Efficient Work Stealing for Portability of Nested Parallelism and Composability of Multithreaded Program

Adnan
Electrical Engineering Dept
Universitas Hasanuddin, Indonesia
adnan@unhas.ac.id

Zahir Zainuddin
Electrical Engineering Dept
Universitas Hasanuddin, Indonesia

Wardi
Electrical Engineering Dept
Universitas Hasanuddin, Indonesia

Abstract

We present performance evaluations of parallel-for loop with work stealing technique. The parallel-for by work stealing transforms the parallel-loop into a form of binary tree by making use of method of divide-and-conquer. Iterations are distributed in the leaves procedures of the binary tree, and the parallel executions are performed by stealing subtrees from the bottom of the tree. The work stealing and divide-and-conquer are used to address the portability problem in nested parallelism and composability. By work stealing and divide-and-conquer techniques, fine-grained parallel-for can be implemented without contributing large work overhead. Low work overhead is important as the number of processor could be less than expected. Low overhead and fine-grained of work stealing scheduler makes highly parallel processor cores are able to scale the performance. In addition, the approach used in this work makes efficient nested parallelism is possible. Because of a low overhead, we show that the work stealing and divide-and-conquer deliver good scalability in nested parallel Sparse LU factorization.

Keywords Composability, nested-parallelism, work-stealing, multicore

1. Introduction

As many-core processors are available recently, composability is an increasingly important feature of programming languages. Composability refers to the ability to develop a parallel program from its components. The components themselves can be executed by a single thread or multiple threads. Whenever threads make a call to modules where parallelism exists, they may launch new threads as children for that modules. Composability allows modular parallel programming. This means that a programmer will not need to have low-level knowledge about the parallelism that exists within a subroutine. Instead, the programmer simply inserts calls to functions and lets the runtime system schedule the parallelism within the function dynamically. Therefore, composability brings as a consequence the need for nested parallelism. In a program with nested parallelism, there is a possibility that a program will lack parallelism at the outer loop while parallelism is ample in the inner loop. In such cases, threads must be scheduled efficiently for the sake of the parallelism that exists in the inner loop. Also, it is sometime optimized parallelism for specific number of processor cores is not portable. Therefore, nested parallelism must be performed automatically at runtime. Without efficient implementation of thread scheduling, the nested parallelism that constitutes composability contributes to a large overhead. For instance, an OpenMP thread must launch a set of new threads, which contributes to overhead, whenever it encounters a nested work-sharing construct. In meanwhile, load imbalances may sometime cause performance degradation because thread assignment for nested parallelism is static. Therefore, it is a strong requirement for the programmer to make a careful assignment of a number of threads in such situations.

Low overhead and flexibility of work stealing technique are the key factors which support the nested parallelism. Work stealing is an efficient multithreaded programming technique. Parallel execution based on work stealing contributes to small overhead. The overhead is as small as the cost of function call, and threads scheduling of work stealing is flexible. Threads can be scheduled for the parallelism which may exist in anyplace of parallel program. Therefore, we are motivated to make use of work stealing as the base of this work. As the instance of work stealing, StackThreads/MP[12] was used to implement efficient parallel-for which capable of efficient nested parallelism.

This paper concerns the portability problem of nested parallelism and composability. In this paper, a form of parallel loop was implemented. The numbers of parallel threads assigned to the loop do not need to be specified. Instead of the numbers of threads, a number of maximum sub iterations assigned for each thread that may run in parallel is specified. A reducer object is featured in the parallel loop. The reducer object can be applied to any data type and size.

The contribution of this paper are as follows :

- We demonstrated that parallel-for which is implemented using divide-and-conquer and work stealing is a solution for the portability of nested parallelism and composability
- We evaluated the performance of nested parallel Sparse LU factorization using parallel-for.

The remainder of this paper is organized as follows. We discuss some related works of this paper on section 2. We discuss the portability of nested parallelism and composability problems in section 3. We discuss work stealing based execution in section 4. In section 5, we discuss performance evaluation and its result in nested parallel Sparse LU factorization. We conclude this work on section 7.

