Analyzing Task Parallel Program Traces based on DAG Visualization*

An Huynh[1,a)] Douglas Thain[2,b)] Miquel Pericàs[3,c)] Kenjiro Taura[1,d)]

Abstract: Task parallel programming models are conceived as a promising paradigm that brings intricate parallel programming techniques to a larger audience of programmers because of its high programmability. Programmers just need to expose parallelism in their programs by creating "task", which is a lightweight work unit that can execute in parallel with the rest, at arbitrary places in their code, then all other low-level burdens in the parallel execution of these tasks such as thread management, task scheduling, and load balancing are handled automatically by runtime systems. However, this dependence on runtime systems also hides execution mechanisms of a task parallel program from programmers, making it difficult for them to understand the cause of suboptimal performance of their programs. As an effort to tackle this problem, we have developed tools that capture and visualize the trace of an execution of a task parallel program in the form of a directed acyclic graph (DAG) which is augmented with relevant performance information on the execution. Each node of the DAG is an interval of a serial code segment in the program. Each edge represents a dependency between two nodes. This DAG visualization provides a task-centric view of the program, which is different from other popular visualizations such as thread-centric timeline visualization and code-centric hotspots analysis. Our tool (DAGViz) displays a DAG in a hierarchical manner, expands to more detailed views on demand which helps users view aggregate performance information at various levels of detail. Beside DAG view, DAGViz also provides a timeline visualization which is coordinated with the DAG. This coordination helps users relate interesting points between thread-centric timeline and task-centric DAG structure. DAGViz is expected to support effectively the process of analyzing task parallel performance and developing scheduling algorithms for task parallel schedulers.

Keywords: task parallel, performance analysis, profiler, tracer, DAG visualization

1. Introduction

Nowadays computer processors have increasingly many cores, from several ones in a commodity PC up to dozens or hundreds of them in a high performance computing server. The emerging Many Integrated Core (MIC) architecture of Intel, which combines many smaller lower-performance cores on the same chip area, has promised a highly parallel era of computer hardware. This highly parallel hardware would make it harder for programmers to program parallel software using common parallel programming models such as SPMD (MPI) and native threading libraries (POSIX Threads [1]) which involve programmers in dealing with low-level details of thread management, task scheduling, load balancing, etc.

Task parallel programming models release programmers from such low-level concerns by shifting these burdens to the runtime systems. In task parallel programming, programmers just need to expose parallelism in their programs by creating tasks. These tasks are scheduled to execute in parallel dynamically by the runtime system. A task is a lightweight work unit that can be executed in parallel with the rest of the code. Besides, task-based parallelism allows programmers to create tasks at arbitrary places in their code. This flexibility enables programmers better to express various kinds of parallelism existing in their code. However, the fact that most scheduling aspects in task parallelism are done dynamically at runtime and automatically by the runtime system leads to the consequence that a great deal of performance is out of the programmer's control. The same task parallel program executed by different task parallel runtimes could possibly present significantly different performance. And programmers often lack clues to understand why their programs perform badly.

Common analysis methods such as hotspots analysis and timeline visualization are not sufficient for task parallel programs. Hotspots analysis which shows functions that consume the most CPU time is useful in analyzing sequential execution but fails to pinpoint concurrency bottlenecks in parallel execution. Timeline visualization (Gantt chart) which displays thread activities in the course of the execution is

[1] University of Tokyo, Japan
[2] University of Notre Dame, United States
[3] Chalmers University of Technolog, Sweden
a) huynh@eidos.ic.i.u-tokyo.ac.jp
b) dthain@nd.edu
c) miquelp@chalmers.se
d) tau@eidos.ic.i.u-tokyo.ac.jp
*1 This manuscript appears in the *unrefereed* Japanese Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP) and is not a published paper.

© 2015 Information Processing Society of Japan

thread-centric and not really appropriate for task parallel programs which have dynamic scheduling nature and indeterminism in where tasks are executed. Comparing task parallel executions is more thorough when we compare them based on their logical task structures. Our approach is that we measure and extract the computation directed acyclic graph (DAG) of a task parallel execution and visualize it to analyze the performance. In a task parallel DAG, nodes represent sequential computation and edges represent task parallel dependency among nodes. The measurement part is called DAG Recorder which records the DAG into data file and the visualization part is called DAGViz which visualizes the DAG and provides visual supports for performance analysis.

In order to extract the DAG we build a simple wrapper around task parallel primitives of underlying runtimes such as task creation and synchronization primitives. We instrument at appropriate positions in this wrapper to invoke the measure code at the boundary of application code and task parallel runtime. The DAG is structured hierarchically so that higher-level collective nodes hold aggregate performance information of their inner subgraphs. These aggregate information can be viewed before expanding nodes into their subgraphs. This hierarchical DAG structure is useful for DAGViz to avoid loading the whole big DAG file into memory at once. DAGViz can load only a fraction of DAG file corresponding to the visible part of the DAG on screen. Our DAG visualization provides a quick grasp of task structure of the program and other performance information such as cores on which tasks were executed by node color. Users can also get the original code positions of individual nodes of interest. Via case studies of Sort and SparseLU programs we have demonstrated the usefulness of DAGViz in analyzing task parallel performance.

The rest of this paper is structured as following: section 2 discusses the generic task parallel computation model that our toolset supports, section 3 talks about the hierarchical DAG structure, then in section 4 we describe our DAG visualization techniques. We demonstrate usefulness of DAGViz through some case studies in section 5. Finally, related work is discussed in section 6 and conclusions is in section 7.

