OpenMP Extension for Explicit Task Allocation on NUMA Architecture

Jinpil Lee[1(✉)], Keisuke Tsugane[2], Hitoshi Murai[1], and Mitsuhisa Sato[1]

[1] RIKEN Advanced Institute for Computational Science, Kobe, Japan
{jinpil.lee,h-murai,msato}@riken.jp
[2] University of Tsukuba, Tsukuba, Japan
tsugane@hpcs.cs.tsukuba.ac.jp

Abstract. Most modern HPC systems consist of a number of cores grouped into multiple NUMA nodes. The latest Intel processors have multiple NUMA nodes inside a chip. Task parallelism using OpenMP dependent tasks is a promising programming model for many-core architecture because it can exploit parallelism in irregular applications with fine-grain synchronization. However, the current specification lacks functionality to improve data locality in task parallelism. In this paper, we propose an extension for the OpenMP task construct to specify the location of tasks to exploit the locality in an explicit manner. The prototype compiler is implemented based on GCC. The performance evaluation using the KASTORS benchmark shows that our approach can reduce remote page access. The Jacobi kernel using our approach shows 3.6 times better performance than GCC when using 36 threads on a 36-core, 4-NUMA node machine.

Keywords: OpenMP · Task parallelism · NUMA optimization

1 Introduction

Many-core architecture is widely used in High Performance Computing (HPC) since increasing the number of cores is an efficient way to build an energy efficient processor. Along with the trend, Non-Uniform Memory Access (NUMA) architecture has been introduced to provide high memory bandwidth. Modern CPU architecture has multiple NUMA nodes inside a chip (e.g. the latest Xeon processors with the Cluster-On-Die (COD) technology). We expect that this trend will continue and many HPC systems will have many-core processors with multiple NUMA nodes.

OpenMP has been the de facto standard for thread-level parallel programming. In the early version of OpenMP, the programming model had focused on data parallelism described by loop work sharing, which requires global synchronization in a parallel region. When the number of cores increases, synchronization overhead is getting bigger, and load imbalance among cores causes a significant performance drop. Dynamic task generation was introduced in OpenMP 3.0.

© Springer International Publishing Switzerland 2016
N. Maruyama et al. (Eds.): IWOMP 2016, LNCS 9903, pp. 89–101, 2016.
DOI: 10.1007/978-3-319-45550-1_7

In OpenMP 4.0, task dependency can be specified using the depend clause in the task construct. Task parallelism can exploit potential parallelism in irregular applications. Task dependency can reduce synchronization overhead because it generates fine-grain synchronization between dependent tasks.

To exploit memory bandwidth with the NUMA architecture, OpenMP provides thread affinity options through environment variables such as OMP_PROC_BIND. For OpenMP 4.5, the proc_bind clause is discussed to specify a thread affinity scheme for a parallel region. These can be helpful to improve data locality when performing data parallelism with loop work sharing. However, the current specification lacks functionality to do the same thing for task parallelism. A task can be tied to any thread in the parallel region. It will cause unexpected remote page access across the NUMA interconnection.

The aim of our research is to find an explicit way of improving data locality in OpenMP tasks for the NUMA architecture. In this paper, we propose an OpenMP extension to describe NUMA-aware task allocation explicitly. The extension specifies the data that the target task would access. Our compiler implementation, based on GCC, determines the NUMA node that the specified data is allocated and schedules the task to the node. The programmer can distribute data and tasks among NUMA nodes in the same manner by combining our extension and NUMA APIs. This approach can reduce remote memory access and improve memory performance.

The rest of the paper is organized as follows: Sect. 2 show related works about task parallelism and data locality optimization for the NUMA architecture using OpenMP. In Sect. 3, we propose a new clause for the task construct, which gives a hint about how to schedule tasks on the NUMA architecture. Our prototype implementation based on GNU Compiler Collection (GCC) is explained in the section. In Sect. 4, we introduce the new clause into KASTORS benchmark kernels to improve data locality of tasks. In Sect. 5, the benchmark kernels are evaluated using GCC and our implementation to show how much performance improvement can be achieved by our approach. Finally, we discuss the future work and conclude the paper and in Sect. 6.

