting the space preserved by enlarging the size of SOA-type arrays N_{array} from its minimum requirement N_{min} by a margin factor. That is, with a given per-process average number of particles N_{ave} and load-imbalance tolerance factor $\alpha_t \geq 1$, N_{min} is defined as $N_{\text{min}} = \alpha_t \cdot N_{\text{ave}}$, while N_{array} is determined by the margin factor $\alpha_m \geq 1$ as $N_{\text{array}} = \alpha_m \cdot N_{\text{min}} + |C_n| \cdot g_f$, where C_n is the set of all cells which a process n has. Since the number of particles N_n which the process n accommodates should satisfy $N_n \leq N_{\text{min}} \leq N_{\text{array}}$, we have an unoccupied space of $N_{\text{array}} - N_n$ for gaps in total of all bins in n. More specifically, we have $N_{\text{array}} - N_n = \sum_{c \in C_n} g(c) = |C_n| \cdot g_f + \sum_{c \in C_n} g_v(c)$. Therefore, what we can do for the reduction of overflow frequency is to determine each $g_v(c)$ appropriately and according to $|\pi(c)|$ with the constraint $\sum_{c \in C_n} g_v(c) = N_{\text{array}} - N_n - |C_n| \cdot g_f = M_n$.

Intuitively, it looks natural to make $g_v(c)$ proportional to $|\pi(c)|$, i.e., $g_v(c) = (|\pi(c)|/N_n) \cdot M_n$ because the more a bin has particles, the more quickly the population will grow to require a wider gap. However, since this simple allocation is unaware of how many particles a bin *has had* for past simulation steps, a cell can suffer repetitive overflows if it has *oscillating* population of particles. Worse, such oscillation might be caused by halo-cells by which we reduce the number of occurrences of inter-process particle transfers.

Suppose we have a steady eastbound stream of uniformly distributed particles. Without halo-cells such a stream should cause frequent inter-process particle transfers crossing easternmost boundary surface of subdomains, while per-cell particle density would be kept stable because of the steadiness of the stream and the uniformity of particle distribution. With halo-cells, on the other hand, the frequency of inter-process transfers is drastically reduced, but we have instability of particle populations at halo-cells just outside the easternmost surface and ordinary cells just inside the westernmost surface. That is, the particle population of a halo-cell c_e at the easternmost surface should grow steadily to cause repetitive overflows in which its bins become larger and larger because of the growing $|\pi(c_e)|$. On the other hand, since the corresponding ordinary cell c_w at the westernmost surface does not have any incoming particles but has steadily outgoing ones, $|\pi(c_w)|$ steadily decreases to make its bin smaller and smaller each time we have overflow at c_e. Then we have a particle going out from c_e and thus the process accommodating it to make c_e empty by throwing all particles in it away to the east-neighbor process, while c_w suddenly receives many particles from the west-neighbor to cause overflow of its bin. Therefore, c_w's bin should be inflated again while c_e's should become deflated to minimum size g_f because $|\pi(c_e)| = 0$. Then the story repeats again and again with periodically overflow of c_e during the steps without inter-process transfer, and that of c_w at the transfer.

To cope with such oscillatory repletion and depletion including *artificial* ones which we see in c_e and c_w, we use the margin M not only for enlarging repletive bins but also for keeping the size of (temporarily) depletive bins as large as possible, as follows. We partition the set C_n into two subsets, $C_n^+ = \{c \mid |\pi(c)| > B(c) - g_f\}$ and $C_n^- = \{c \mid |\pi(c)| \leq B(c) - g_f\}$, i.e., according to the *criticality* of the repletion level of bins. Then for $c \in C_n^+$ we let $g_v(c) = (|\pi|/N_n) \cdot M_n$ to enlarge their bins including those overflown because such cells obviously in C_n^+. On the other hand, for $c \in C_n^-$ we let $g_v(c)$ as follows, where $R(c) = B(c) - g_f - |\pi(c)|$.

$$g_v(c) = \frac{R(c)}{\sum_{c \in C_n^-} R(c)} \cdot \frac{\sum_{c \in C_n^-} |\pi(c)|}{N_n} \cdot M_n$$

The equation above means that the space of $(\sum_{c \in C_n^-} |\pi(c)|/N_n) \cdot M_n$ remaining for $c \in C_n^-$ is given to their bins proportional to the size $R(c)$ of their *rooms* in order to keep the room size as large as possible. Since this resizing is aware of the fact that a cell had been replete somewhat *remembered* in its bin size and thus in $R(c)$, in the steady flow case the bins of c_e and c_w are kept sufficiently large to avoid repetitive overflows providing M_n is sufficiently large as well.

