Abstract—This paper reviews some important issues for scalability in programming and future trend with many-core technology. According some experimental results of different parallel programs, such as fast Fourier transform and Unbalanced Tree Search and on twelve cores of a parallel computer, we identified two issues that should be concerned in programming the many-core processor, and . The issues are efficiency, load imbalance. Low efficiency of parallel program makes scalability of parallel program low. Although we could make program to be efficient by making its granularity coarse, load imbalance usually occurs. In addition, it sometimes not only the small task granularity will result in low efficiency, but it also the small load balancing granularity incurs high overhead. Therefore, efficient parallel programming paradigm is a mandatory for programming of many-core processor. For a high utilization of many-core processor, composability by work-stealing support can help programmer exploit efficiently this cutting-edge technology.

Index Terms—manycore; overhead; granularity-control; work-stealing;

I. INTRODUCTION

The component that plays the most important role in computer data processing is the processor. Processor consist of digital circuits (ICs) fabricated onto component called chips. Each such chip consist of a large number of transistors. According to Moore’s Law, the number of integrated transistors that can be placed on an IC doubles approximately every 18-24 months. This law is expected to remain valid in the future since intel announced 3D gate transistor in which the size can be made smaller. Processors have a large number of transistor integrated allow their architecture advanced. The using of pipeline and superscalar processors introduced parallelism in instruction level. However, since dependencies between instructions of a single thread obstruct parallel execution, adding more functional unit within a processor will not scale up performance significantly for the majority of programs. Therefore, it was decided that adding multiple thread controls to a chip is the best performance improvement choice, and simultaneous multi threading (SMT) processors were introduced to add multiple thread control. However, SMT technology has been unable to overcome the resource conflict problem such as competing shared floating point unit. Therefore, a new strategy was developed that called for integrating multiple processors into a single chip, resulting in a multicore and manycore processor.

Nowadays, multicore and manycore available for higher parallelism in thread level. Multicore and manycore processors can be classified as a type of shared memory multiprocessor. However, they are different from traditional multiprocessors, in which two or more discrete CPUs are connected. In traditional multiprocessor, each CPU has access to small on-chip cache memory, register and an execution unit and all processors share the data held in their shared memories. Since their introduction by CPU manufactures, multicore processors have rapidly increased in popularity and now most CPUs on the market contain multiple processing unit, which popularly referred to as cores. Their hierarchical design distinguishes multicore processors from standard shared multiprocessors. Not only does the multicore processor integrate two or more execution units on a single chip, a multicore processor also includes L1, L2 cache and a memory controller. Some manufacturers even integrate a large-size shared cache (known as L3) into their products of multicore processor. Researchers believe that large size of cache may result in some specific type of software achieve superlinear speedup[1].

Multicore and manycore processors are shared memory multiprocessors. Although distributed memory programming models are applicable for them, shared memory programming model is the most appropriate model for multicore and manycore processors. Therefore, in this paper we concern only shared memory programming model, and we urge to make use of the shared memory programming model for these new architectures.

The remainder of this paper is organized as follow, in section two, we discuss the relation between speedup and efficiency. In the same section, we elaborate some issues that affect efficiency of parallel program. In section three, we discuss efficient work-stealing based execution of parallel program, and its implementation that support both data and task par-
allelism. In section four, we discuss experimental results that show work overhead and critical path overhead. In section five, we discuss the future trend of parallel programming and we conclude this paper.

II. SPEEDUP AND EFFICIENCY OF PARALLEL PROGRAM

In this section, two basic performance metrics of parallel program’s are discussed. Both of them are speedup and efficiency. There is a strong relation between them.

A. Speedup

Speedup is a performance metric of parallel program. Given a parallel computer with $P$ processors, the ideal parallel execution time of a parallel program is $T_P = T_S / P$, where $T_S$ is serial execution time of the parallel program. In practice, one measure $T_S$ as the execution time of pure serial code. Therefore $T_S$ must exclude parallel overhead. Here, parallel overhead means additional execution time required by parallel program. Intuitively, performance improvement of parallel program is expected as a speedup on $P$ processors

$$S_P = T_S / T_P$$  \hspace{1cm} (1)