2. Computation Model

There are various task parallel programming models in existance. They offer some kinds of API that are slightly different from each others. We propose a generic model that our toolset supports. This model does not cover all distinct differences between existing task parallel API(s) but it is generic enough for analyzing necessary applications to our knowledge. DAG Recorder and DAGViz can record and visualize computation DAG of any task parallel programming model that conforms to this generic one. Currently we have applied our techniques to five separate systems: OpenMP [2], Cilk Plus [3], Intel TBB [4], Qthreads [5]. and MassiveThreads [6] [7].

In our generic model, a program starts as a single task performing its main function. A task can execute ordinary user computation which does not change the parallelism of the program and additionally other task parallel primitives which can change the program's parallelism. These primitives are following three semantics:

CreateTask : The current task creates a new child task.

WaitTasks : The current task waits for all tasks in current *section*, explained below, to finish. This primitive also terminates the current section.

MakeSection : This primitive is used to mark the creation of a section inside a task or another section. A section is defined as a synchronization scope which is ended by a WaitTasks primitive and all tasks created inside it get synchronized all together by that WaitTasks. The purpose of section notion is to support a task that waits for a subset of its children. Our model supports sections that are either nested or disjoint, but must not intersect.

	CreateTask	WaitTasks
OpenMP	`#pragma omp task`	`#pragma omp taskwait`
Cilk Plus	`cilk_spawn`	`cilk_sync`
Intel TBB	`task_group::run()`	`task_group::wait()`
Qthreads MassiveThreads	`pthread_create()`	`pthread_join()`

Table 1: Correspondance of five task parallel API(s) with our generic model

Task parallel primitives of OpenMP and Cilk Plus models can be translated to our model straightforwardly. The *task* and *taskwait* pragmas in OpenMP are replaced by CreateTask and WaitTasks respectively. The *spawn* and *sync* operations in Cilk Plus are also replaced by CreateTask and WaitTasks respectively (**Table 1**). In addition, however, a *task* pragma and a *spawn* operation perform an additional MakeSection operation if the current task has no open section.

Intel TBB model is more flexible than our generic one. The section notion is represented by *task_group* class in Intel TBB. A task is created by calling *run* method of a *task_group* object, and a call to a *task_group* object's *wait* method would synchronize all tasks created by that object's *run* method. One can choose an arbitrary subset of children of a task to synchronize in Intel TBB by creating these children with the same *task_group* object, whereas our generic model does not allow intersected task subsets, and a new section is opened only when the previous section has been closed. Except this restriction, Intel TBB code can be translated into our model by replacing *task_group.run* with CreateTask, *task_group.wait* with WaitTasks, and *task_group* object's declaration with MakeSection.

There are two more task parallel libraries that our system currently supports. They are Qthreads and MassiveThreads. These two are both lightweight thread libraries that expose a POSIX Threads-like interface: one function call to create a task and one function call to synchronize a task. They are as flexible as Intel TBB and translating them to our generic model is imposed with the same restriction.

ⓒ 2015 Information Processing Society of Japan

3. DAG Structure

Although an execution interval of a task parallel program between two task parallel primitives always happens entirely on a single worker (core), two consecutive intervals separated by a task parallel primitive may take place on two different workers. This is because the execution control is always given back to the runtime system at task parallel primitives where a task migration, among other runtime mechanisms, may happen and change the worker that executes the next interval.

We model an execution of a task parallel program as a DAG consisting of a set of nodes and a set of edges. Each **node** represents a sequential execution **interval** of the program, a node is associated with a seamless code segment in the program's code which is enclosed by two task parallel primitives and does not contain any primitive in it. Each **edge** represents the "task parallel" **dependency** between two nodes that it connects, or in other words each edge is a reflection of a task parallel primitive in the program's code. There are three kinds of dependencies that an edge can represent: *creation*, *continuation* and *synchronization*. An interval ended by a CreateTask primitive has a creation dependency with the first interval of the new task. Two nodes of two contiguous code segments in the program's code, one of which precedes and the other follows one same task parallel primitive, have continuation dependency. This continuation dependency can be divided further into create cont. and wait cont. based on the task parallel primitive intermediating the two intervals. The last interval of a task has synchronization dependency with the interval of the code segment following the WaitTasks primitive that synchronizes that task.

A node starts either by the first instruction of a task or an instruction immediately following CreateTask or WaitTasks, and it ends either by the last instruction of a task or an instruction immediately before CreateTask or WaitTasks. We classify nodes into three kinds by the ways how they end. A node ends by calling CreateTask primitive is of `create` kind, ends by calling WaitTasks primitive is of `wait` kind, and ends by the last instruction of a task is of `end` kind. **Fig. 1** shows an example task parallel program and the corresponding DAG of its execution.

DAG's structure is **hierarchical**. Beside three leaf node kinds of `create`, `wait` and `end`, there are two collective node kinds of `section` and `task` that comprise subgraphs of the DAG. The `section` node kind corresponds to the section notion in the task parallel programming models. A `section` node contains one or more `create` nodes and other `section` nodes before ending by one `wait` node. In Fig. 1, rounded square shapes that represent `section`(s) are illustrated to also cover tasks that its child `create` nodes created. The `task` node kind corresponds to the task entity in task parallel runtime systems. A `task` node contains zero or more `section` nodes before ending by one `end` node. In Fig. 1, the