2 Related Work

Barcelona OpenMP Task Suite (BOTS) [1,3] consists of several benchmark kernels exploiting tasks in OpenMP 3.0. The KASTORS benchmark suite (KASTORS) [4] developed by Inria is inspired by BOTS. The major difference between BOTS and KASTORS is that KASTORS utilize the task depend clause in OpenMP 4.0 to exploit dependency between tasks. Virouleau et al. [10] showed that fine-grain task dependencies can replace global synchronization of all tasks in a parallel region and improve the scalability of task parallelism in OpenMP.

The NUMA-aware task scheduler has been studied extensively [2,6–9]. Most of them focus on work-stealing algorithms in runtime to handle recursive algorithms. Muddukrishna et al. [5] showed that manual data distribution among NUMA nodes and their NUMA-aware task scheduling algorithm in runtime can

improve the parallel performance. This approach is similar to ours since our approach also requires explicit data distribution. However, task allocation is done explicitly using the extended OpenMP task construct in our approach.

3 OpenMP Extension for NUMA-Aware Task Allocation

The NUMA architecture, as its name suggests, provides non-uniform memory performance, which depends on the distance between a memory location and a core. Generally, improving data locality and reducing remote memory access can exploit potential memory performance on the NUMA architecture. The same is true for task parallelism in OpenMP. A task should be executed on the NUMA node where its processing data is allocated to get the highest memory bandwidth. In this section, we propose a new clause named node_bind for the OpenMP task construct. It specifies a NUMA node that the target task should be scheduled.

3.1 Overview

Figure 1 shows the conceptual model of our approach. The software system consists of multiple task queues connected to each NUMA node respectively. Assume that an application generates a number of OpenMP task which carries out computations on a single element of array A. The figure shows how tasks and data can be allocated on NUMA nodes and matched with the help of information given by the programmer.

First, the programmer distributes the array among NUMA nodes by using existing NUMA libraries such as libnuma. Then the programmer describes OpenMP tasks with a hint about which element would be accessed in the task.

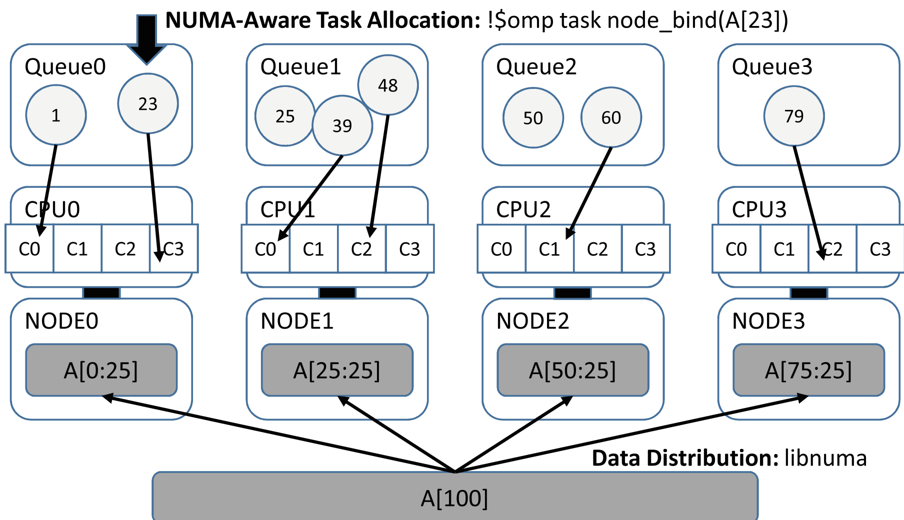

Fig. 1. NUMA-aware data distribution and task allocation

The node_bind clause used in the task construct, which we propose in this paper, gives the information to the OpenMP compiler and runtime. The OpenMP compiler can determine the node id that the specified element is allocated. The OpenMP runtime utilizes it to schedule the task to the corresponding task queue. In our implementation, a group of cores connected to the same NUMA node has a higher priority to access the corresponding task queue than others so that the cores would have more chance to access the local memory. This approach provides an explicit way of improving data locality in tasks by combining explicit data distribution.

