修士論文

DAGViz: A DAG Visualization Tool for Analyzing Task Parallel Programs (DAGViz: タスク並列計算の性能解析のためのDAG可視化ツール)

平成27年2月5日提出

指導教員　　田浦 健次朗 准教授

電子情報学専攻

48-136431 Huynh Ngoc An

Abstract

Task parallel programming model has been considered as a promising mean that brings parallel programming to larger audience thanks to its high programmability. In task parallel programming, programmers just need to specify tasks that can be executed in parallel then these tasks would be distributed to available processor cores and executed in parallel dynamically by the runtime system. However, this dynamic characteristics of task parallelism hides all execution mechanisms of a task parallel application from programmers, which makes it difficult for them to understand suboptimal performance of their application. We have developed tools to capture relevant data during the execution of a task parallel application as a directed acyclic graph (DAG) of sequential code sections of application code. then visualize captured data. Our visualizer displays DAG in a hierarchical way that helps users conceive DAG structure at various levels of detail. It also provides multiple views for a single DAG and supports coordination between them, which has yielded interesting information as we case-study the applications in Barcelona OpenMP Task Suite (BOTS). Specifically, the tool could pinpoint relevent code sections that causes low parallelism time periods of interest. Moreover, this approach is prospective in the way that we can visually compare two isomorphic DAGs generated by the same application running on different environments which we intend to do in future work. This comparison is expected to expose differences between task parallel runtime systems and exhibit insights useful for developing scheduling algorithm.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to some physical limitations such as heat dissipation, the development of computer hardware has changed from increasing clock speed of a single-core CPU to increasing the number of cores integrated in a multi-core CPU for more than a decade. There are more and more cores which are integrated in a computer's CPU, from 4 to 8 ones in a commodity PC up to 64 ones in a high performance computing server. Along with that change in hardware, another programming model is needed to create software that can run on the new hardware architecture which is usually called as parallel programs. A parallel program consists of multiple computing threads which run at the same time, and each thread resides on one separate core.

A traditional approach to creating parallel programs on shared memory systems is that the programmer demands thread creation one by one and assign work for each thread by themself. Given that a computer has 64 cores, a programmer who wants to use all that computer's processing power has to write code to create 64 threads and divide work to 64 portions in his program. POSIX Threads [1] is a library that provides this kind of programming model. However, this kind of programming style makes programmers exhausted of thread managing work rather than focusing their strength on developing the program's algorithm. Moreover, recent surge of Many Integrated Core (MIC) architecture developed by Intel has put a new prospect on shared memory parallel programs which may now need to manage up to several hundred simultaneously running threads. There is a need for a parallel programming model that releases programmers from low-level detailed concerns so that they can take care better of higher-level things and parallel programming can reach to more programmers who are usually ignorant of low-level system and hardware knowledge.

Task parallel programming models help solving this problem. It has a very high programmability, meaning easy to use for programmers to create parallel program. In the next section, we talk about some task parallel programming models and the common API that we created to unify their code.

1.1 Task Parallel Programming Models

In task parallelism, programmers just need to create as many tasks as possible, each task is an execution of some function, and a task can create other tasks recursively. The other burdens in parallel execution such as thread management, task stealing/migration, load balancing are taken care by the runtime system. Therefore, the runtime system plays an important role in the performance of a task parallel program.

Figure 1.1 illustrates a pseudo-code example of the *fibonacci* program written in task parallelism. This example exhibits two main interfaces of a task parallel programming model. They are an interface to create a task (Create_Task) and another interface to wait for tasks that have been created (Wait_Tasks). This simple example has demonstrated task parallelism's high programmability. As programmers just need to transform their function calls into task

```
1  int fib(int n) {
2    if (n < 2)
3      return 1;
4    else {
5      int x, y;
6      x = Create_Task( fib(n−1) );
7      y = Create_Task( fib(n−2) );
8      Wait_Tasks();
9      return x+y;
10   }
11 }
```

Figure 1.1. A task parallel *fibonacci* program

creation primitives so that their program becomes a parallel one which can run on any (shared-memory) parallel hardware environment, task parallelism is easy to use and particularly fit with divide-and-conquer algorithms. Thus, it is promising to bring the tricky parallel programming technique to a wider adoption among general programmers.