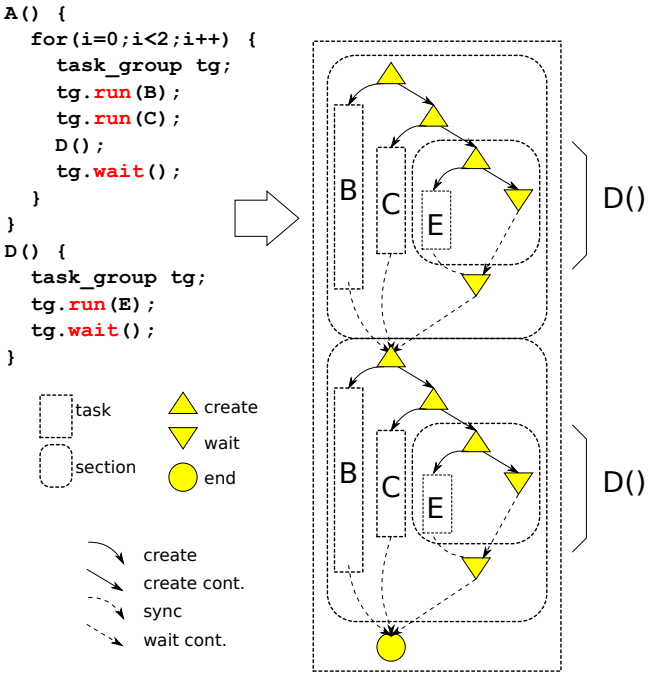


Fig. 1: An example of a task parallel program execution visualized as a DAG

whole execution of the program is originally the only `task` node. This original `task` node is expanded into two `section` nodes and one `end` node. The two `section` nodes are further expanded into two similar inner topologies as they are two iterations of the *for* loop. If a DAG is expanded completely, it has only leaf node kinds of `create`, `wait` and `end`.

Each node v in DAG is augmented with performance data of its corresponding execution interval such as start time ($v.start$), end time ($v.end$) of the interval, the worker ($v.worker$) on which the interval took place, start and end code positions (file names, lines) of the corresponding code segment of the interval. In case of collective nodes, these data items hold aggregate performance data about the node's inner subgraphs.

Recording DAG

DAG Recorder instruments code in the process through which a task parallel primitive in our generic model gets translated into a specific task parallel API so that the measurement can get called at appropriate positions during the execution of a task parallel program to record the DAG. In order to capture the DAG structure as described in the previous section, DAG Recorder instruments measure code at following seven positions:

- before and after CreateTask primitive
- before and after WaitTasks primitive
- at start and end of a task
- at start of a section

The positions after CreateTask, after WaitTasks and the start of a task are where an interval (a node) begins. The positions before CreateTask, before WaitTasks and the end of a task are where an interval ends.

ⓒ 2015 Information Processing Society of Japan

4. DAG Visualization

Capturing all relevant events then visualizing them is a comprehensive strategy to analyze thoroughly the cause of performance bottlenecks. In the particular case of task parallelism, all the indeterministics and dynamics which are the cause of performance problems happen inside task parallel primitives, so breaking down the entire execution into sequential segments separated by the primitives is a validly practical approach. Being able to manipulate these sequential pieces (DAG's nodes) which are the building blocks of task parallel performance allows us to boil down to the root cause of the problem.

Moreover, different from code-centric hotspots analysis and thread-centric timeline visualization, DAG visualization provides a task-centric view of the execution which is the logical task structure of the program. This logical task structure is more relevant and familiar from the programmers' perspective. As well, there is a need to compare task parallel executions to clarify the subtle differences among runtimes or among scheduling policies for the purpose of developing appropriate scheduling algorithms. Due to the freedom in assigning which workers execute which tasks it is more meaningful when comparing executions task by task.

4.1 Hierarchical Layout Algorithm

Because the DAG structure is hierarchical beginning with a single `task` node representing the whole original application, we can traverse all DAG's nodes recursively beginning from this root `task`.

A node holds such coordinate variables that x, y are absolute coordinates of the node, *xpre* is the relative x coordinate based on its predecessor node, *xp* is the relative x coordinate based on its parent node. *lw*, *rw* and *dw* which stand for left width, right width and down width (distances from point x, y to the left, right and down) describe the bounding box covering itself and all its inner children. *link_lw*, *link_rw* and *link_dw* describes the bounding box covering itself, its subgraph and all successor nodes reached when traversing along the links, and their subgraphs too. The layout algorithm traverses the DAG recursively and sets values to these variables of each node. The algorithm consists of two phases. In the first phase, it sets values for each node's xpre, y, lw, rw, dw and link_lw, link_rw, link_dw. In the second phase, it sets values for xp and absolute x coordinate of every node.

Fig. 2 shows the DAG extracted from an execution of Sort program. The DAG is originally only one node (left most), from left to right it shows the DAG's hierarchial expansion. The original node gets expanded into three `section`s and one `end`. Next the first `section` gets expanded, then the second `section` and the third `section`. When users click on a node, DAGViz displays a box showing detailed information that DAG Recorder has recorded about that interval. Node color represents for the worker that has executed the node. The mixed color (of orange, yellow and cyan) represents for the subgraphs that have been executed by multiple workers rather than a single one. **Fig. 3** shows the same DAG that has been expanded to depth 6 while the full DAG has max depth of 66 and contains dozens of thousands of nodes.

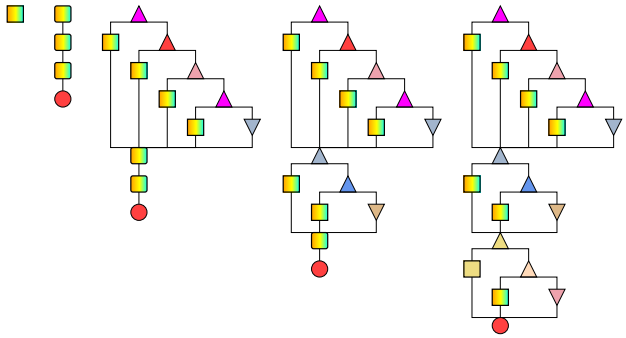


Fig. 2: Sort's DAG(s) at depth 0 (first), 1 (second) and 2 (later 3). Node color represents the worker that executed the node, the mixed color (of orange, yellow and cyan) indicates the node's subgraph was executed collectively by multiple workers.