2. Related Works

Many work stealing[1] techniques work based on lazy task creation, including Cilk[2, 3] and StackThreads/MP[6, 11, 12]. Cilk and StackThreads/MP create logical threads and allow the scheduler to schedule bottommost threads so that any idle worker can steal them. Unlike Cilk and StackThreads/MP, Tascell[5] is logical-thread-free technique that also provides a work stealing capability. However, Tascell only allows work stealing after receiving a steal request while performing polling, which is similar to StackThreads/MP. We implemented parallel-for that built on top the StackThreads/MP.

Tanaka[10] implemented nested parallelism for OpenMP via StackThreads/MP for Omni OpenMP[7]. This work also described implementation of a parallel team of threads as logical threads created by workers using divide-and-conquer recursion. Omni OpenMP specifies the number of logical threads that a parent thread creates but our parallel loop implementation specifies a lower-bound and upper-bound of a maximum number of iterations for loop control variable for each logical thread. We do not specify the exact number of threads a parent thread creates. Therefore, a programmer would be allowed to implement a parallel-for that resembles the serial form of the loop-for.

Another loop parallelization based on work stealing is the Window algorithm[13]. The Window algorithm differs from divide-and-conquer recursion in that divide-and-conquer divides sequences of n elements into n/m chunks of elements that are adjacent.

Parallel-for is one natural form of regular parallelism, and the OpenMP [8] is one popular standard that features a parallel-for definition. OpenMP defines that implementations have a scheduler that schedule parallel team of threads to the iteration space. Cilk++ implemented cilk_for by means of divide and conquer recursion to provide more parallel slackness and better load balancing.

Cilk++ implemented reducers as an C++ hyperobject[4] independently from cilk++ runtime system. As a hyperobject Cilk++ reducer work flexible in both regular and irregular parallelism. We currently implemented reducer as a c structure. We implemented reducer so that it tightly coupled to divide-and-conquer recursion. Together with parallel_for,Intel Threaded Building Block[9] also provided a parallel_reducer.

3. Portability of Nested Parallelism and Composability Problem

In this section, we discuss parallel-for and compare two different techniques which are used to implement. The technique are work sharing and work stealing.

3.1 Portability of Nested Parallelism

One common form of parallel construct in many parallel languages is a parallel form of loop-for. For example, one expresses parallel-loop using a pragma of `omp for` in OpenMP. Parallel loop-for of OpenMP is one of construct of OpenMP work sharing. In work sharing, there is a parallel region of threads which are created using a pragma `omp parallel`. A programmer may use then the pragma `omp for` within a parallel region or use the pragma `omp parallel for`. The latter is a combination the both of the pragmas. Runtime systems assigned parallel team of threads separately to the different chunks of iterations. The main task of programmers is to statically decide a number of assigned threads by either `num_threads()` clause or `OMP_NUM_THREADS`.

For the nested parallel-for of OpenMP, each parallel loop must be written within a parallel region. Therefore, the degree of parallelism on both outer and inner parallel regions are determined by either the OMP_NUM_THREADS environment variable or the `num_threads()` clause. Code in Fig. 1 shows an example of nested parallel loop using `omp parallel for` pragma. In Fig. 1, a programmer must specify the degree of parallelism at both outer loop and inner loop appropriately, so that the optimal performance can be achieved. However, because the number of threads is statically assigned, portable performance is difficult on that different number of processors of varying machines.

```
1. #pragma omp parallel for num_threads(12)
2. for(ii=0;ii<Y;ii++)
3. {
5.  #pragma omp parallel for num_threads(2)
6.  for(jj=ii+1;jj<Y;jj++)
7.   { //body_of_inner_loop; }
8. }
```

Figure 1. Nested parallel loop of OpenMP work sharing

```
par_for(i=0;i<XSIZE;i++)
   foo(i);

void foo(int i)
{
   int j;
   par_for(j=i+1;j<XSIZE;j++)
     anotherfoo();
}
```