3.2 Language Definition

Listing 1.1 shows the definition of the node_bind clause. node_bind is defined as an additional clause to the task construct. It takes one variable reference that its address can be determined by the compiler. The compiler assigns the target task on the same node that the specified variable is allocated. When multiple node_bind is given, the compiler uses the last node_bind clause.

Listing 1.1. node_bind clause definition

```
#pragma omp task [clause[[,] clause] ...] new_line
   structured_block
 clause := untied
         | depend(dependence_type : list)
         | . . .
         | node_bind(variable)
```

Listing 1.2 shows an example code of the node_bind clause. The code is taken from the Strassen kernel in KASTORS. The output array C is given in the node_bind clause. When M2 and C are allocated on the same NUMA node, the task can be executed without any remote memory access. In some cases, the depend clause has enough information to specify the NUMA node to be allocated, instead of using the node_bind clause. Using output dependency for task allocation may be a good idea because usually there is one output dependency for each task, and the output array can be easily distributed among NUMA nodes compare to input arrays. However, we propose to use more explicit way of using the node_bind clause in this paper because task allocation is to be controlled explicitly and we are interested in seeing how performance changes by the initialization scheme.

Listing 1.2. node_bind clause example code

```
#pragma omp task depend(inout: C) depend(in: M2) \
                 private(Row, Column) node_bind(C[0])
 for (Row = 0; Row < QuadrantSize; Row++)
   for (Column = 0; Column < QuadrantSize; Column += 1)
     C[RowWidthC*Row+Column] += M2[Row*QuadrantSize+Column];
```

3.3 Prototype Implementation Using GCC

We have implemented the node_bind clause modifying GNU Compiler Collection (GCC) version 5.3.0. GCC 5.3.0 supports OpenMP 4.0 features including the task depend clause. The GCC implementation determines the address of the variable specified in the depend clause and passed it to the runtime. We used the mechanism to implement the node_bind clause. The compiler determines the address of the variable specified in the node_bind clause and add it to the argument list of the GOMP_task() function which generates OpenMP tasks.

When the address is passed to the OpenMP runtime system, our implementation calls a Linux system call, get_mempolicy() to determine the NUMA node id on which the specified variable is allocated. The node id is used to select the corresponding task queue. The GCC implementation creates a single global task queue shared by all tasks in a parallel region. Our implementation also has a global task queue and creates multiple task queues assigned to each NUMA node respectively. If get_mempolicy() returns an available NUMA id, the corresponding task queue is selected. If the function could not determine the NUMA node (it usually happens when the memory area has been allocated, but not touched by any thread), the global task queue is selected. Tasks without a node_bind clause are scheduled to the global task queue.

Task handling functions in GCC dequeue a task from the global task queue and execute it. Our implementation is modified to use NUMA task queues. As long as tasks exist in the local queue, cores dequeue tasks from the local task queue. If there is no task left in the local task queue, tasks in the global queue are scheduled. If there is no task both in the local queue and the global queue, cores take tasks from other NUMA nodes. This improves workload balance between cores at the cost of remote page access.

4 KASTOR Kernel Optimization with node_bind

In this section, we introduce the node_bind clause into the KASTOR benchmark kernels, Jacobi, SparseLU, and Strassen. Each kernel is implemented in two ways, TASK and TASK DEP. The TASK version is implemented using independent tasks in OpenMP 3.0, which is equivalent to BOTS. Depend clauses in OpenMP 4.0 are added in the TASK DEP version to replace global synchronization. We have modified both versions adding node_bind clauses. Explicit data distribution schemes using NUMA APIs have been tested on each kernel.

4.1 Jacobi Kernel

The Jacobi kernel solves a 2D Poisson equation on evenly-divided N $\times$ N grid points. Along with the TASK and TASK DEP version, KASTOR implements the FOR version for Jacobi. It uses the parallel for construct to perform loop work sharing, which is a straightforward way to parallelize stencil computation. The Jacobi kernel performs 5-point 2D stencil computation known to be memory-intensive.

Listing 1.3 shows the TASK DEP version of Jacobi. Each task calculates assigned grid points and stores the result data in the output array. The node_bind clauses are added to specify the first element of the assigned output array block in each task. Since each grid point can be calculated independently, we can distribute the calculation evenly not only among cores but also NUMA nodes. The parallel for construct was used to initialize array data so that the arrays are evenly distributed among NUMA nodes. It relies on the first-touch memory allocation policy of the Linux OS. The TASK version was also modified in the same way.