Because a parallel program has a fraction of code that is serial $\alpha$, according to Amdahl’s law[2] the speedup $S_P < P$ as in equation 3. Equation 3 tell us that there is no advantage increasing the number of processors beyond $1/\alpha$ in such parallel code.

$$S_P \leq \frac{P}{1 + P\alpha}$$  \hspace{1cm} (2)

$$\lim_{P \to \infty} S_P \leq \frac{1}{\alpha}$$  \hspace{1cm} (3)

The equation in 3 implies that the speedup $S_P$ is less than $P$. It means the scalability of parallel program on parallel computer must be in either linear or sub linear speedup[1]. However, some authors reported that some cases of parallel programs achieved super linear speedup evaluated on multicore processors. Some researchers believe the facts of super linear affect by large cache size. However, the Amdahl law still valid because this law assumes that the fastest sequential execution time is used as its baseline. In this case the fastest sequential execution time of the program should be used as the absolute baseline.

B. Efficiency

In previous subsection, the fraction of code which is serial affects the scalability of parallel program on parallel computer. Not only the fraction of serial code affects the scalability, but it is also the efficiency of parallel program. In this subsection, firstly, we discuss a term called works. If a program consists of $N$ independent tasks but are sequentially executed, the total execution time of all those tasks is called work $T$, $t_i$ is execution time of task $i$.

$$T = \sum_{i=0}^{N-1} t_i$$  \hspace{1cm} (4)

Because serial program does not incur overhead, its sequential execution time $T_S = T$. However, because parallelization contributes to overhead $c = T_1 / T_S \geq 1$, the execution time of parallel program using one processor $T_1 = cT_S$. The larger $c$ is, the longer is the $T_1$. At this point, we may assume that serial program is a efficient program of 100%. We call the overhead $c$ as work-overhead. The work-overhead increases proportional to the number of parallel tasks. Therefore, the execution of parallel program on parallel computer with $P$ processors is

$$T_P = c \frac{T_S}{P}$$  \hspace{1cm} (5)

and we find the speedup as

$$S_P = \frac{T_S}{T_P} = \frac{P}{c}$$  \hspace{1cm} (6)

If $c = 1$ in equation 6 the scalability is called linear speedup. If $c < 1$ the scalability of parallel program is called sub linear speedup. We sometimes find some a case where its scalability is super linear. super linear speedup does not imply that work-overhead is less than 1. Instead of $c < 1$, total large cache size of multicore processors may decrease the execution time $T_P$ on multicore so that $T_S > cT_P \times T_P$ but it remains fix the work and overhead in $T_1 = cT_S$.

According to the equation 6, implementation of parallel program should have a low work-overhead $c$ so that the parallel program may efficiently utilize highly parallel processors. Efficient implementation of parallel program must contribute only small overhead so that its parallel execution time is fast and deliver high scalability on parallel processors such as multicore processors and manycore.

Parallel execution time is expected to decrease as the number of processors increases. However, another overhead also increases as the number of processors increases. This overhead is known as critical-path-overhead. The critical-path overhead is defined as the smallest constant $c_\infty$ in equation 7. This overhead is small enough to be neglected when the number of processor is small so that only the work-overhead was dominant. The critical-path overhead is dominant after the number of processors exceeds the degree of parallelism in a parallel software. After the number of processors reach the critical number, the parallel execution time could not improve further but the execution becoming slower.

$$T_P \leq c \frac{T_S}{P} + c_\infty T_\infty$$  \hspace{1cm} (7)

III. EFFICIENT WORK STEALING BASED EXECUTION OF PARALLEL PROGRAM

A. Work Stealing Strategy

For scalable multithreaded computation on shared memory multiprocessors such as multicore, efficient scheduling must be applied. For efficient implementation of parallel computing, the total cost spent for scheduling a set of processors should be considerably less than the total amount of useful work paid. From this point of view, coarse-grained thread scheduling
is efficient in many cases of the regular form of parallel computation. However, efficient scheduling is not the only the requirement necessary for optimal scalability. Even if the total scheduling cost is considerably less than the total work, load imbalances can be unfavorable for parallel computation. When load imbalance occur, some processors spend most of their time working while the remaining processors remain idle most of the time. Unfortunately, improving efficiency of computation by coarsening the task granularity may worsen the load imbalance.