Figure 2. Composability problem

3.2 Portability of Composable Parallel Program

By composability, we refers to the ability to develop a parallel program from its components. The components are that parallelized of procedures. To support parallel programs which are composable, parallelism should be expressed dynamically at runtime. The portability problem occurs in parallel programs which composable of parallel libraries procedures. The program makes calls to a procedure. The procedure has the parallelism that is difficult to be predicted. In addition, granularity and load imbalance problems add complexities while determining the degree of parallelism in those procedures of libraries. In OpenMP work sharing, if parallel library is highly parallel and the number of available processors is small, parallel overhead increases the execution time. In contrast, if a procedure of parallel library is lack of parallelism and the number of available processors is large, assigning many parallelism for that procedure may result in load imbalance and low processors utilization. We describe these requirements using the simple example in Fig. 2.

The number of processor cores now tends to be vary from small to large number, the parallelized routines are the binary code, and the degree of parallelism of them is difficult to be determined. Therefore, it is impossible to tune the routines that suit for that different number of processors of machines. In addition, it is possible that while other processors are busy, the remaining processor is not available. Only the processor that is executing current thread that available for the routine `foo()`. If this is the case, that current processor must execute `foo()` at the expense of serial procedure call cost. Only the lazy task creation technique is known capable of such small work overhead.

3.3 Nested Parallel-for and Work Stealing Strategy

This work is different from the OpenMP parallel-for in that it allows implementation of parallel-for in ways that does not require to specify the number of parallel threads that will be assigned to the parallel iterations. The benefit of leaving the number of threads

```
for(i=lower_bound;i<upper_bound;i++)
   body;

par_for(i=lower_bound;i<upper_bound;i+=step)
{
  for(ii=i->lower;ii<i->upper;ii++)
     body;
}
```

Figure 3. Parallel loop blocking does not parallelize all iteration

```
 1. par_for(i=0;i<Y;i+=nb1)
 2. {
 3.   int x;
 4.   par_for(k=0;k<X;k+=nb2)
 5.   {
 6.      for(ii=i;ii<i+=nb1;ii++)
 7.      for(kk=k;kk<k+nb2;kk++)
 8.         do_something(x);
 9.   }
10. }
```

Figure 4. A simple example of nested Parallel-for

unspecified is seen in situations where nested parallelism with load imbalance problems occurs or the parallelism can not be predicted.

A parallel-for is implemented by transforming the form of the loop to the form of a binary tree. This transformation makes use of divide-and-conquer recursion by distributing iterations on the leaves of the binary tree. Different workers execute different iterations by stealing a subtree from the root of the binary tree. This method creates a binary tree with a depth of $log_2(n+1)$. Therefore, it contributes to large overhead if parallelizing all iterations of large iteration space is necessary. However, we have the option of not parallelizing all iterations. Instead of parallelizing the first loop, we prefer to parallelize the second loop of Fig. 3.

In the latter form, the programmer can parallelize the outer loop and leave the inner loop serial. Each parallel thread works on a block of iterations instead of individual iteration. Doing so makes the granularity of the threads larger and result in an efficiency improvement. Further optimization then can be performed to that sequential of inner-loops. Optimization is possible to be performed by make use of loop optimizing technique such as loop unrolling.

In the second loop of Fig. 3, it only transforms the outer loop to the form of a binary tree. No evidence has yet been obtained to indicate whether benefits can be obtained by parallelizing the above inner loop. Perhaps, if threads work on outer iterations that are still large and programmers require the threads to work on smaller iterations so that they fit to the cache line, we may obtain benefits from parallelizing the inner loop as a nested parallel loop.

Let us define `par_for` as a parallel form of a loop. Let us now consider that two loops i and k are applying loop blocking as shown in Fig. 4. Each iteration thread has a private copy of the upper and lower bounds of the loop control variable of the loop. In addition, a parent thread transmits its loop control variable to all of its children. Therefore, the loop control variable i is visible from the inner loop.