3.3.3 Overflow Buffering

Though the resizing shown in Section 3.3.2 is aware of the past, it cannot be aware of the *future* of course. For example, if a small but congested mass of particles moves along a way, the cells on the way should suffer overflow due to the unusual denseness. Unfortunately, such a *local* congestion may occur even in the case that the particle density is smoothly changed without any local anomalies when we have a *global* deviation of the density, by an artifact of OhHelp's load-balancing mechanism.

That is, with the global deviation, OhHelp occasionally performs the reassignment of secondary subdomains to reestablish perfect balancing when the past assignment is unable to keep good balancing due to the change of spatial conformation of the deviation by, e.g., the move of the large mass with relatively high density. This rebalancing could give a secondary subdomain m with quite small number of particles to a process n when its primary subdomain is also congested. Then suppose the mass moves to make the secondary subdomain denser and thus the load of the process n heavier. This increase of the load, however, may be suppressed in a short period due to the sufficiently heavy load of n's primary subdomain so that other processes also responsible for the subdomain m receive more incoming particles. Therefore, since the secondary subdomain m in the process n was quite *rarefied* at the rebalancing, n has a small mass of artificial local congestion due to a small number of inter-process transfers and will suffer repetitive overflows caused by the move of the artificial mass.

In order to avoid such a repetition caused by a small number of particles overflown from bins, we keep them in thread-local *overflow buffers* instead of performing the bin rearrangement immediately at each overflow. That is, overflown particles are stored in the buffer and on them particle-push and current-scatter are performed in a *SIMD-unaware* manner, i.e., with random accesses to the array of $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$ according to their positions. For efficiently multithreading the operations, however, we preserve the conflict-free property of current-scatter and inter-cell transfer by giving colors and sub-colors to the buffers. As shown in the left half of Figure 7, each thread has 2^D sets of colored buffers each of which consists of 2^D buffers with sub-colors (circled numbers

[*8] More precisely, for cells outside halo cells, their $g(c)$ is always $g_f/2$.

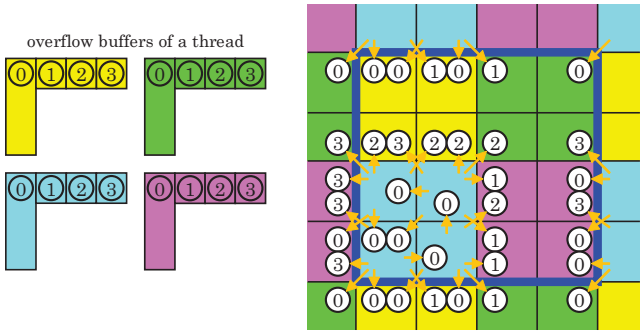

Fig. 7 Colored overflow buffers of a thread and the correspondence of their sub-colors and the direction of particle motions.

in the figure), one is large and others are small, where D is the dimensionality of the cell block decomposition.

That is, a particle overflown from the bin of a cell of a color is stored in one of the buffers of the same color. As for the choice of sub-color of the destination buffer, it is determined as follows. If the source and destination cells of the particle motion share a color, the destination buffer is *major* one of sub-color 0 because the move cannot conflict those made by other threads. Otherwise, i.e., if the particle crosses a colored sub-block boundary, the destination being major or one of *minors* is determined by the direction of the motion as shown in the right-half of the figure so that moves made by multiple threads to a particular cell or cell sub-block are conflict free. As for particle-push and current-scatter for the particles in the buffers, their conflict-free multithreading is easily implemented by performing them color-by-color with barrier synchronizations on color switching.

Though the buffering will significantly reduce the frequency of bin rearrangement, we need to perform them emptying all buffers in the following cases.