Listing 1.3. Jacobi TASK DEP Kernel with node_bind clauses

```
for (it = itold + 1; it <= itnew; it++) {
  for (block_x = 0; block_x < max_blocks_x; block_x++)
    for (block_y = 0; block_y < max_blocks_y; block_y++)
#pragma omp task shared(u_,unew_,block_size,nx,ny) \
        depend(in: unew[...]) depend(out: u[...]) ... \
        node_bind(u_[block_x*block_size*nx+block_y*block_size])
      copy_block(nx,ny,block_x,block_y,u_,unew_,block_size);

  for (block_x = 0; block_x < max_blocks_x; block_x++)
    for (block_y = 0; block_y < max_blocks_y; block_y++) ...
#pragma omp task shared(u_,unew_,f_,dx,dy,nx,ny,block_size) \
        depend(out: unew[...]) depend(in: f[...], ...) ... \
        node_bind(unew_[block_x*block_size*nx+...])
      compute_estimate(block_x,block_y,u_,unew_,f_,dx,dy,
                       nx,ny,block_size);}
```

4.2 SparseLU Kernel

The SparseLU kernel calculates the LU decomposition of a sparse matrix. Listing 1.4 shows the TASK version of SparseLU. BENCH is a 2D array of which each element is the memory pointer to the submatrix. SparseLU allocates a submatrix to the locations where the problem matrix has non-zero values. The LU decomposition is carried out to the non-NULL submatrices.

Listing 1.4. SpaseLU TASK Kernel with node_bind clauses

```
lu0(BENCH[kk*matrix_size+kk], submatrix_size);
for (jj=kk+1; jj<matrix_size; jj++)
  if (BENCH[kk*matrix_size+jj] != NULL)
#pragma omp task untied firstprivate(kk, jj) shared(BENCH) \
                 node_bind(BENCH[kk*matrix_size+jj][0])
     fwd(BENCH[kk*matrix_size+kk],
         BENCH[kk*matrix_size+jj],submatrix_size);
for (ii=kk+1; ii<matrix_size; ii++)
  if (BENCH[ii*matrix_size+kk] != NULL)
#pragma omp task untied firstprivate(kk, ii) shared(BENCH) \
                 node_bind(BENCH[ii*matrix_size+kk][0])
     bdiv (BENCH[kk*matrix_size+kk],
           BENCH[ii*matrix_size+kk],submatrix_size);
#pragma omp taskwait
```

```
for (ii=kk+1; ii<matrix_size; ii++)
  if (BENCH[ii*matrix_size+kk] != NULL)
    for (jj=kk+1; jj<matrix_size; jj++)
      if (BENCH[kk*matrix_size+jj] != NULL) {
        if (BENCH[ii*matrix_size+jj]==NULL)
          BENCH[ii*matrix_size+jj] =
            allocate_clean_block_numa(submatrix_size, jj);
#pragma omp task untied firstprivate(kk, jj, ii) shared(BENCH) \
                node_bind(BENCH[ii*matrix_size+jj][0])
        bmod(BENCH[ii*matrix_size+kk],BENCH[kk*matrix_size+jj],
             BENCH[ii*matrix_size+jj], submatrix_size);
      }
#pragma omp taskwait
```

The submatrix allocation pattern is irregular in SparseLU because it depends on the sparsity of the input matrix. Figure 2 shows the allocation pattern of the default input used in SparseLU. Each square indicates a submatrix. The submatrices allocated in the initialization routine are drawn as black squares. Submatrices allocated during the LU decomposition are drawn as gray blocks. White squares are zero matrices which are not accessed in the LU decomposition. As the figure shows, every column at even indices has non-zero elements. Given the situation, we distributed each column in a block-cyclic manner. 4 columns are grouped into a block, and blocks are distributed among NUMA nodes in a round-robin fashion. The columns are distributed among 4 NUMA nodes in Fig. 2.