1.1.1 A Common Interface for All Six Task Parallel Systems

There are various task parallel programming models existing such as *OpenMP* [2], *Cilk* [3] (*Intel CilkPlus* [4]), *Intel TBB* [5], *QThreads* [6], *MassiveThreads* [7] [8], and *Nanos++* [9]. They may be a language (*OpenMP*, *Cilk*, *Intel CilkPlus*) or just a library (*Intel TBB*, *QThreads*, *MassiveThreads*, *Nanos++*) that provides interface functions to access the task parallel runtimes. Although each model has distinct differences in its API, they all support the two basic interfaces showed in Figure 1.1, Create_Task and Wait_Tasks.

OpenMP model defines an additional set of compiler directives to a language (C, C++, Fortran) in order to provide task parallelism for that language. In *OpenMP*, Create_Task manifests as a directive pragma beginning with "*#pragma omp task*", and Wait_Tasks manifests as a pragma of "*#pragma omp taskwait*" (Figure 1.2b). *Cilk* language is formed by adding some additional keywords to the C language to support task parallelism. These keywords include "*spawn*" which is equivalent to Create_Task, and "*sync*" which is equivalent to Wait_Tasks (Figure 1.2c).

On the other hand, *Intel TBB* model is a C++ library providing *task_group* class in the *tbb* namespace. *task_group*'s *run* function corresponds to the Create_Task primitive and *task_group*'s *wait* function corresponds to the Wait_Tasks primitive (Figure 1.2d). We have implemented another *task_group* version that interfaces with the other 3 task parallel libraries *QThreads*, *MassiveThreads* and *modelnanox*. Thus, writing code for these libraries is as much similar as writing code for *Intel TBB*.

In order to evaluate all these six task parallel systems together with ease we have created a common application program interface (API) covering all common grounds as well as distinct differences between them. Another primitive of Make_Task_Group (*mk_task_group*) is added to represent *Intel TBB* model's task_group declaration, which does not exist in *OpenMP* and *Cilk* models. We built a macro wrapper that translates code based on this common API into six individual executables corresponding to the six systems in compile time. An example of the common API used in *fibonacci* program is showed in Figure 1.2a. Figure 1.2 gathers 4 versions of the *fib* program based on the common API, *OpenMP*, *Cilk* and *Intel TBB* models. In addition, it also illustrates how a common API code gets transformed into these models. In general, the method is that a common syntax between the common API and the target model is replaced by the target model's semantics, and a syntax that exists in the common API but does not exist in the target model will get deleted.

This common API eases the process of porting a program's code into all six runtime systems. We now just need to write code once and get it compiled to six individual executables corresponding to the six systems.

1.1.2 Critical Role of Task Parallel Runtimes

In task parallelism, the burden on programmers has been released, but that on the runtime system has been piled up instead. The runtime system's job in task parallelism can be generalized in following 3 parts:

- interfacing with the underlying hardware.
- managing all created tasks and their parent-child relationships.
- delivering tasks to free doing-nothing threads so that the work is balanced between available threads.

All these parts are done dynamically at runtime. A common approach to the first part is that the runtime system would initiate a certain number of concurrent threads at the beginning of the program's execution, which corresponds to the number of available processor cores in the underlying hardware, each thread is bound to a single separate processor core, and usually referred to as *worker threads* or simply *workers*. For the second part, each worker maintains a work queue of its own, and every task is stored in the work queue of the worker on which it is created. A worker executes tasks in its work queue one by one until it runs out of tasks (its work queue gets empty). At that time, it will go to steal work from other workers who have tasks waiting in their queues. The stealing mechanism is that the free worker chooses a victim worker randomly, then goes to see if the victim worker's work queue has any task available, if there is, it will migrate that task to its thread and execute it. If there is no task remained in the victim's queue, it will choose another victim and go stealing again, and continue with other victims continuously until it can steal a task. This load balancing mechanism is called *work stealing* [10]. Work stealing is one of the mechanisms that a task parallel runtime uses to accomplish its third part.