- A particle overflown from a cell also causes overflow of its destination buffer. If this happens, we have to suspend the push operations of particles in the ordinary bins or buffers, perform the bin rearrangement, and then resume the pushes.
- The cumulative number of particles processed in the buffers exceeds a threshold, because SIMD-unaware particle-push and current-scatter on particles in buffers are significantly slower than their SIMD-aware counterparts on those in bins. More specifically, we accumulate the per-buffer maximum number of particles among all threads and sub-colors and, if it exceeds a threshold θ being the double of the size of a major buffer, the rearrangement takes place.
- Inter-process particle transfer takes place, because it is extremely tough to perform the position-aware transfer with particles in the buffers whose positions are random. Similarly, if it is known that particles incoming to a process cause overflow, the rearrangement takes place as well prior to their reception.

4. Performance Evaluation

4.1 Environment and Settings

We implemented the prototype with C99 and OpenMP 3.0, compiled it by Intel Composer version 14.0.5 with `-ipo` and `-O3` options of optimization, and linked it with Cray MPI version 7.1.3. The prototype is run on up to 32 nodes of Cray X30, whose node is comprised of a Intel Xeon Phi 5120D, whose peak DPFP performance 1.01 TFlops is given by 60 cores of 1.053 GHz, hosted by a Xeon E5-2670v2 (Ivy Bridge), in *native-mode* to make all computations done only by Xeon Phi.

Table 1 (a) System domain size according to the number of nodes N, and (b) subdomain size according to the number of per-node processes P.

N	system domain size
1	$30\cdot1\times60\cdot1\times120\cdot1 = 30\times 60\times120$
2	$30\cdot2\times60\cdot1\times120\cdot1 = 60\times 60\times120$
4	$30\cdot2\times60\cdot2\times120\cdot1 = 60\times120\times120$
8	$30\cdot4\times60\cdot2\times120\cdot1 = 120\times120\times120$
16	$30\cdot4\times60\cdot2\times120\cdot2 = 120\times120\times240$
32	$30\cdot4\times60\cdot4\times120\cdot2 = 120\times240\times240$

(a)

P	subdomain size
1	$30/1\times60/1\times120/1 = 30\times60\times120$
2	$30/1\times60/1\times120/2 = 30\times60\times 60$
3	$30/1\times60/1\times120/3 = 30\times60\times 40$
4	$30/1\times60/2\times120/2 = 30\times30\times 60$
5	$30/1\times60/1\times120/5 = 30\times60\times 24$
6	$30/1\times60/2\times120/3 = 30\times30\times 40$
10	$30/1\times60/2\times120/5 = 30\times30\times 24$
12	$30/1\times60/2\times120/6 = 30\times30\times 20$
15	$30/1\times60/3\times120/5 = 30\times20\times 24$
20	$30/1\times60/4\times120/5 = 30\times15\times 24$
30	$30/2\times60/3\times120/5 = 15\times20\times 24$
60	$30/2\times60/3\times120/10 = 15\times20\times 12$

(b)

For the test simulations, we have two cases with artificial settings of particle distribution namely *uniform* and *congested* ones. In the latter case, particles are also uniformly distributed but in a *quadrant* of the system domain, $[0,S_x/2)\times[0,S_y/2)\times[0,S_z)$ at initial where S_x, S_y and S_z are the sizes of the system domain along three axes, to cause severe imbalance of particle distribution. In both cases, the following common settings are used.

- All particles of $128\times S_x\times S_y\times S_z$ (i.e., average per-cell density is 128) are partitioned into two *species* one of which steadily travels to east and the other to west with common velocity 1/64 in the system domain, whose boundaries are fully periodic, in the simulation of 2000 time steps. Note that categorizing particles into a few species saves the amount of memory space required by particles because the mass m_p and charge q_p are common for all p of a particular species, as done in many of PIC simulators. Also note that the velocity 1/64 reflects the fact that in production-level simulation with EMSES the per-cell and per-step number of incoming (and thus outgoing) particles is 1–2 % of the total number, while it is 1.56 % to have $2 = 128\times(1/64)$ particles crossing a cell-boundary on average in this setting.
- To each node, a fixed size cuboid grids of $30\times60\times120$ is allocated. Then using this unit cuboid as the building block, N-node weak-scaling simulations such that $N\in\{1,2,4,8,16,32\}$ are carried out for system domains of $S_x\times S_y\times S_z = 30N_x\times60N_y\times120N_z$ where $N_x\cdot N_y\cdot N_z = N$ so as to minimize the sum $S_x+S_y+S_z$ and to make $S_x\le S_y\le S_z$ for tie break, to have domain size shown in Table 1(a). With the per-cell average density of 128, we have nearly 1 billion particles in 32-node executions, or 0.88×10^9 of them more accurately.