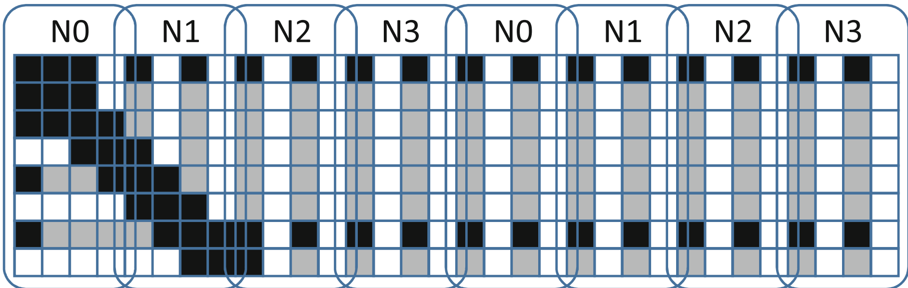

Fig. 2. Data distribution in SparseLU

The data distribution is implemented by using libnuma. numa_alloc_onnode() takes a NUMA node id as an argument and allocates a memory chunk on the specified node. Given the column index *jj*, the initialization routine allocates the submatrix on node ((*jj* /B) % N) where B is the block size and N is the number of nodes to be used. Each task is scheduled to the node that the output submatrix is allocated. As Listing 1.4 shows, the first element of the output submatrix is given in the numa_node clauses so that no remote page access will occur when accessing the output. Note that the original version calls the allocation function

in a task region. The modified version calls the allocation function in the single region with a target node id. The difference between these allocation schemes is explained in Sect. 5.

4.3 Strassen Kernel

The Strassen kernel calculates the multiplication of dense matrices using the Strassen algorithm. The algorithm reduces the number of multiplication operations by splitting each matrix into 4 equally divided submatrices. Figure 3 shows how the output matrix is divided in recursive function calls. The output array C is split into 4 submatrices (C0-C3 in Fig. 3) in the first matrix multiplication function call. Each submatrix is calculated in parallel using independent tasks. Each task splits the submatrix into 4 smaller submatrices and generates tasks to handle them. This recursive computation guarantees that the child tasks always compute the output elements which are allocated in the parent task, as we can see in Fig. 3.

Since the output matrix is split into 4 submatrices, the array elements can be distributed among 4 nodes at most. We distributed the array elements explicitly by using the OpenMP parallel construct and libnuma APIs. First, aligned_alloc() is used to allocate the output array with a page boundary alignment. The starting index of the corresponding submatrix is calculated in a parallel region. A thread is selected for each NUMA node in the parallel region. Then the thread calls numa_setlocal_memory() to migrate memory pages to the local NUMA node. As a result, the submatrices C0-C3 shown in Fig. 3 are allocated on multiple NUMA nodes.

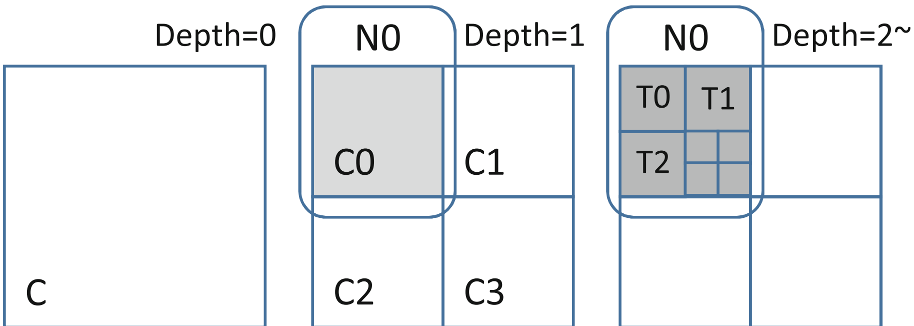

Fig. 3. Data distribution in Strassen

Listing 1.5 shows the TASK DEP version of Strassen. The kernel performs 7 multiplications and 4 of them access the output array C. node_bind clauses are specified for them. Since we wanted to use the node_bind clause at the top level of recursive calls, our modified GCC runtime performs task allocation only if the task does not have the parent task. When the parent task exists, the NUMA node id assigned to the parent task will be used.