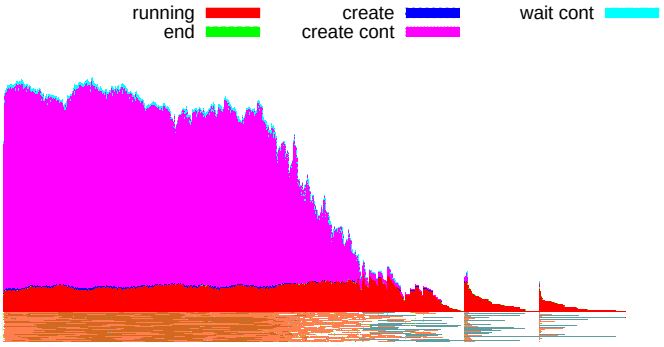


Fig. 4: Sort's timeline is the lower part consisting of 32 rows. Sort's parallelism profile is the upper part consisting of a red area (actual parallelism) and stacked-up areas of other colors (different kinds of available parallelisms).

4.2 Timeline View with Parallelism Profile

The layout algorithm of the DAG can be modified a little to produce timeline view of the execution. In timeline view the x-axis is the time flow and y-axis consists of a number of rows each of which corresponds to one worker thread. The rows contain boxes representing works that a worker was doing at specific points in time during the program's execution. Each node of the DAG becomes a box in the timeline, so its y coordinate is fixed based on its worker number. The node's x coordinate is calculated based on its start time, and its length is based on its work time ($= v.end - v.start$). Therefore, the timeline layout algorithm is somewhat easier than that of the DAG. Besides, DAGViz also draws a parallelism profile along with and above the timeline (**Fig. 4**). In Fig. 4, the lower part consisting of 32 rows is the timeline, the upper part (from red area upward) is the parallelism profile of the execution which is the time series of actual and available parallelisms of the execution:

- Time series of actual parallelism (red part): is the num-

ⓒ 2015 Information Processing Society of Japan

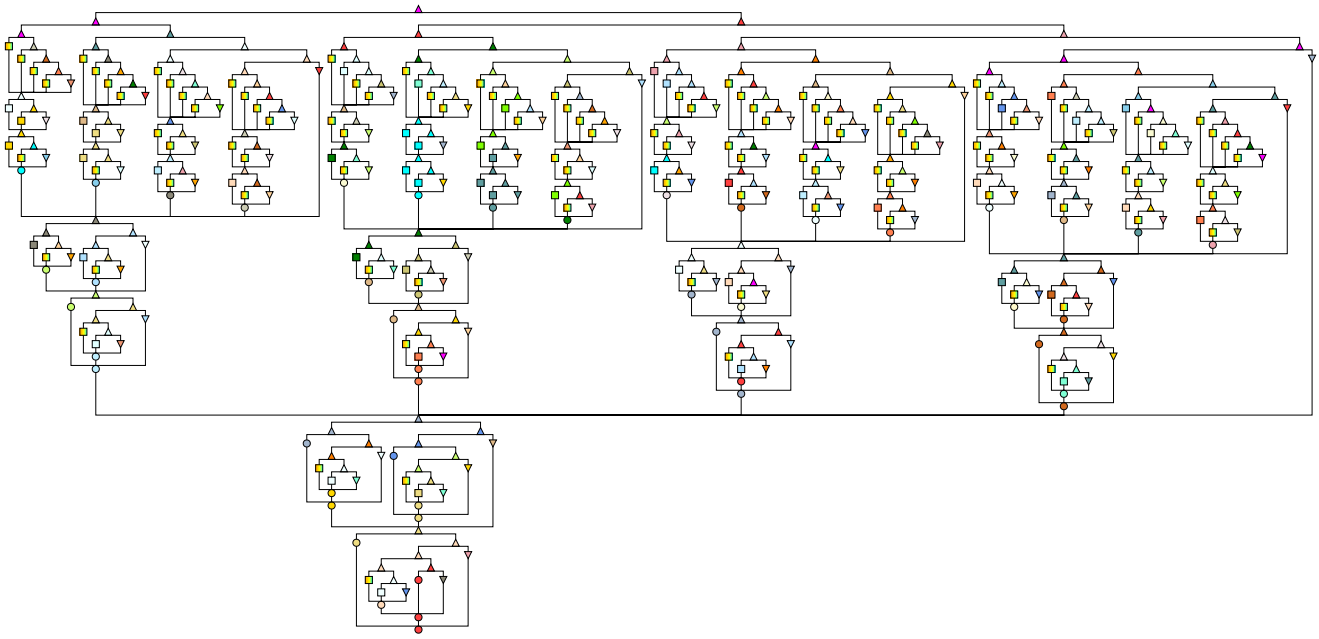


Fig. 3: Sort's DAG expanded to depth 6 (max depth is 66)

ber of tasks actually running at every point in time. Actual parallelism at time t, denoted by $P_{\text{actual}}(t)$, can be obtained by:

$$P_{\text{actual}}(t) = \sum_{v \in V} \text{running}(v, t)$$

where V is the set of all nodes in DAG, running(v, t) is 1 if v is running at time t and 0 otherwise. Formally,

$$\text{running}(v, t) = \begin{cases} 1 & \text{if } v.\texttt{start} \leq t \leq v.\texttt{end} \\ 0 & \text{otherwise} \end{cases}$$