When a worker creates a new task, it has two choices to proceed, one is to pause the current task and switch to executing the new task. The other is opposite, the worker pushes the new task into its work queue and continue executing the current task. These approaches are usually referred to as *work-first*, and *help-first* respectively [11]. *Work-first*'s execution order is similar to that of a serial execution. Therefore, it is expected to maintain the best data locality and hence perform better than *help-first*. On the other hand, *help-first* tends to spawn as many tasks as possible, exhibiting better parallelism for the scheduler to feed available workers. There is no absolute answer to the question of which one, *work-first* or *help-first*, is better yet. Because which one performs better may also depends on the computation model of the program's algorithm and possibly the number of underlying available workers. As we know, *OpenMP*, *Intel TBB* and *QThreads* adopt *help-first* policy in their schedulers. *Intel CilkPlus* and *MassiveThreads* adopt *work-first*.

Apparently the third part doing load balancing is the most important part of a task parallel runtime system which decides its implementation's quality. The better it is implemented the better performance the runtime can deliver when executing task parallel programs. However, the fact that load balancing is done dynamically at runtime and automatically by the runtime system leads to another fact that a great deal of performance formation is out of the programmer's control. The same task parallel program, meaning the same algorithm, executed by different task parallel runtimes could possibly present significantly different degrees of performance. And the programmer has no clue of why it happens because all mechanisms inside the runtimes are hidden from him.

Solving this problem is critical to the development of task parallel programming models. As an attempt to attack it, we have built a tool that records relevant events during the execution of

```
cilk int fib(int n) {
  if (n < 2)
    return 1;
  else {
    int x, y;
    mk_task_group;

    create_task( x, x = spawn fib(n−1) );

    create_task( y, y = spawn fib(n−2) );

    wait_tasks;
    return x+y;
  }
}
```

(a) Common API

```
  int fib(int n) {
  if (n < 2)
    return 1;
  else {
    int x, y;
                    ;
#pragma omp task shared(x)
                        x = fib(n−1) );
#pragma omp task shared(y)
                        y = fib(n−2) );
#pragma omp taskwait
                ;
    return x+y;
  }
}
```

(b) OpenMP

```
cilk int fib(int n) {
  if (n < 2)
    return 1;
  else {
    int x, y;
                    ;

                        x = spawn fib(n−1) ;

                        y = spawn fib(n−2) ;

    sync;
    return x+y;
  }
}
```

(c) Cilk

```
  int fib(int n) {
  if (n < 2)
    return 1;
  else {
    int x, y;
    tbb::task_group tg;

    tg.run([&x,=] { x = fib(n−1); });

    tg.run([&y,=] { y = fib(n−2); });

    tg.wait();
    return x+y;
  }
}
```

(d) Intel TBB

Figure 1.2. Common API code gets translated into *OpenMP*, *Cilk* and *Intel TBB* codes automatically

a task parallel program so that we can analyze the performance of the execution port-mortemly. It is called DAG Recorder and built upon the common API so it can work with all six task parallel systems.

1.2 DAG Recorder

DAG Recorder is a performance measurement module that runs along with an execution of a task parallel program and records all relevant performance events occurring during that execution then stores them in a format of a directed acyclic graph (DAG) of nodes and edges. DAG Recorder is built inside the common API, so programs written in this common API can use DAG Recorder immediately with a minimal involvement of the programmer. This is following three main API functions that the programmer need to remember:

- dr_start()
- dr_stop()
- dr_dump()

The programmer uses *dr_start()* to order DAG Recorder to start its measurement, *dr_stop()* to stop DAG Recorder's measurement and *dr_dump()* to make DAG Recorder dump its measurement data currently stored in memory out to files. Besides, DAG Recorder also defines several environment variables for programmers to tweak its behaviors. DAG Recorder's usage is just that simple. We are going to describe DAG Recorder's three aspects of what kind of DAG that it records, how it does the measurement, and what kind of information it can provide.