Listing 1.5. Strassen TASK DEP Kernel Code with node_bind clauses

```
void OptimizedStrassenMultiply_par(double *C, double *A, ...
#pragma omp task depend(in: A, B) depend(out: M2)
  OptimizedStrassenMultiply_par(M2, A, B, ...);
#pragma omp task untied depend(in: S1, S5) depend(out: M5)
  OptimizedStrassenMultiply_par(M5, S1, S5, ...);
#pragma omp task untied depend(in: S2, S6) depend(out: T1sMULT)
  OptimizedStrassenMultiply_par(T1sMULT, S2, S6, ...);
#pragma omp task untied depend(in: S3, S7) depend(out: C22) \
               node_bind(C22[0])
  OptimizedStrassenMultiply_par(C22, S3, S7, ...);
#pragma omp task untied depend(in: A12, B21) depend(out: C) \
               node_bind(C[0])
  OptimizedStrassenMultiply_par(C, A12, B21, ...);
#pragma omp task untied depend(in: S4, B22) depend(out: C12) \
               node_bind(C12[0])
  OptimizedStrassenMultiply_par(C12, S4, B22, ...);
#pragma omp task untied depend(in: A22, S8) depend(out: C21) \
               node_bind(C21[0])
  OptimizedStrassenMultiply_par(C21, A22, S8, ...);
#pragma omp task depend(inout: C) depend(in: M2) ...
  for (Row = 0; Row < QuadrantSize; Row++)
    for (Column = 0; Column < QuadrantSize; Column += 1)
      C[RowWidthC*Row+Column] += M2[Row*QuadrantSize+Column];
```

5 Performance Evaluation

In this section, we measured the performance of the KASTORS benchmark kernels using GCC and our implementation. Table 1 shows the hardware configuration and the memory performance used for the evaluation. Each CPU has 18 physical cores and 2 NUMA nodes (when the COD mode is enabled). The OpenMP version of the Stream Triad benchmark is used to measure the sustainable memory bandwidth. OMP_PROC_BIND is set to CLOSE so that OpenMP threads use the smallest number of NUMA nodes. We used the same value to evaluate KASTORS. We compiled the original KASTOR kernels using GCC, and the modified kernels shown in Sect. 4 using our implementation.

5.1 Result of Jacobi Kernel

Figure 4 shows the performance speedup of the Jacobi kernel against the serial version. The matrix size is 16384×16384 and the block size is 1024. The original FOR, TASK, and TASK DEP version (*task init* and original in Fig. 4) initializes the grid point values in parallel execution of independent tasks so that the memory pages are distributed among NUMA nodes in a random manner. On the other hand, the modified TASK and TASK DEP version showed in Sect. 4 (*node_bind* in Fig. 4) initializes the grid points using loop work sharing in a parallel region. As a result, the memory pages are evenly distributed among nodes.

Table 1. Evalustion environment

Item	Name/Value	
CPU	Intel (R) Xeon (R) CPU E5-2699 v3, 2 sockets	
	18 cores with HT, 2.30 GHz, COD enabled	
Memory	DDR4 128 GB	
Stream performance (Triad, GB/s)	1 thread: 14.49	
	9 threads (1 node): 21.82	18 threads (2 nodes): 43.36
	27 threads (3 nodes): 65.01	36 threads (4 nodes): 86.49
Memory	DDR4 128 GB	
OS	Red Hat Enterprise Linux Server release 7.1	
	Linux Kernel: 3.10.0-229.7.2.el7.x86_64	
Compiler	GNU Compiler gcc version 5.3.0	