- Time series of available parallelism (upper parts of other colors): is the number of tasks *ready to run* but not actually running at every point in time. Available parallelism at time t, $P_{\text{avail}}(t)$, can be obtained by:

$$P_{\text{avail}}(t) = \sum_{v \in V} \text{ready}(v, t)$$

where ready(v, t) is 1 if all of v's predecessors have been finished at time t but v has not been started; and 0 otherwise. Formally,

$$\text{ready}(v, t) = \begin{cases} 1 & \text{if } u.\texttt{end} < t < v.\texttt{start} \text{ for all} \\ & u \rightarrow v \\ 0 & \text{otherwise} \end{cases}$$

5. Case Studies

We have measured and recorded DAG(s) of all ten programs in the Barcelona OpenMP Task Suite (BOTS) benchmark suite [8] with five task parallel runtime systems DAG Recorder currently supports; OpenMP, Cilk Plus, Intel TBB, MassiveThreads and Qthreads. The experimental environment is shown in **Table 2**, and parameters for each benchmark described in **Table 3**. The overhead of DAG Recorder is shown in **Fig. 5**. Except for particular cases of Health and UTS programs which have many fine-grained tasks, DAG Recorder is feasible for all other programs with overhead within 10% of the original program's runtime.

Compiler	Intel Compiler 14.0.2
OS	CentOS 6.4 (Linux 2.6.32-x86_64)
CPU	AMD Opteron 6380 2.5GHz 16 cores (8 modules) per socket
# Sockets	4 sockets (64 cores or 32 modules in total)
Runtimes	OpenMP, Cilk Plus, Intel TBB, MassiveThreads, Qthreads

Table 2: Experiment environment

App	stack	cut off	other args
Alignment	2^{20}	-	`-f prot.100.aa`
FFT	2^{15}	-	`-n` 2^{24}
Fib	2^{15}	manual	`-n 47 -x 19`
Floorplan	2^{17}	manual	`-f input.20 -x 7`
Health	2^{14}	manual	`-f medium.input -x 3`
Nqueens	2^{14}	manual	`-n 14 -x 7`
Sort	2^{15}	manual	`-n` 2^{27} `-a 512 -y 512`
Sparse LU	2^{14}	-	`-n 120 -m 40`
Strassen	2^{14}	manual	`-n 4096 -x 7 -y 32`
UTS	2^{14}	-	`-f tiny.input`

Table 3: Summary of benchmarks settings

We have summarized the experimental results in **Fig. 6**, which plots the utilization (= speedup/cores) of all ten programs with all five systems, using 32 cores. Each dot represents the utilization of an execution of a program by a system; the higher it is, the better. Among many cases of our interest, we look into two of them here. First, Sort's speedup is poor in all systems, which suggests that the program's code is the cause of performance bottleneck. The other case is SparseLU, as it is a peculiar case in which Cilk Plus's scalability is poorer than other systems, when Cilk Plus performs well in most other benchmarks.

ⓒ 2015 Information Processing Society of Japan

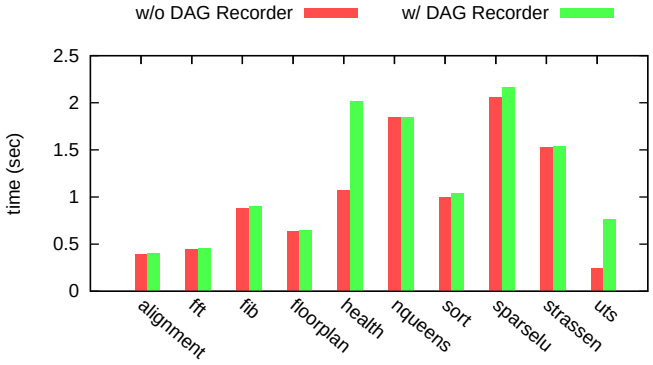


Fig. 5: DAG Recorder's overhead in running programs in BOTS with MassiveThreads on 32 cores

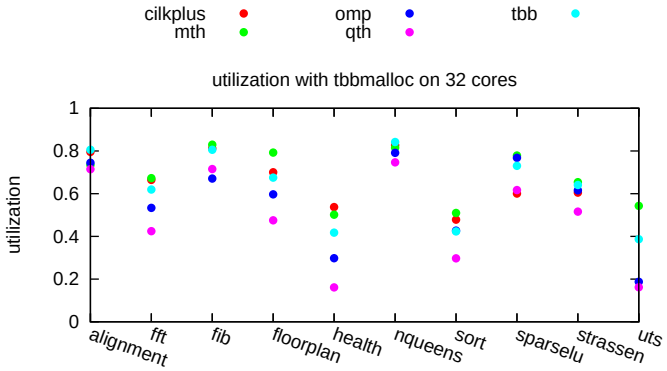


Fig. 6: Utilizations of BOTS run by 5 systems on 32 cores

Sort

Sort program sorts a random permutation of n 32-bit numbers with a parallel variation of mergesort [8]. The input array is divided into smaller parts which are sorted recursively before being merged, also recursively, to become the sorted result array. The recursive parallel merge is turned to simple sequential memory copy when the smaller array is empty. This trivial condition itself causes the lack of available parallelism accompanied with many long-running tasks at the stage near the end of the execution in Fig. 4. It is because the condition that the smaller array is empty does not guarantee the larger array is sufficiently small, but contrarily the larger array might be very large making the sequential memory copy costly. By replacing this sequential memory copy with a version of parallel memory copy, the lack of parallelism in merging phase was fixed.

Similar to Sort, Strassen is another example where performance suffers from the lack of parallelism. The timeline of Strassen program in **Fig. 7** shows that the program's parallelism is very low near the start. By zooming in and relating the long running box with DAG structure, we identified the code segment which enforced this low parallelism situation.