One crucial aspect that must be obvious in nested parallelism is the scope of data. In Fig. 4, threads that live at outer loop i have private views of variable x. However, threads that live at inner loop k have a shared view of the variable x. Modifications of this variable by a child thread at the inner context must be visible by the parent thread and other children. Visibility of a variable from the outer parallel loop to the inner parallel loop is different from visibility of variable of the outer loop to the inner parallel loop. Visibility of variables makes use of the stack, while visibility of the loop

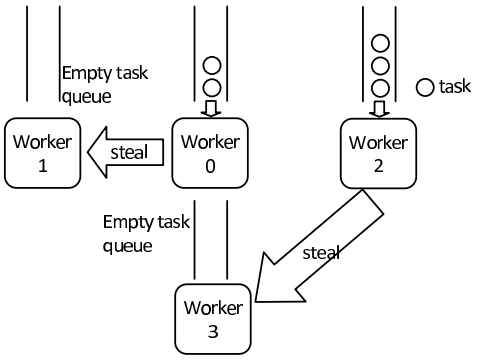

Figure 5. Work stealing strategy

control variable makes use of the thread local storage of the parent, which we restricted from the application in order to modify it. The variables `nb1` and `nb2` are passed by value. Nonetheless, modifying those variables from the application may break the sequential loop. Therefore, programmer must not modify those variables.

4. Work Stealing Based Execution

In this section, we introduce Lazy Task Creation Based Work Stealing. This efficient work stealing technique was used to implement parallel-for by divide-and-conquer algorithm.

4.1 Efficient Work Stealing for parallel-for

Work stealing refers to a scheduling mechanism for parallel tasks in which parallel task execution occurs because of task stealing. In work stealing computation, a program comprises a number of parallel tasks that are executed in parallel by different processors. In work stealing, a set of workers is grouped together. These workers are logical processor entities that execute threads. Workers that are busy create tasks and idle workers steal tasks from busy workers. A victim is defined as a busy worker from which tasks are being stolen, while the thief refers to an idle worker that steals tasks from victims.

In the work stealing mechanism, each worker maintains a task queue. While executing a task, the worker may create other tasks and place them in its task queue. Other workers may have empty task queues. Those workers with empty task queues steal tasks from busy workers. In Fig. 5 Workers 1 and 3 have empty task queues. As a result, Worker 1 steals a task from Worker 0 and Worker 3 steals a task from Worker 2.

4.2 StackThreads/MP Work Stealing Thread Library

StackThreads/MP is a thread library that implemented work stealing technique. In StackThreads/MP terminology, a worker refers to an OS thread, and a thread that is created by a worker is referred to as a fine-grained thread. The word thread is sometime used interchangeably with task, except when another task implementation is specified. By StackThreads/MP, a multithreaded program is allowed to be fine grain and capable of load balancing. Creating a new task or a thread is similar to a function call. A worker allocates and pushes a new frame on top of the parent tasks.

ST_THREAD_CREATE is one of APIs of StackThreads/MP. StackThreads/MP creates a new thread or task by making use of asynchronous function calls. Such asynchronous function calls are specified by the `ST_THREAD_CREATE` macro. Fig. 6 shows an example use of the macro. This example make use of parallel divide-and-conquer algorithm. In this example, a parent task inlines child task in line 16. It parent's continuation in line 18 available to

```
 1. void is_par_start(int a, int b, int nb,
 2.      Task *task, st_join_counter_t *jc)
 3. {
 4.  if(a+nb >= b)
 5.  {
 6.  /* execute Tasktask here
 7.    (*task->fn)(task->args);
 8.   */
 9.  st_join_counter_finish(jc);
10. }
11. else
12. {
13.  int m = (a+b)/2;
14.  st_join_counter_t cc[1];
15.  st_join_counter_init(cc,2);
16.  ST_THREAD_CREATE(is_par_start(a,m,nb,
17.                                 task,cc));
18.  is_par_start(m,b,nb,task,cc);
19.  st_join_counter_wait(cc);
20.  st_join_counter_finish(jc);
21.  }
22.}
```

Figure 6. An example use of ST_THREAD_CREATE in parallelizing divide-and-conquer algorithm

be stolen by any idle worker so that children tasks in line 16 and 18 can run in parallel by task stealing.