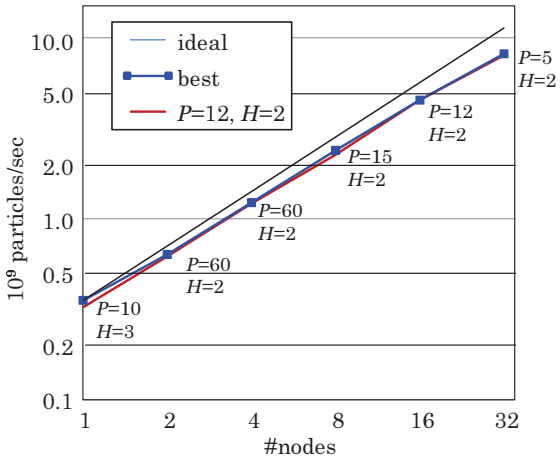

Fig. 8 Performance of uniform case with best process/hyper-threading configuration (blue) and fixed one with $P = 12$ and $H = 2$ (red).

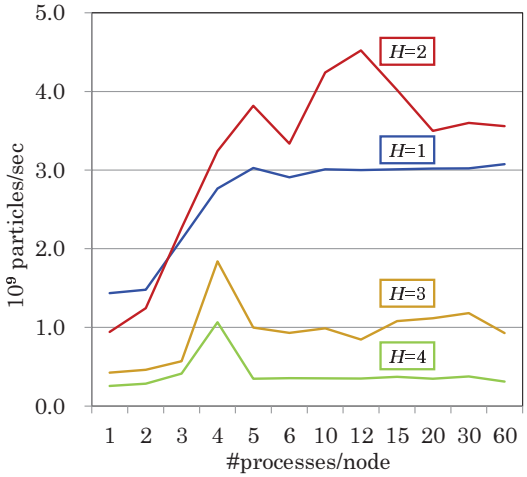

Fig. 9 16-node performance of uniform case with all $12 \times 4 = 48$ configurations of P and H.

- The cuboid for a node is decomposed for P intra-node processes such that P is one of 12 divisors of the core count 60 to have one of 12 process configurations $P_x \times P_y \times P_z = P$ and per-process cuboids of $s_x \times s_y \times s_z = 30/P_x \times 60/P_y \times 120/P_z$ to be given to processes as their primary subdomains, as shown in Table 1(b). In addition to the cells in the cuboid, each process has two types of halo cells surrounding its primary and secondary subdomains; *inner* ones just outside the cuboid to keep particles going out from subdomains in multiple time-steps; and *outer* ones just outside the inner halo cells by which particles are temporarily accommodated because the intrusion of a particle to such a cell triggers inter-process particle transfer at the end of a time-step.
- The subdomain cuboid for a process is decomposed for $T \times H$ threads such that $T \times P = 60$ and $H \in \{1, 2, 3, 4\}$ for hyper-threading. The decomposition is 2-dimensional and is applied to yz-plane so that threads have *beams* and form an array of $T_y \times T_z = T \times H$ so as to minimize $s_y/T_y + s_z/T_z$ and then to minimize $T_y + T_z$ for tie break[*9]. Note that both inner and outer halo cells are given to threads whose beams have surfaces being those of a subdomain cuboid as well.
- Other configuration parameters are set as follows; load imbalance tolerance factor $\alpha_t = 1.1$; the margin factor of particle arrays $\alpha_m = 1.2$; fixed constant gap between two bins $g_f = 8$; the size of thread-local buffer for bin shift is 8192; the size of major overflow buffers is $(\alpha_m \cdot N_{\min})/(16 \cdot T \cdot H)$ while that of minor ones is $(\alpha_m \cdot N_{\min})/(128 \cdot T \cdot H)$.

4.2 Uniform Case

Figure 8 shows the performance of the uniform case in terms of the number of particles processed (e.g., pushed) in one second. The blue line in the chart is for the best performance with each node count among $12 \times 4 = 48$ combinations of P and H (shown in the chart), while the red line almost overlapping the blue one is for the fixed configuration of $P = 12$ and $H = 2$ by which we have at least 91.6 % ($N = 1$) of the best performance. The fact that both curves (almost lines) have slopes slightly gentle compared with the ideal line means that the parallel efficiency gradually decreases as the node count N increases resulting in the performance approximately proportional to $N^{0.91}$ rather than N, probably because of the slow global all-reduce on an integer due to long latency caused by the host processor interposed in the physical communication path. Nevertheless, the best 32-node performance 8.15×10^9 particles/sec achieved by 1920 small cores of 1.053 GHz surpasses 7.81×10^9 which EMSES on 64-node Cray XE6 managed to perform with 2048 big cores of 2.5 GHz[1].