SparseLU

SparseLU program computes an LU matrix factorization over sparse matrices [8]. DAG visualization of SparseLU (**Fig. 8**) and also its source code show that it has a serial loop creating very many tasks, none of which recursively creates further tasks. Therefore, the program's parallelism in-

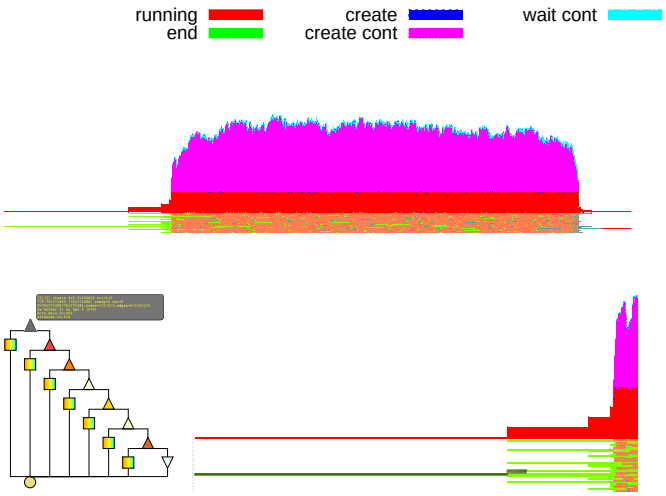


Fig. 7: Strassen's case study: top node of the DAG (first interval of the execution) is actually a too-long-running one demonstrated by the timeline view.

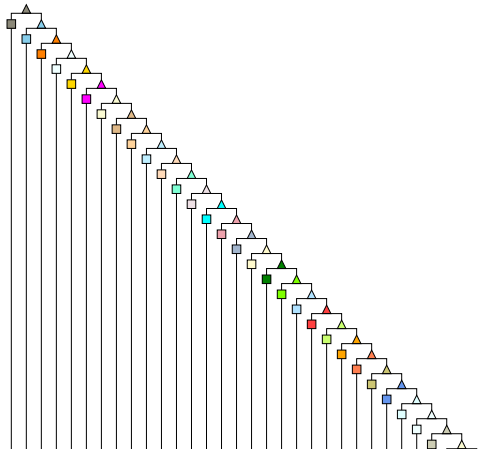


Fig. 8: (Head part of) SparseLU's DAG by Cilk Plus

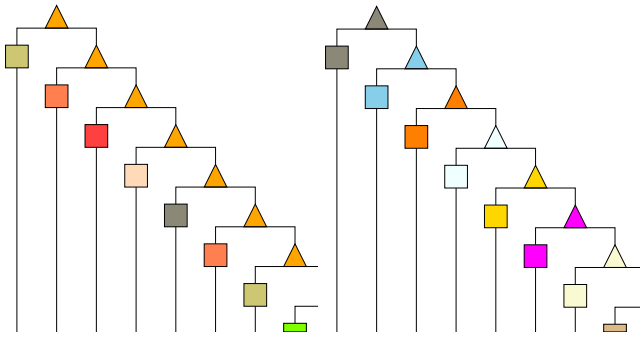


Fig. 9: (Head parts of) SparseLU's DAG(s) by Intel TBB (left) and Cilk Plus (right)

crements one only after each iteration of the loop. The comparison of DAG(s) from Cilk Plus and Intel TBB in **Fig. 9** expresses a noticeable difference between two systems. All nodes along the spine in Intel TBB's DAG (left one) are executed together by the same worker (of orange color), whereas in Cilk Plus's DAG (right one) these spinal nodes are executed separately by different workers (of different colors). This is because in Intel TBB when a worker creates a new

ⓒ 2015 Information Processing Society of Japan

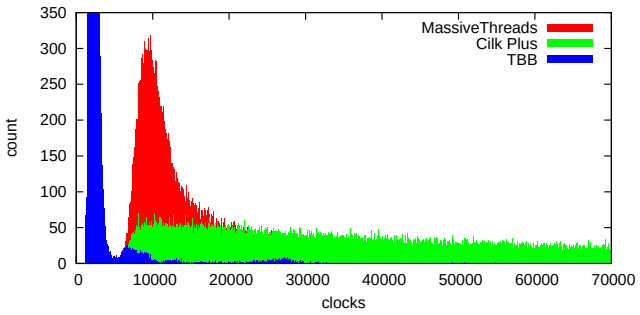


Fig. 11: Distribution of work stealing delay in SparseLU

task it pushes the new task into its work queue and continues executing the current one, whereas in Cilk Plus the worker would pause the current task to switch to executing the new task. So, every parallelism increment requires a work stealing operation in Cilk Plus's execution. It is understandable that systems with help-first policy (OpenMP, Intel TBB, Qthreads) would execute SparseLU better than systems with work-first policy (Cilk Plus, MassiveThreads).

However, MassiveThreads still has significantly better utilization than Cilk Plus. **Fig. 10** shows parallelism profiles of MassiveThreads and Cilk Plus on 32 cores. It is noticeable that Cilk Plus exposes a low parallelism (around 25, as opposed to nearly 32 with MassiveThreads). The reason why MassiveThreads performs better than Cilk Plus can be explained by the expensive work stealing of Cilk Plus. Figure 11 compares the distribution of time gaps between two consecutive nodes on the spine. Cilk Plus takes much longer to advance a computation along it, implying that it takes longer to steal a task. In our previous microbenchmark, we have confirmed that work stealing operation in MassiveThreads is more than an order of magnitude faster than in Cilk Plus [9].

6. Related Work

Tallent et al. [10] categorized parallel execution time of a multithreaded program into 3 categories: *work*, *parallel idleness*, and *parallel overhead*. They use sampling method that interrupts workers regularly after a fixed period of time to record a sample of where workers are working on. They proposed techniques to measure and attribute *parallel idleness* and *parallel overhead* back to application-level code based on an additional binary analysis process of the executable to re-construct the program's user-level call path. Their approach has been implemented in the HPCToolkit performance tool of the Rice University. They claim that these two *parallel idleness* and *parallel overhead* metrics can help to pinpoints areas in a program's code where concurrency should be increased (to reduce idleness), or decreased (to reduce overhead).