StackThreads/MP not only provides APIs to parallelize tasks, it also provides data structures and routines for locking and synchronization mechanisms. One such data structure `is st_join_counter_t`. This data structure has an integer element that should only be manipulated by certain routines such as `st_join_counter_init()` and `st_join_counter_ spawn()`. The data structure st_join_counter_t can be used to synchronize a parent thread and its children using a routine called `st_join_counter_wait()` and `st_join_co unter_finish()`.

5. Experiment

5.1 Nested Parallelism of Sparse LU Factorization

In this subsection, we discuss the different implementations of nested parallel Sparse LU factorization. The first implementation for nested parallel make use of OpenMP work sharing construct, and the second implementation for it make use of work stealing strategies. We make use of our parallel-for implementation, and we compare the performance with both performances of OpenMP work sharing and also Cilk work stealing.

5.1.1 Nested omp parallel implementation

In this subsection, we discuss the application of nested `omp parallel` of OpenMP work sharing in Sparse LU factorization. Sparse LU factorization of Barcelona OpenMP Task suite (BOTs) which was developed using `omp for`was modified. The Sparse LU of BOTs has four computational kernels. The kernels are `lu()`, `fwd()`, `(bdiv())` and `bmod()`. We modified the Sparse LU factorization benchmark software so that nested `omp parallel` and `omp for` were added to parallelize inner loop of `bmod()` computations. Fig. 7 shows the modification lines for the Sparse LU factorization of BOTs.

5.1.2 Nested Parallel implementation by work stealing

In this subsection, we discuss the application of nested parallelism and work stealing in Sparse LU factorization. The Sparse LU is adopted from Barcelona OpenMP Task benchmarks, BOTs. In the original OpenMP code, Sparse LU code was written in parallel-for

```
#pragma omp for schedule(dynamic)
  for (ii=kk+1; ii<bots_arg_size; ii++)
   if (BENCH[ii*bots_arg_size+kk] != NULL)
#pragma omp parallel private(jj) num_threads(2)
    {
     #pragma omp for schedule(dynamic)
     for (jj=kk+1; jj<bots_arg_size; jj++)
     {
```

Figure 7. Modification for Nested parallel Sparse LU from BOTs

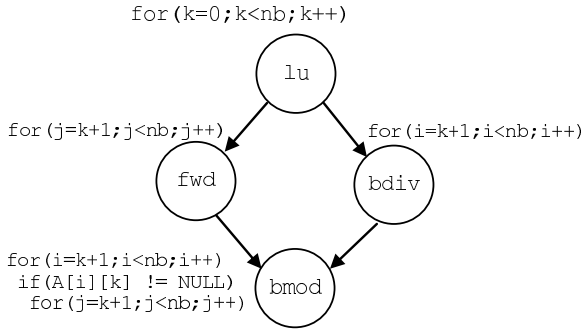

Figure 8. Task graph of Sparse LU factorization

a work-sharing construct. In a work-sharing construct, a parallel team of threads shares the work. All threads execute the same code with different data. In other words, it is a data parallel model. The scheduler assigns iteration chunks to parallel threads in a team. In an experiment, we assigned eight threads to the outer loops and three threads to the inner loops. In order to remedy the problem of load imbalance, implementation allows the threads to create tasks. However, those tasks contribute to overhead.

In contrast, Cilk-style work stealing (Cilk and StackThreads/MP) is not a data parallel model. Nonetheless, using divide-and-conquer recursion, Cilk can imitate data parallel model. Difficulties may arise whenever data-parallel by work-stealing needs to execute in a single region. In such cases, other threads that are waiting on the thread barrier must yield. This leads to idle threads and the steal overhead increases. Therefore, we took a parallel program application that we could not parallelize to the outer loop with the form of parallel-for by work stealing without the use of a thread barrier. Fig. 8 shows dependencies between tasks of Sparse LU factorization. There are difficulties with this program. The first one is dependency. Computational kernel of `lu` is executed within a loop sequentially. While one worker is computing the `lu()` the other workers are requesting works from busy workers. This is because lu execution occurs within loop k, time spent in idle is a multiple of nblocks. The same problem occurs with the cost from both divide-and-conquer recursion and steal overheads. Matrix sparseness contributes to overhead because of load imbalance. The work overheads of both recursions and steals are multiples of nblocks.