Work-stealing[3] refers to a scheduling mechanism for parallel tasks in which parallel ask 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 processor. By work-stealing, a set of workers is grouped together. The workers are logical processor entities that execute threads. Workers that create tasks are called busy workers, 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. By work-stealing mechanism, each worker maintains a task queue. While executing a task, the worker may create new tasks and place the tasks in its task queue. Other workers may have empty task queues. Those workers with empty task queue steal tasks from busy workers. In fig 1 workers 1 and 3 have empty task queues. As a result, worker 1 steal a task from worker 0 and worker 3 steals task from worker 2.

B. Work-Stealing Implementation with Lazy Task Creation

Lazy task creation[4] is a well known technique for overcoming the problem of task granularity. Lazy task creation with task-stealing capability keeps processor loads are balanced. In lazy task creation, the tasks executions are performed in LIFO manner. This way resembles the function calls are performed. A parent-task creates child task as if a parent-function called a child-function. Doing so the overhead of lazy task creation is as small as the cost of a function call. An implementation of lazy task creation such as StackThreads/MP[5] treats a task as a asynchronous function-call. A worker creates children tasks and let a thief stealing parent continuation.

Task stealing is performed as if the children tasks were return, but victim still continue working on child tasks and thief. The function call as a task creation contributes small work-overhead and stealing a task from the bottommost stack contributes overhead differently from one strategy to others. The best technique of Cilk which uses THE protocol, task stealing does not contribute work-overhead and contribute small critical-path overhead in large-load balancing granularity cases. Figure 2 shows a worker has tasks in stack and thief steal a task from the victim’s bottommost stack. Tasks in the victim are stacked so that the total granularity is coarse. The best case is after the thief steal a task, it make task granularity coarse by creating new children.

One implementation of lazy task creation such as Cilk have additional overhead for allocating stack frames from heap.

C. Load Balancing Granularity Control

Coarsening load-balancing granularities may reduce total steal overhead. Nonetheless, it is difficult to have the load-balancing granularity of certain parallel program fits on different size of parallel processors. When the number of parallel processors vary from small to large number, controlling load balancing granularity dynamically may solve the problem. Load balancing granularity was introduced by Faxen[6] and we redefine the load balancing granularity[7] in equation 8

$$T_{S} = \sum_{j=1}^{N_{steal}} g_{steal}(j)$$

Load balancing control[7] can be performed by either fixed-length or dynamic-length work-stealing strategy[8]. Fixed-length strategy is a work stealing strategy which a thief steals fixed number of stacked tasks from the bottom of victim’s stack. Dynamic-length work-stealing is one strategy which a thief steals tasks from bottom half of a victim stack. The idea of load-balancing control is described as follow. Distribution of task granularity of parallel program is defined as $T_{S} = \sum_{i=1}^{N_{tasks}} g_{task}(i)$. The parallel tasks are scheduled by work-stealing so that $g_{steal}(j) = \sum_{i=1}^{d} g_{task}(i)$, where $d$ is either a static number of half of the number of existing tasks of a victim.

IV. EXPERIMENTAL IN WORK-STEALING BASED OF PARALLEL PROGRAM

In this section, we describe some experiments. As benchmarks in these evaluations, a benchmark from the Barcelona OpenMP Tasks suite[9] is adopted. A binomial tree of the UTS is selected as the representative of an irregular workload. We select FFT as representative of regular workload.
In our evaluations, we used GCC 4.4.3, Intel C Compiler 11.1 and GCC 2.8.1. We used GCC 4.4.3 as a complete compiler for GCC OpenMP[10]. GCC 4.4.3 also was used as the back end for Cilk. To compile all benchmarks with the OpenMP by StackThreads/MP scheduler, GCC 2.8.1 is used. We compiled all benchmarks with a -O3 compiler switch.

A. Experiment Configuration

We conducted some experiments on a machine with two 6168 AMD Opteron CPUs. Each CPU has 12 cores, 12x512KB L2 Cache and 6M shared L3 Cache. The machine is installed with Linux CentOS 5.3 as its operating system. The machine is configured with 12 GB RAM.

B. Fast Fourier Transform Benchmark

The FFT computes one-dimensional discrete Fourier transform using the Cooley-Tukey[11] algorithm. At initial stage, the FFT pre-computes coefficient \( W \) which is a matrix. After obtaining the matrix \( W \), FFT computes the factors \( r \) of length \( n \). At the final stage, the FFT divides DFT into \( r \) smaller DFTs of length \( n/r \) and multiply them by twiddle factors. This algorithm is applied to a vector of the complex data type. In this experiment, the vector sizes are 32 M of the complex data type.