Olivier et al. [11] had taken a step further than [10] by identifying that the inflation in *work* is in some cases more critical than *parallel idleness* or *parallel overhead* factors in task parallelism. They systemize the contributions of the 3 factors of *work inflation*, *idlness* and *overhead* in the performance loss of applications in BOTS. They demonstrated that work inflation accounted for a dominant part and proposed a locality-aware scheduler which mitigated this factor.

The TAU performance system [12] is an open source system that has a powerful automatic instrumentation toolset. Intel VTune Amplifier software [13] uses sampling method and does not need to instrument the executable. These tools focus on the analysis of only one single execution of the application. They can pinpoint the most costly code blocks in the application-level code which consume most of the execution time. To analyze the *work inflation* factor we need to compare a pair of executions on fewer and more numbers of cores, which these tools do not support.

Liu et al. [14] has built a NUMA profiler for multithreaded programs. It can assess the severity of remote access bottleneck and provide optimization guidance of redistributing data based on memory access patterns of threads. But for task-parallel applications, when tasks are distributed dynamically, the solution needs to take into account the structure of the DAG.

Vampir [15] visualizes trace files of an MPI program. Its main visualization is a timeline view (Gantt chart) with edges pointing from box to box to represent communication among processes. It simultaneously shows a statistical view that displays aggregate information of a chosen time interval in the timeline. Iwainsky et al. [16] have used Vampir to visulize remote socket traffic on the Intel Nehalem-EX. Jumpshot [17] is a more general timeline visualizer. It visualizes data from text files of its own format. Jumpshot is not very flexible. It can only display up to 10 different categories which have 10 different colors. Jedule [18] is a tool to visualize schedules of parallel applications in timeline style. Olivier et al. in [11] has used Jedule to visualize a timeline view for analyzing the locality of a scheduling policy.

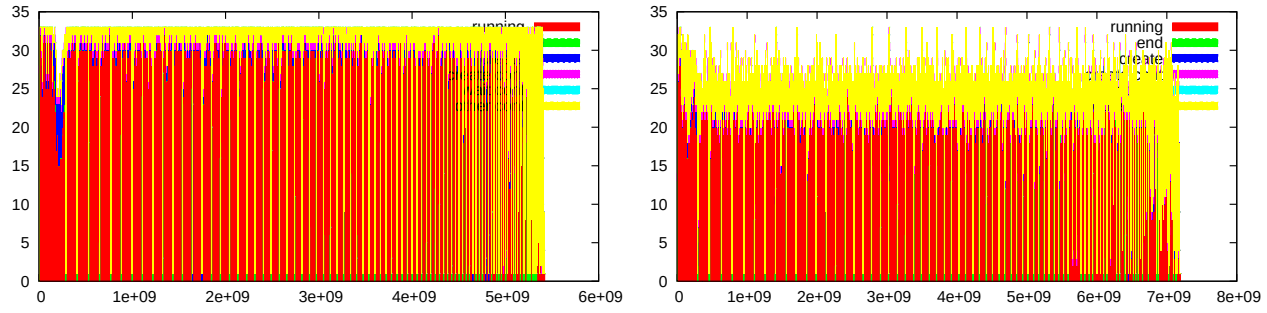
Wheeler and Thain [19] in their work of ThreadScope have demonstrated that visualizing a graph of dependent execution blocks and memory objects can enable identification of synchronization and structural problems. They use existing tracing tools to instrument multithreaded applications, then transform result traces to dot-attributed graphs which are rendered by GraphViz [20]. GraphViz tool is scalable up to only hundreds of nodes and slow with large graphs of more than a thousand nodes because its algorithm [21] focuses on the aesthetic aspect of graphs rather than rendering speed. And most of all, GraphViz is not interactive.

Aftermath [22] is a graphical tool that visualize traces of an OpenStream [23] parallel programs in timeline style. OpenStream is a dataflow, stream programming extension of OpenMP. Although Aftermath is applied in a narrow context of OpenStream (subset of OpenMP), it instead provides an extensive functionalities for filtering displayed data, zooming into details and various interaction features with users.

7. Conclusions & Future Work

We have built DAGViz - a tool that visualizes the DAG

ⓒ 2015 Information Processing Society of Japan



(a) MassiveThreads (b) Cilk Plus

Fig. 10: SparseLU's parallelism profiles by MassiveThreads and Cilk Plus. While MassiveThreads constantly reaches 32 parallelism, Cilk Plus mostly floats around 25.

extracted from an execution of a task parallel program and provides interaction functionalities for the user to explore the DAG. Through case studies, DAG visualization has proved its usefulness in helping users to understand the structure of task parallel programs. Along with DAG view, DAGViz can also visualize timeline view with parallelism profile of the DAG which provides a thread-centric view on the execution.

In future work, we would like to implement and combine the sampling method with current instrumentation measurement to get a more complete observation of long running intervals. DAG Recorder currently records only time metrics, we intend to enhance it to record hardware performance counters [24] as well in order to get more thorough measures to reason about the performance. DAGViz and DAG view will be enhanced to convey performance insights to the users. Besides, comparing isomorphic DAG(s) produced by the same program running on different configurations to analyze work stretch and performance difference among systems is a potential direction of DAGViz.

References

[1] Garcia, F. and Fernandez, J.: POSIX Thread Libraries, *Linux J.*, Vol. 2000, No. 70es (online), available from ⟨http://dl.acm.org/citation.cfm?id=348120.348381⟩ (2000).