5.2 Experimental Configuration

Experiments were performed on a machine with two AMD Opteron 6168 CPUs. Each CPU comprises 12 cores with a 6 MB L3 shared cache. In each CPU, processor cores were equipped with a 64 KB L1 and 512 KB L2 caches. The machine was installed with 12 GB DDR 3 1066 MHz. The machine used Centos 5.3 Linux as its OS.

5.3 Performance Evaluation Results

5.3.1 Serial Program execution time

We presents serial program execution time of Sparse LU factorization in table 1. Serial program execution time should reflect that processing speed of one processor in actual work load of the Sparse LU factorization. Performance analyzes of parallel execution time can be provided after considering the serial execution time. Table 1 presents also work-overhead for different implementation of SparseLU factorization. With work-stealing strategy (StackThreads/MP & Cilk_for), inserting nested parallel-loop work-overhead could not be observed. But OpenMP work-sharing, inserting nested parallel-for incurs work-overhead. Therefore, work-stealing support composability better than the conventional OpenMP.

Table 1. Serial program execution time (sec) and work overhead of Sparse LU Factorization

GCC 4.4.3 (omp)	GCC 2.8.1 StackThreads/MP	Intel Parallel Studio Cilk_for
80.26	83.42	81.84
1.2%	0%	0%

5.3.2 Parallel Program Execution Time

In Table 2, the parallel execution times in this research are presented. We conducted two experiment for the Sparse LU with nested `omp parallel`. The first experiment make use of OMP_NUM_THREADS environment variable to set the number of threads assigned for both outer parallel region and inner parallel region. We refers the result of this experiment by mxm label in the table 2. If the environment variable is set to 2, each time a thread encounters `pragma omp parallel`, it creates a team of two parallel threads. Therefore, the total number of threads should be a power series, i.e. four threads if we set the environment variable to 2. The second experiment makes use of `num_threads()` clause to assign the number of threads for the outer parallel region and inner parallel region. Threads assignment using the clause is more flexible because different number can be assigned for outer parallel region and inner parallel region.

Lets compare the serial program execution times and their related sequential execution times which are including parallelism overhead. The sequential execution time of those using work stealing of parallel Sparse LU is slower than their related execution time of related serial program. These fact indicates that the work overhead of this technique is considerably small. It is equal to the cost of function call where this characteristic is favor the composability. In contrast, the work overhead of nested `omp parallel` implementation using GCC 4.4.3 is slightly large where this characteristic may increase execution time of the program on small size of multicore processors. Table 1 shows the fact of this.

Table 2. Parallel execution time of Sparse LU Factorization (sec)

# cores	(mxn) of omp parallel	(mxm) of omp parallel	is_par_for WS	cilk_for WS
1	81.27	81.39	83.16	81.59
2	40.9 (2x1)	40.9 (2x1)	41.5	40.93
4	20.75(4x1)	23.02(2x2)	20.6	20.552
8	12.98(4x2)	NA	10.7	10.525
16	6.91(8x2)	7.64(4x4)	5.6	5.489
24	5.1(12x2)	NA	3.9	3.841

The results in Tables 2 for Sparse LU factorization show evidence that when using nested `pragma omp parallel`, better performance can be achieved by specifying an appropriate number of threads in the inner parallel region than by leaving the number of threads unspecified. The situation is different when nested parallel-for by work stealing and the divide-and-conquer technique was used. In the latter case, the performance is better than with nested `omp parallel`, even though number of threads assignment is not specified. In this results were obtained without optimization switch. Therefore, we can improve the sequential execution of using -O3 of GCC 4.4.3 so that the OpenMP version of this benchmark run faster than cilk_for version. The same improve can also be observed in the case using of StackThreads/MP as if we compiled separately the computational kernel with -O3 switch of newer GCC. It seems that GCC 4.4.3 generated better sequential code of the computational kernel than that Intel Parallel Studio.