It could be surprising that the best configurations shown in the chart have large P up to its maximum 60 ($N \in \{2, 4\}$) and small $H = 2$ for all node counts except for $N = 1$. This observation is also supported by the nearly-optimal fixed configuration whose $P = 12$ is larger than $T \cdot H = 8$. That is, it looks that we need some large number of per-node processes though it incurs overhead of intra-node inter-process communication which should be significantly larger than that of intra-node/process inter-thread communication. Note that in this uniform case, per-thread and per-process load are almost perfectly balanced to make the result more surprising.

These surprising tendencies are also confirmed by a more detailed 16-node performance chart shown in Figure 9 for all $12 \times 4 = 48$ configurations of P and H. From this chart, we clearly see that the performance superiority of hyper-threading degree H is in the order $2 \to 1 \to 3 \to 4$, while our one-node single-process implementation given in [3] showed us a different order $3 \to 2 \to 1 \to 4$ with a significant outperformance of $H = 3$ over $H = 2$. As for the per-node process count, we see a clear peak at $P = 12$ (and a mysterious valley at $P = 6$) in the case of $H = 2$, while other cases show a gradual up-slope to $P = 5$ and a plateau in $P \geq 5$ if we ignore mysterious peaks at $P = 4$ in the cases of $H = 3$ and $H = 4$.

In order to find the reason why we need some many processes in a node, we measured 16-node performance of $H = 2$ again but with two extremely artificial settings. That is, the blue line of Figure 10 is for the setting that velocity of all particles is 0 to let them stay in their initial position. The other setting for the light-brown line of the figure is *logically incorrect* because we inhibit the inter-process communication of arrays for $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$, while

[*9] If we still have ties, they are broken by choosing the minimum T_y.

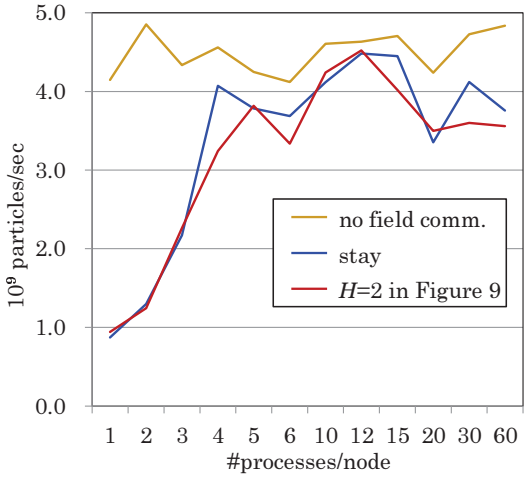

Fig. 10 16-node performance of uniform case with all 12 configurations of P and $H = 2$ together with extremely artificial settings to let particles stay and to omit inter-process communications of $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$.

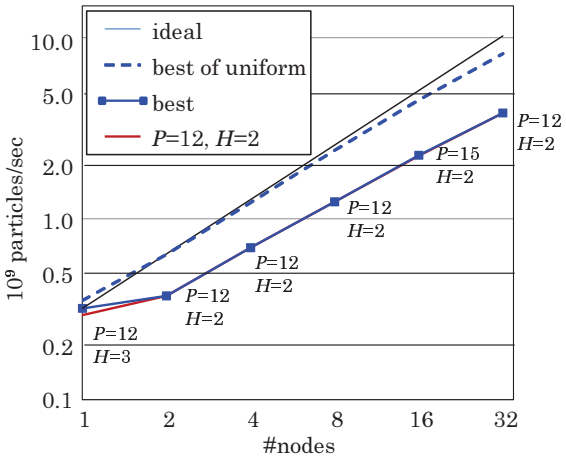

Fig. 11 Performance of congested case with best process/hyper-threading configuration (blue) and fixed one with $P = 12$ and $H = 2$ (red) together with the performance of uniform and best-configuration case (dashed blue).

particles travels exactly same as the original setting[*10] and thus inter-process transfers take place. These two evaluations reveal the reason why we have the optimal process count $P = 12$. First, the setting to let particles stay does not show much difference from the original though the elimination of inter-cell and inter-process particle transfers seem to give some advantage over the original.