[2] OpenMP Architecture Review Board: OpenMP Application Program Interface, Technical Report July, OpenMP Architecture Review Board (2013).

[3] Intel: Intel Cilk Plus homepage. Accessed Sep. 16th, 2014.

[4] Pheatt, C.: Intel(R) Threading Building Blocks, *J. Comput. Sci. Coll.*, Vol. 23, No. 4, pp. 298–298 (online), available from ⟨http://dl.acm.org/citation.cfm?id=1352079.1352134⟩ (2008).

[5] Wheeler, K. B., Murphy, R. C. and Thain, D.: Qthreads: An API for programming with millions of lightweight threads, *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, pp. 1–8 (online), DOI: 10.1109/IPDPS.2008.4536359 (2008).

[6] Nakashima, J. and Taura, K.: MassiveThreads: A Thread Library for High Productivity Languages, *Festschrift of Symposium on Concurrent Objects and Beyond: From Theory to High-Performance Computing (to appear as a volume of Lecture Notes in Computer Science)*, (online), available from ⟨http://www.eidos.ic.i.u-tokyo.ac.jp/ nakashima/index_e.html⟩ (2012).

[7] Nakashima, J., Nakatani, S. and Taura, K.: Design and implementation of a customizable work stealing scheduler, *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers - ROSS '13*, New York, New York, USA, ACM Press, p. 1 (online), DOI: 10.1145/2491661.2481433 (2013).

[8] Duran, A., Teruel, X., Ferrer, R., Martorell, X. and Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, *2009 International Conference on Parallel Processing*, IEEE, pp. 124–131 (online), DOI: 10.1109/ICPP.2009.64 (2009).

[9] Taura, K. and Nakashima, J.: A Comparative Study of Six Task Parallel Programming Systems (in Japanese), *IPSJ SIG Technical Report HPC*, Vol. 140(16), IPSJ, pp. 1–10 (2013).

[10] Tallent, N. R. and Mellor-Crummey, J. M.: Effective Performance Measurement and Analysis of Multithreaded Applications, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, New York, NY, USA, ACM, pp. 229–240 (online), DOI: 10.1145/1504176.1504210 (2009).

[11] Olivier, S. L., de Supinski, B. R., Schulz, M. and Prins, J. F.: Characterizing and Mitigating Work Time Inflation in Task Parallel Programs, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 65:1–65:12 (2012).

[12] Shende, S. S. and Malony, A. D.: The Tau Parallel Performance System, *Int. J. High Perform. Comput. Appl.*, Vol. 20, No. 2, pp. 287–311 (online), DOI: 10.1177/1094342006064482 (2006).

[13] Intel: Intel VTune Amplifier, Intel Inc. (online), available from ⟨http://software.intel.com/en-us/intel-vtune-amplifier-xe⟩ (accessed 2013).

[14] Liu, X. and Mellor-Crummey, J.: A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures, *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, New York, NY, USA, ACM, pp. 259–272 (online), DOI: 10.1145/2555243.2555271 (2014).

[15] Nagel, W. E., Arnold, A., Weber, M., Hoppe, H.-C. and Solchenbach, K.: VAMPIR: Visualization and Analysis of MPI Resources, *Supercomputer*, Vol. 12, pp. 69–80 (1996).

[16] Iwainsky, C., Reichstein, T., Dahnken, C., Mey, D., Terboven, C., Semin, A. and Bischof, C.: An Approach to Visualize Remote Socket Traffic on the Intel Nehalem-EX, *Euro-Par 2010 Parallel Processing Workshops* (Guarracino, M., Vivien, F., Trff, J., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knpfer, A., Di Martino, B. and Alexander, M., eds.), Lecture Notes in Computer Science, Vol. 6586, Springer Berlin Heidelberg, pp. 523–530 (online), DOI: 10.1007/978-3-642-21878-1_64(2011).

[17] Zaki, O., Lusk, E. and Swider, D.: Toward Scalable Performance Visualization with Jumpshot, *High Performance Computing Applications*, Vol. 13, pp. 277–288 (1999).

[18] Hunold, S., Hoffmann, R. and Suter, F.: Jedule: A Tool for Visualizing Schedules of Parallel Applications, *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pp. 169–178 (online), DOI: 10.1109/ICPPW.2010.34 (2010).

[19] Wheeler, K. B. and Thain, D.: Visualizing massively multithreaded applications with ThreadScope, *Concurrency and Computation: Practice and Experience*, Vol. 22, No. 1, pp. 45–67 (online), DOI: 10.1002/cpe.1469 (2010).

[20] Bilgin, A.: Graphviz - Graph Visualization Software (1988).

[21] Sugiyama, K., Tagawa, S. and Toda, M.: Methods for Visual Understanding of Hierarchical System Structures, *Systems, Man and*

ⓒ 2015 Information Processing Society of Japan

Cybernetics, IEEE Transactions on, Vol. 11, No. 2, pp. 109–125 (online), DOI: 10.1109/TSMC.1981.4308636 (1981).

[22] Drebes, A., Pop, A., Heydemann, K., Cohen, A. and Drach-Temam, N.: Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems, *Proceedings of 7th Workshop on Programmability Issues for Heterogeneous Multicores*, MULTIPROG '14 (2014).

[23] Pop, A. and Cohen, A.: OpenStream: Expressiveness and Data-flow Compilation of OpenMP Streaming Programs, *ACM Trans. Archit. Code Optim.*, Vol. 9, No. 4, pp. 53:1–53:25 (online), DOI: 10.1145/2400682.2400712 (2013).

[24] Mucci, P. J., Browne, S., Deane, C. and Ho, G.: PAPI: A Portable Interface to Hardware Performance Counters, *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10 (1999).

© 2015 Information Processing Society of Japan