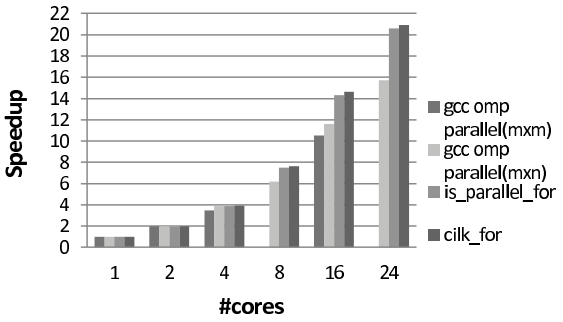

Figure 9. Speedup relatives to serial program using GCC 4.4.3

Although its serial program execution time is slower than the serial program using GCC 4.4.3, The parallel execution time of nested parallelism of this work, that make use of GCC 2.8.1, is faster than for nested omp parallel of OpenMP. The performance of cilk_for, that make use of Intel C Compiler, is comparable than the performance of this work. Fig. 9 shows the performance comparison of this work with other implementations such as nested `omp parallel` and nested `cilk_for`. Nonetheless, the performance of this work will be faster when newer back end GCC compiler is used to compile the `lu()`, `fwd()`, `bdiv()`, and `bmod()` computational kernels

6. Conclusion

Efficient nested parallelism implementation is required for efficiency since many-core processors will bring high levels of parallelism to application programs. By incorporating efficient nested parallelism within a program, many-core processors can be utilized efficiently because increased parallelism can be extracted. However, current parallel programming models such as OpenMP encounter portability difficulty when parallelism needs to be specified in the inner parallel region.

In this research, a work stealing strategy and a divide-and-conquer algorithm has been used for the implementation of parallel-for. With the work stealing strategy and divide-and-conquer algorithm, nested parallelism can be decided automatically at runtime. With this strategy, programs with nested parallelism can be made portable without difficulty. The parallel-for was evaluated using Sparse LU factorization that parallelism is nested. Performance in Sparse LU factorization using parallel-for and work stealing achieved a speedup of 20 using 24 processor cores.

References

[1] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999. ISSN 0004-5411.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN*, 30:207–216, August 1995. ISSN 0362-1340.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN*, 33:212–223, May 1998. ISSN 0362-1340.

[4] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: http://doi.acm.org/10.1145/1583991.1584017. URL `http://doi.acm.org/10.1145/1583991.1584017`.

[5] T. Hiraishi, M. Yasugi, S. Umatani, and T. Yuasa. Backtracking-based load balancing. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (14th PPOPP'09)*, pages 55–64, Raleigh, NC, USA, Feb. 2009. ACM.

[6] K. Taura, K. Tabata, A. Yonezawa. Integrating futures into calling standar. Technical report, University of Tokyo, 1999.

[7] K. Kusano, S. Satoh, and M. Sato. Performance evaluation of the omni OpenMP compiler. In *In Proceedings of International Workshop on OpenMP: Experiences and Implementations (WOMPEI), volume 1940 of LNCS*, pages 403–414, 2000.

[8] OpenMP ARB. OpenMP application program interface, v.3.0. Online, 2008.

[9] Reinders. *Intel Threading Building Blocks*. O'Reilly, July 2007.

[10] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of OpenMP applications with nested parallelism. In S. Dwarkadas, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers, (5th LCR'2000)*, volume 1915 of *Lecture Notes in Computer Science (LNCS)*, pages 100–112. Springer-Verlag (New York), Rochester, NY, USA, May 2000, Selected Paper.

[11] K. Taura. *StackThreads/MP User's Manual*. University Of Tokyo, http://venus.is.s.u-tokyo.ac.jp/sthreads/, 2010.

[12] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: Integrating futures into calling standards. *SIGPLAN*, 34:60–71, May 1999. ISSN 0362-1340.

[13] M. Tchiboukdjian, V. Danjean, T. Gautier, F. Le Mentec, and B. Raffin. A work stealing scheduler for parallel loops on shared cache multicores. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 99–107, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21877-4.