Second and much more importantly, the elimination of inter-process communication of field-arrays makes the performance almost equal to the best performance of original setting almost regardless of per-node process counts. The reason why we have this result is explained as follows. For $P < 10$, inter-node communications are performed by less than ten processes and thus ten low-performance cores which are insufficient to fill the inter-node communication bandwidth and thus to mitigate the long latency due to the host processor, while at least ten processes at each *surface* of the cuboid for a node work on the communication in the cases of $P = 10$ and $P = 12$. For $P > 12$, on the other hand, we have a significantly large amount of inter-process but intra-node communications to consume the memory bandwidth at the communication, in addition to the smaller size of transferred data to make it tougher to mitigate the effect of long latency. Therefore, it is expected that the next-generation hostless Knights Landing (KNL) version will make a large improvement of the performance with a fewer number of per-node processes with which inter-process communication overhead will be greatly reduced.

On the other hand the reason why $H = 2$ is the best and $H = 3$ is inferior to not only $H = 2$ but also $H = 1$ is still mysterious, because the elimination of field-array communications does not improve the best performance 1.84×10^9 so much though the performance becomes less dependent on per-node process count to result in the range $1.63 \times 10^9 - 1.88 \times 10^9$. A further investigation of this phenomenon and of the fact the parallel efficiency of executions with $H = 3$ rapidly decreases as node count increases is left as our future work.

[*10] Since the setting of steady travel with a constant velocity is achieved by making Lorentz force exerting on any particles is $\boldsymbol{0}$, inhibiting the inter-process communication resulting in incorrect $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$ does not affect the particle motion.

4.3 Congested Case

Figure 11, 12 and 13 show performance numbers of the congested case corresponding to those shown in Figure 8, 9 and 10 respectively for the uniform case, some of which are given in the charts by dash lines. That is, we present Figure 11 for the performance with the best and nearly-optimal fixed configurations of per-node process count $P = 12$ and hyper-threading degree $H = 2$, Figure 12 for 16-node performance with all possible combinations of P and H, and Figure 13 for that with $H = 2$ and the settings to let particles stay at their initial positions and to eliminate inter-processes communications of field-arrays.

From Figure 11, the best configuration is given by $P = 12$ and $H = 2$ again, except for the cases of $N = 1$ and $N = 16$ for which the nearly-optimal configuration gives 91.9 % and 99.1 % of the best respectively. The other observation from the figure is that the performance slope for $N \geq 2$ is a little bit gentler than that in the uniform case to result in the performance approximately proportional to $N^{0.84}$ probably due to the all-reduce on $\boldsymbol{J}$ discussed later.

More clearly, the absolute performance is significantly lower than that in the uniform case, about the half, except for the single-node case. That is, though the best 32-node peformance 3.84×10^9 particles/sec still surpasses 3.73×10^9 of the 32-node execution of EMSES on Cray XE6 with a similar congested formation, the advantage over the predecessor is greatly reduced. Figure 12 also shows this degradation, while the performance superiority order of hyper-threading degrees H is the same as the uniform case, i.e., $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$ but peaks and valleys in the chart are less distinctive. There are various reasons that a congested-case simulation is less efficient than a uniform-case one as follows.

(1) Though OhHelp tries to keep good load balancing, the tolerance factor $\alpha_t = 1.1$ makes a process loaded more heavily, up to 10 %, than the processes in the uniform case.
(2) Each process must perform field-solve twice for its primary

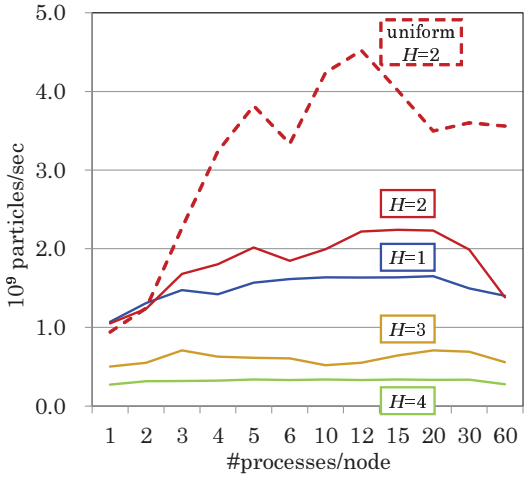

Fig. 12 16-node performance of congested case with all $12 \times 4 = 48$ configurations of P and H together with the performance of uniform case with $H = 2$ (dashed red).