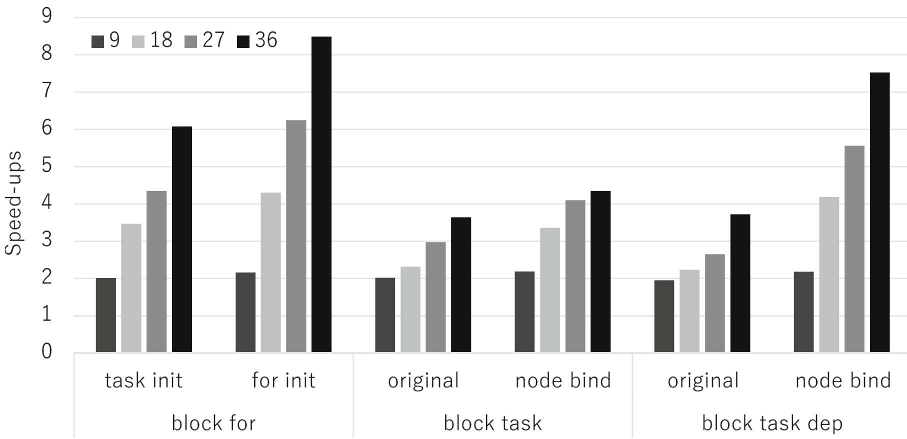

Fig. 4. Parallel performance of Jacobi

The performance of the FOR version shows that the initialization scheme can change the performance. Compared with *for init*, the FOR version initialized using the parallel for construct, *task init* achieves the lower performance than *for init* because the memory pages are allocated randomly among NUMA nodes. *for init* achieves the best performance because the access pattern of the initialization and the computation is perfectly matched.

The modified TASK and TASK DEP version achieve better performance than the original versions for the same reason of *for init*. It reduces remote page access by matching the data allocation pattern and the task scheduling pattern. The reason why the TASK version show the lower scalability than the TASK DEP version is that there is a global synchronization (taskwait construct) between the update phase and the computation phase. The GCC OpenMP runtime eagerly uses the master thread to execute the child tasks to handle the

taskwait construct. The same thing happens in our implementation so that the specified task allocation scheme is ignored in the global synchronization.

5.2 Result of SparseLU Kernel

Figure 5 shows the performance speedup of the SparseLU kernel against the serial version. The matrix size is 128 and the submatrix size is 64. The original TASK version allocates submatrices in each task so that the memory pages are distributed in a random manner. The original TASK DEP version allocates submatrices on the master thread before task creation in order to specify task dependency using the submatrix indices. In both cases, remote page access occurs when accessing the output submatrix since the GCC OpenMP runtime does not consider data locality in task scheduling.

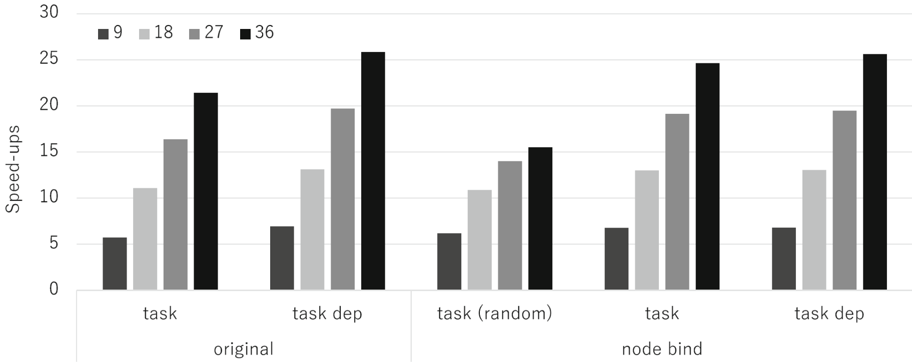

Fig. 5. Parallel performance of SparseLU

node bind in Fig. 5 shows the performance of the modified TASK and TASK DEP version shown in Sect. 4. *task (random)* uses node_bind clauses and submatrices are allocated in the same manner with the original TASK version. *task (random)* achieves lower performance than the original version because of the irregular allocation pattern. All submatrices are allocated in the first iteration of the LU decomposition (when *kk* is 0 in Listing 1.4) and reused in the subsequent iterations. *task (random)* uses the irregular allocation pattern to allocate tasks in every iteration. It causes load imbalance between NUMA nodes.

The modified TASK version solves the problem. It distributes submatrices evenly among nodes using NUMA APIs. The block-cyclic manner used in Fig. 2 guarantees that each NUMA node has the balanced workload in every iteration. As a result, the modified version achieves better performance than the original version. The result shows that the performance of the node_bind clause is sensitive to data distribution.

5.3 Result of Strassen Kernel

Figure 6 shows the performance speedup of the Strassen kernel against the serial version. The modified TASK and TASK DEP version distributes the output matrix C among nodes. The output matrix and temporary arrays allocated in the parent task are used in the child tasks in recursive function calls. Our implementation schedules child tasks to the same node used to the parent task. The explicit data distribution and the task scheduling scheme increase the performance of the TASK DEP version by 7 %.