1.2.1 DAG Structure

A computational DAG of a task parallel program consists of a set of nodes and a set of edges. Each node represents a sequential code segment in the application-level code that does not contain any task parallel primitive. Edges are the manifests of those task parallel primitives, they represent the task-parallelism-style relationships between nodes. These relationships are:

- parent-child (spawning) relationship
- continuation relationship
- synchronizing relationship

For example, two contiguous code segments separated by a Create_Task primitive in the application's code would have *continuation* relationship between them. The code segment preceding Create_Task and the first code segment of the function assigned to the task created by Create_Task would have *spawning* relationship. The last code segments of all tasks (or functions) synchronized by a Wait_Tasks primitive and the code segment preceding that Wait_Tasks primitive would bear a *synchronizing* relationship with the code segment following that Wait_Tasks primitive. There are possibly two or more nodes that have *synchronizing* edges pointing to a single node of the code segment following Wait_Tasks depending on how many tasks that Wait_Tasks synchronizes.

DAG Recorder classifies nodes into 3 node kinds based on the task parallel primitives that end their code segments. A code segment ended by Create_Task is represented by a *create* node. A code segment ended by Wait_Tasks is represented by a *wait* node. The last code segment of a task (function) is called an *end* node. Based on this naming, a small DAG of a *fib(3)* program is showed in Figure 1.3.

Actually the DAG's structure is hierarchical. Three kinds of *create*, *wait* and *end* are just terminal nodes that do not contain any sub-graph inside them. There are two other kinds of

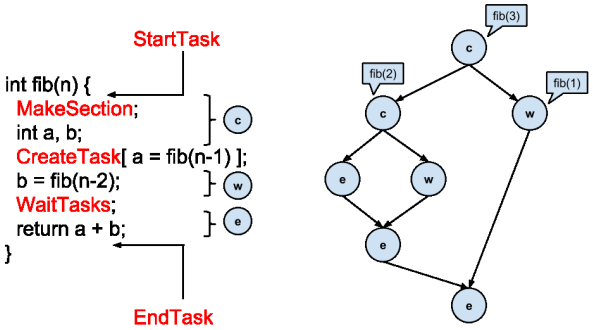


Figure 1.3. DAG of fib(3) execution

Figure 1.4. All node kinds of DAG Recorder (or PIDAG)

non-terminal nodes which are *section* and *task*. They are collective nodes containing sub-graphs of terminal or other collective nodes inside. All these 5 kinds of nodes are shown in Figure 1.4.

A *task* node corresponds to a task entity in the runtime system. It can contain none, one or multiple *section* nodes before ending by an *end* node. A *section* node contains one or multiple *create* and *section* nodes before ending by a *wait* node. The purpose of *section* kind is to mark all tasks that will get synchronized by a Wait_Tasks primitive. All tasks created by child *create* nodes in a *section* node are synchronized by the last child *wait* node of that *section* node. Figure 1.5 shows the member variables of a DAG Recorder's node's structure: *info* is a child structure that holds all performance information related to the code segment that the node represents, *next* is a pointer to the next node in the list of all child nodes of its parent, if the node is of *create* kind *child* pointer will point to the *task* node that it creates, if the node is a collective node of kind *task* or *section* *subgraphs* list will contain all its child nodes.

1.2.2 Methodology

In order to capture the DAG structure as described in the previous sub-section, DAG Recorder needs to instrument measurement code at following seven positions:

- *EnterCreateTask*: right before entering the Create_Task primitive
- *LeaveCreateTask*: right after leaving the Create_Task primitive
- *EnterWaitTasks*: right before entering the Wait_Tasks primitive
- *LeaveWaitTasks*: right after leaving the Wait_Tasks primitive
- *StartTask*: right before starting a new task
- *EndTask*: right before ending a task
- *MakeSection*: to mark the start of a new *section*

Let's consider the above items also as the code that need to be inserted at corresponding positions. DAG Recorder has modified the process through which a common API primitive is translated into a specific task parallel API so that it puts these code at appropriate positions. Inside Create_Task, DAG Recorder puts *EnterCreateTask* as near the upper code as possible, *LeaveCreateTask* as near the lower code as possible, *StartTask* right before executing the task function and *EndTask* right after the task function finishes. Inside Wait_Tasks,

```
1 struct dr_dag_node {
2   dr_dag_node_info info;
3   struct dr_dag_node * next;
4   union {
5     struct dr_dag_node * child;
6     dr_dag_node_list subgraphs[1];
7   }
8 }
```