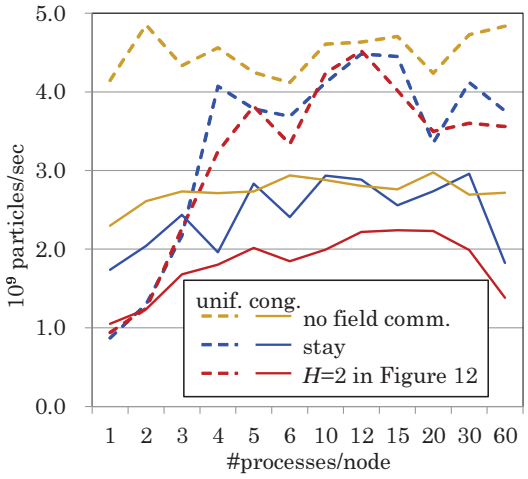

Fig. 13 16-node performance of congested (solid) and uniform (dashed) cases with all 12 configurations of P and $H = 2$ together with extremely artificial settings to let particles stay and to omit inter-process communications of $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$.

and secondary subdomains, and the helpand process must broadcast $\boldsymbol{E}$, $\boldsymbol{B}$ and $\boldsymbol{J}$ at the boundary of its primary subdomain obtained from neighbors to its helper processes working on the subdomain as their secondary ones.

(3) For $\boldsymbol{J}$, in addition to the boundary communication, the helpand and helper of a subdomain must perform all-reduce summation for the whole of the subdomain.

(4) Since in our setting the mass of the congested particles has a *solid* surface perpendicular to its moving direction, the particle density of the cells in front of the surface should steeply increase to cause frequent particle overflow.

Note that the reasons (1)–(3) are common to all OhHelp'ed simulations including our EMSES, whose congested-case performance is merely about 5 % inferior to uniform-case[1]. Therefore, first suspect is the reason (4), i.e., frequent overflow by which in 1092 time-steps out of 2000 at least one process performs the costly particle bin rearrangement in the simulation of $N = 16$, $P = 12$ and $H = 2$, though our overflow buffering tries to reduce the frequency of the rearrangements.

In fact, the blue solid line labeled "stay" in Figure 13 is clearly above the red solid line of the moving case and the gap between them is clearly wider than the corresponding gap between the dashed blue and red lines for the uniform case. For example, in the case of $P = 12$, moving the mass increases the simulation time by about 30 % whose major part is almost surely brought by the bin rearrangement done a little bit more frequently than every two time-steps.

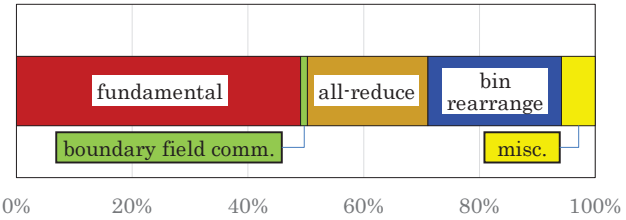

Fig. 14 Hypothetical break-down of 16-node execution with $P = 12$ and $H = 2$.

However, the other overhead of inter-process communications is equally or little bit more heavily affects the performance as we observe from the gap between the brown solid line labeled "no field comm." and the red solid line. Since the gap is also wider than that between corresponding dashed lines around the point of $P = 12$ to bring an overhead less than 3 %, we are almost sure that it is caused by the all-reduce summation of $\boldsymbol{J}$ (3), by which we incur another 26 % overhead. Since this amount is much larger than the 3.2 % overhead we incurred in the execution of EMSES on Cray XE6[1], it should be also charged to the poor single-core MPI performance of Xeon Phi and the host processor on the communication path. In addition, it is believed that this overhead is the reason why we have the parallel performance proportional to $N^{0.84}$ because the all-reduce communication incurs longer latency as the node count increases.