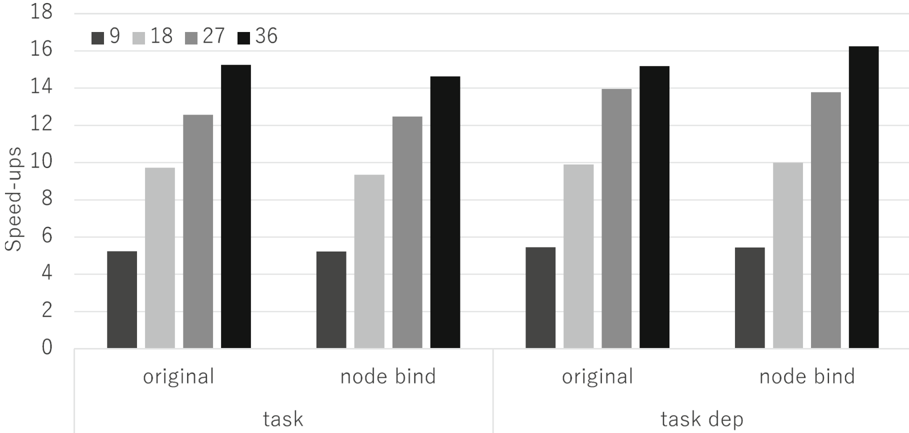

Fig. 6. Parallel performance of Strassen

While the output matrix is localized, the input matrices should be copied from remote NUMA nodes due to the data dependency coming from the Strassen algorithm. For the further improvement, we are testing some techniques to reduce the remote page access, such as duplicating input data among nodes.

6 Conclusion

In this paper, we proposed the node_bind clause for the OpenMP task construct specifying the NUMA node id that the task should be scheduled. The extension can be combined with explicit data distribution to reduce remote page access, as shown in Sect. 4. Although it requires additional programming effort, the results of the performance evaluation using the KASTOR benchmark showed that NUMA-aware task allocation improved the parallel performance. The Jacobi kernel using our approach shows 3.6 times better performance than GCC when using 36 threads on a 36-core, 4-NUMA node machine. Techniques for distributing data and reducing communication have been studied extensively in cluster computing. We found that those techniques can be also helpful for the NUMA architecture. Currently, we are designing an OpenMP extension to describe data distribution instead of using Linux OS system calls and NUMA APIs.

References

1. Barcelona OpenMP Task Suite (BOTS). https://pm.bsc.es/projects/bots/
2. Drebes, A., Heydemann, K., Drach, N., Pop, A., Cohen, A.: Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. ACM Trans. Archit. Code Optim. **11**(3), 30:1–30:25 (2014). http://doi.acm.org/10.1145/2641764
3. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proceedings of the 2009 International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE Computer Society, Washington, DC (2009). doi:10.1109/ICPP.2009.64
4. KASTORS Benchmark. https://gforge.inria.fr/projects/kastors/
5. Muddukrishna, A., Jonsson, P.A., Vlassov, V., Brorsson, M.: Locality-aware task scheduling and data distribution on NUMA systems. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 156–170. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40698-0_12
6. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. Int. J. High Perform. Comput. Appl. **26**(2), 110–124 (2012). doi:10.1177/1094342011434065
7. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings ofthe International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12. IEEE Computer Society Press, Los Alamitos (2012). http://dl.acm.org/citation.cfm?id=2388996.2389085
8. Tahan, O.: Towards efficient OpenMP strategies for non-uniform architectures. CoRR abs/1411.7131 (2014). http://arxiv.org/abs/1411.7131
9. Vikranth, B., Wankar, R., Rao, C.R.: Topology aware task stealing for on-chip NUMA multi-core processors. Procedia Comput. Sci. **18**, 379–388 (2013). 2013 International Conference on Computational Science. http://www.sciencedirect.com/science/article/pii/S187705091300344X
10. Virouleau, P., Brunet, P., Broquedis, F., Furmento, N., Thibault, S., Aumage, O., Gautier, T.: Evaluation of OpenMP dependent tasks with the KASTORS benchmark suite. In: DeRose, L., de Supinski, B.R., Olivier, S.L., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2014. LNCS, vol. 8766, pp. 16–29. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11454-5_2