Figure 1.5. DAG Recorder's node

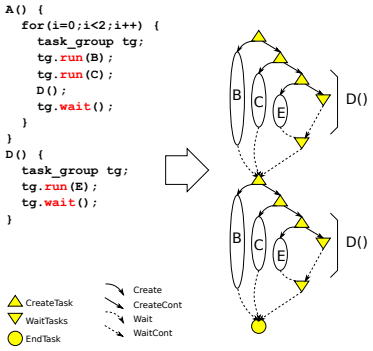


Figure 1.6. An example of task parallel computational DAG

DAG Recorder puts *EnterWaitTasks* as near the upper code as possible and *LeaveWaitTasks* as near the lower code as possible. Inside Make_Task_Group, DAG Recorder sets a flag to remember to create a new *section* to enclose all following nodes that is other than *end* node and a *wait* node would end that *section*.

StartTask, *LeaveCreateTask* and *LeaveWaitTasks* are positions where an interval (a node) begins, so DAG Recorder records necessary information such as file name, line number, time, worker number, cpu number in order to later combine with an end interval information to construct a full node. End interval information is recorded at *EndTask*, *EnterCreateTask*, and *EnterWaitTasks*.

Figure 1.6 demonstrates the collective *task* node kind and how a *section* marks effective *task*s to be synchronized by a *wait*. White nodes with one character inside are of *task* kind. The transition to a new *section* is indicated by the *task_group tg;* declaration.

DAG Recorder has an useful feature of on-the-fly contraction that it can contract uninteresting subgraphs into only one node. An uninteresting subgraph is a subgraph that was executed wholly on a single worker. There was no work stealing, task migration happenning in that subgraph. Such kind of subgraphs is not interesting from the perspective of task parallel performance analysis. Contracting them does not affect our performance analysis potential, and also helps reduce a considerable number of nodes in the DAG which is helpful for the visualizer that visualizes the DAG. When a *task* or a *section* is contracted, the statistical data of its child nodes are aggregated into its *dr_dag_node_info* structure (Figure 1.7). For a *section*, DAG Recorder also includes statistical data of all *task* nodes created by the *section*'s *create* nodes.

1.2.3 Capability

DAG Recorder observes all execution intervals of a task parallel program and manifests these intervals as nodes in the DAG. Each node of the DAG holds a *dr_dag_node_info* structure (Figure 1.7) that stores relevant performance data of the interval's execution. In short, currently

```
typedef struct dr_dag_node_info {
  dr_clock_pos start; /* start clock, start position (filename,line#), worker, cpu */
  dr_clock_pos end; /* end clock, end position (filename,line#), worker, cpu */
  dr_clock_t est; /* earliest start time */
  dr_clock_t t_1; /* work */
  dr_clock_t t_inf; /* critical path */
  dr_clock_t first_ready_t; /* time at which this node became ready */
  dr_clock_t last_start_t; /* time at which the last node started */
  dr_clock_t t_ready[dr_dag_edge_kind_max]; /* weighted sum of ready tasks */
  long logical_node_counts[dr_dag_node_kind_section]; /* including collapsed nodes */
  long logical_edge_counts[dr_dag_edge_kind_max]; /* including collapsed edges */
  long cur_node_count; /* actual node count excluding collapsed nodes */
  long min_node_count; /* min # of nodes if all collapsable nodes are collapsed */
  long n_child_create_tasks; /* # of direct children of create_task type */
  int worker; /* worker */
  int cpu; /* cpu */
  dr_dag_node_kind_t kind; /* kind of this node */
  dr_dag_edge_kind_t in_edge_kind; /* kind of edge from last node initiating it */
} dr_dag_node_info;
```

Figure 1.7. Node info structure

DAG Recorder records time metrics and source code positions. Source code position information is critical for tracing back to the responsible application-level code blocks that occur the time metrics quantities. The recording source code positions also shows the superiority of DAG Recorder's instrumentation approach. By instrumenting measurement code into application code, DAG Recorder can avoid the painful process of extracting application-level information from binary executables that is needed by the sampling measurement approach. In general, DAG Recorder can attributing time metrics back up to application-level code which is useful for programmers to analyze their application's performance.