In summary, the execution time of the congested case with $N = 16$, $P = 12$ and $H = 2$ can be hypothetically broken down based on the results of various runs of different settings as shown in Figure 14. In the chart, "fundamental" means the part for the executions required for any distribution of particles, while other portions are caused by the severe imbalance of particle distribution with the mass having solid surfaces. This chart strongly suggests that we will have much better performance with the hostless configuration of KNL, and with typical formations of imbalanced particle distribution expectedly without any solid surfaces of congested particle masses even when they exist.

5. Conclusions

In this report, we showed major implementation issues to have the prototype of manycore-aware OhHelp'ed PIC simulators from the proof-of-concept single-node baseline presented in [3]. The issues include the way to keep thread-level good load balance by distributing particles transferred from a process to another as uniformly as possible in the recipient's subdomain. The other important issue is to manage particle bins and particles overflown from them to reduce the overhead of our on-the-fly particle sorting as much as possible.

From our performance evaluations on Cray XC30 equipped with Xeon Phi 5120D using two artificial settings of particle dis-

tribution and their motion, we confirmed our sophisticated implementation techniques work well especially in the uniform case though we need an unexpectedly large number of processes on a node due to the poor MPI performance of the KNC version of Xeon Phi and its hosted configuration. This poorness also degraded the congested-case performance much more than what we experienced in our previous work with Cray XE6[1], though the parallel performance with respect to the same node count still surpasses that of the predecessor.

Starting from the prototype implementation and its performance, we will go forward to the design of a production-level simulator with the following fairly long to-do list. First, we have to examine the appropriateness of various implementation parameters some of which we decided fairly intuitively. Similar intuitive decisions also may be hidden in our implementation methodologies such as AOS/SOA-choice of data structures some of which could have to be reversed for better performance of, e.g., inter-process communications.

Second, though we believe our sophistications in the implementation should earn some good reward in terms of the performance, the superiority of them over simpler implementations has not been evaluated thoroughly though we have found the necessity of some of them during our development. Therefore, we have to evaluate the effectiveness of them with test cases not only artificial ones shown in this report but also more realistic ones. In these evaluations, of course we should use a KNL-based system which we will have in near future.

Third, though we may expect that the KNL version of Xeon Phi will improve the MPI performance significantly, we need to explore the possibility of improvement in the implementation side especially for the all-reduce on $\boldsymbol{J}$ in imbalanced cases. Since we represent $\boldsymbol{J}$ by SOA-type arrays for three components for the sake of performance of SIMD-vectorized field-solve, we need an all-reduce communication for each component and thus three times in total. In addition, in order to make the reduction done in many communicator for *families* of helpands and their helpers in parallel as much as possible, we repeat the three all-reduce communications twice with a red-black coloring of the families. Therefore we have six serialized all-reduce communications, but the serialization is logically unnecessary to give us the opportunity to use the asynchronous all-reduce introduced in MPI 3.0 to hide the latency of each communication.

Finally, in the development of the production version perhaps with FORTRAN, we will have to avoid to recode the prototype completely but to reuse as a large portion of it as possible. A promising way to do that is to encapsulate complicated functions in the prototype into a kind of library, which perhaps can be a extended part of OhHelp itself. Since more than 70 % of 2622 C source lines of the prototype is not for the kernel operations of PIC simulation, splitting them from the kernel using our domain specific programming framework should greatly help the development.

References

[1] Miyake, Y. and Nakashima, H.: Low-Cost Load Balancing for Parallel Particle-in-Cell Simulations with Thick Overlapping Layers, *Proc. Intl. Symp. Parallel and Distributed Processing with Applications (ISPA)*, pp. 1107–1114 (2013).

[2] Nakashima, H.: OhHelp Library Package for Scalable Domain-Decomposed PIC Simulation, http://www.para.media.kyoto-u.ac.jp/ohhelp/ohhelp-man.pdf (2010).

[3] Nakashima, H.: Manycore Challenge in Particle-in-Cell Simulation: How to Exploit 1 TFlops Peak Performance for Simulation Codes with Irregular Computation, *Computers and Electrical Engineering*, Vol. 46, pp. 81–94 (online), DOI: 10.1016/j.compeleceng.2015.03.010 (2015).

[4] Nakashima, H., Miyake, Y., Usui, H. and Omura, Y.: OhHelp: A Scalable Domain-Decomposing Dynamic Load Balancing for Particle-in-Cell Simulations, *Proc. Intl. Conf. Supercomputing (ICS)*, pp. 90–99 (2009).