From experiments on serial code and conventional OpenMP code of FFT using GCC 4.4.3 (omp task + gcc 4.4.3), obtained serial and parallel execution time of single core are 18.55 sec and 22.76 sec respectively. Hence work-overhead \( T_P/T_S = 1.22 \). Adopting work-stealing technique for OpenMP, different results are obtained. Using GNU C compiler 2.8.1 and work-stealing (omp task + ws), serial execution time is 18.55 sec and parallel execution time of single processor is 19.57. OpenMP with work-stealing shows a lower work-overhead \( T_P/T_S = 1.05 \).

Table I shows different parallel execution time on 24 cores for parallel program of FFT. Eight processor cores still scale the performance of OpenMP program for FFT. However, if more than eight processors are used, all processors fail scaling the performance further. Performance is worse when the number of cores is more than eight. We obtained different results from OpenMP program featured with work-stealing capability for the same benchmark. Sixteen cores speed the performance up to eight. More than sixteen cores could not scale the performance better and performance shows deceleration. According equation 7, critical-path overhead makes parallel execution time longer than execution time of in peak performance. In addition, table I shows speedup loses due to critical-path overhead. Figure 3 shows different scalability of different implementations of parallel FFT. Fig 3 also shows that work-overhead, although it is not dominant, makes scalability becoming sub-linear speedup. Work-stealing featuring OpenMP demonstrates better performance than conventional OpenMP.

C. Unbalanced Tree Search

The UTS[12] problem is a problem of counting the number of nodes explored in an implicit tree. Execution threads explore the tree of UTS in a depth-first search manner. In UTS, execution threads generate nodes in a parallel and recursive way. SHA-1 computation is applied to a 20-byte descriptor of the parent node to obtain a new 20-byte descriptor for each child. This 20-byte descriptor is to calculate the probability function of non leaf nodes that have \( m \) children. In the UTS, each node is a task. Nodes without children are fine-grain tasks, whereas nodes with children are non fine-grain tasks. Only the coarse grain tasks that computing SHA-1 algorithm. Therefore, load imbalance occurs between coarse-grained and fine-grained tasks. Load imbalance in the UTS benchmark depends on parameters \( m \) and \( q \). In a binomial tree, these parameters specify that a node in an unbalanced tree has \( m \) children with a probability \( q \). In the experiment of this paper,
of UTS is below linear curve. In addition, it seems that icc increases parallel execution time so that the scalability $S(1)P$ overhead. Work-overhead of icc omp (intel OpenMP task) is compiled with Intel C compiler, contributes large work-overhead. This improvement. Intel OpenMP task version of UTS, which is compiled with Intel C compiler, contributes large work-overhead. Work-overhead of icc omp (intel OpenMP task) increases parallel execution time so that the scalability $S(1)P$ of UTS is below linear curve. In addition, it seems that icc omp lost performance due to lost of locality.

V. FUTURE WORKS

Work-stealing based on lazy-task-creation, by its nature, is a task parallelism model. Nonetheless, the work-stealing also support data parallel model such as parallel loop. Parallel loop can be implemented by work-stealing and divide-and-conquer. Divide-and-conquer builds a binary tree. Thieves steal a node or a subtree from the bottom of the victim’s execution stack. A parallel frame-work that support parallel loop by work-stealing is cilk_for in cilk++ and intel cilkplus. We currently in progress doing research in this work-stealing based of parallel loop and its reducer structure. Work-Stealing based parallel-loop allows composability to perform efficiently because of low overhead.

VI. CONCLUSION

Some issues that important to be considered for the best scalability of applications in multicore and manycore processors are load imbalance and critical-path overhead. Load imbalance requires load-balancing scheduler, but small load balancing granularity may contribute to both work-overhead and critical-path overhead. In some number of processors cases, critical-path overhead will not cost the performance of parallel program. However, manycore processors challenge researchers to minimize the cost of mutex that contribute large critical-path overhead. Controlling load-balancing granularity may help work-stealing scheduler works more efficiently.

REFERENCES