The information that a node can provide about the performance of the interval it represents can be described shortly below:

- *start* and *end* positions (filename, line number) of the code segment.
- *est* (earliest start time): this is the start time of the node if it is single, or the start time of its head child node if it is collective.
- *t_1*: work time of the node or collective work time of its child nodes.
- *t_inf*: critical path of its subgraph or just equals *t_1* if the node is single.
- *first_ready_t*: time at which the last of its dependent nodes finishes, making it ready to execute.
- *last_start_t*: time at which the last of its child nodes gets started, or just equals *est* if the node is single.
- *worker*: the worker on which this node was executed on
- *cpu*: the core number on which this node was executed on

We are next going to discuss some useful information based on the DAG that DAG Recorder records such as work, critical path, time series of actual and available parallelism, steal history.

Work

Based on the DAG we can know when a worker is working on the application's code and when it is not. The time that a worker is working on application code is considered as **work time** or simply **work**. The time that it executes other code such as runtime system's code and the time it is idle are not work time. This non-work execution time of a worker can be classified into two

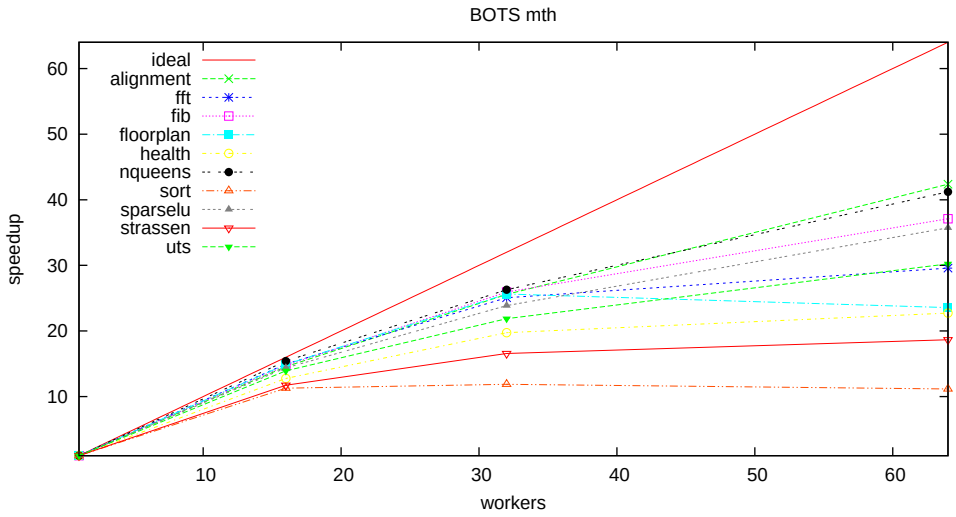


Figure 1.8. BOTS's scalability

categories of **delay** and **nowork** (idleness). By subtracting the end and start time of a node, we can calculate its work, and summarizing them of all nodes in a DAG would give us the total work of the execution. The time a worker moves from a *EnterCreateTask* instrumentation point to its consecutive *StartTask* point (for work-first based scheduler) is considered as **delay** time. It is also **delay** for the intervals between *EnterCreateTask* and *LeaveCreateTask* (for help-first based scheduler). The time that a worker is not executing anything is **nowork**.

The total elapsed time of all workers in an execution equals to the multiplication of the execution time and the number of processor cores on which the execution occurred. We refer to this kind of time quantity as **worker time** with the meaning that it is the time an application uses workers. This is analogous to the definition of "CPU time" which is commonly defined the time that a process uses computer CPU.

Based on the DAG, we can break down worker time of an execution into 3 categories of **work**, **delay** and **nowork**.

We have run DAG Recorder with all ten applications in the Barcelona OpenMP Task Suite (BOTS) benchmark suite [12] and aquired the DAG of every execution. Experiment environment and parameters for each benchmark are described in Chapter 4's Table 4.1 and Table 4.3.

Based on simple execution times of the experiments we can draw a scalability graph of all *MassiveThreads* model based applications in Figure 1.8.

The experiments were conducted on the same machine with the same compiler. Thus, there are still only 3 experiment parameters varying, they are the application (*app*), the number of cores (*ppn*) and the task parallel model (*type*) in use. A combination of specific *app-ppn-type* indicates a single experiment execution (or a single DAG) of the *type* model-based *app* application on *ppn* cores. In Figure 1.8, the *type* parameter remains the same as *MassiveThreads*, only *app* and *ppn* vary in the graph. On the other hand, in Figure 1.9 the *app* parameter remains the same as Sort, *ppn* and *type* vary. One more different point is that Figure 1.8 plots the speedup values which are the ratios of the execution time of $ppn = 1$ over other *ppn*s, while as Figure 1.9 plots the worker times of the executions.

By fixing *ppn* parameter, the graph in Figure 1.9 can get split into 4 graphs in Figure 1.10. Moreover, in Figure 1.10 each bar which represents an execution is divided to 3 parts of 3 colors corresponding to **work**, **delay** and **nowork** components of the execution's worker time.

On the other hand, by fixing *type* parameter, bars in Figure 1.9 gets rearranged and composes

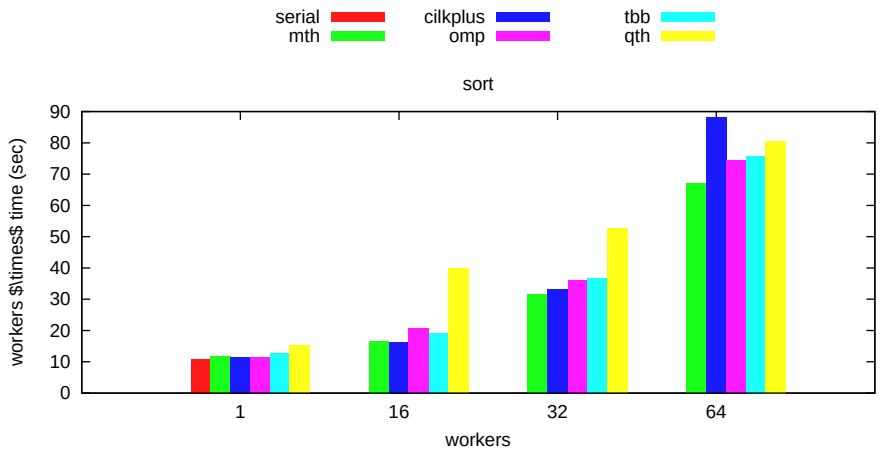


Figure 1.9. Compare six task parallel systems running Sort

4 graphs corresponding to 4 task parallel systems in Figure 1.11.

Critical path

DAG Recorder can provide information about the critical path of a single node, a collective node with subgraph or the whole DAG. The critical path of a single node is equal to the work of that node:

$$info.t\_inf = info.t\_1 = info.end.t - info.start.t$$

For a collective node, its *t_inf* is accumulated as the longest path in its subgraph. This can be calculated straightforwardly based on the structure of the subgraph and that *t_inf* of all child nodes have been accumulated. The critical path of a whole DAG is stored in *t_inf* of its original *task* node.

Time series of parallelism

Based on the start time (*start.t*) and end time (*end.t*) of every node, together with the time it became ready for execution (*first_ready_t*), we can calculate the time series of available and actual parallelism of a DAG. A node contributes one point to avaiable parallelism during its ready period from *first_ready_t* to *start.t*, and one point to actual parallelism during its execution time from *start.t* to *end.t*.

For a collective node, its available parallelism is calculated by following formula

$$\frac{t\_ready}{last\_start\_t - first\_ready\_t}$$

in which *t_ready* is the total time that its child nodes spends in ready state, *first_ready_t* is the earliest time that one of its child nodes becomes ready, and *last_start_t* is the latest time that one of its child nodes starts. Output these time series of parallelism into file and have it drawn by Gnuplot, we can have result graph like Figure 1.13 which is called parallelism profile graph of an execution of a task parallel program.

Steal history

Because DAG Recorder annotated every execution interval (node) with the worker that it was on, the history of task migrations between workers can be easily computed.

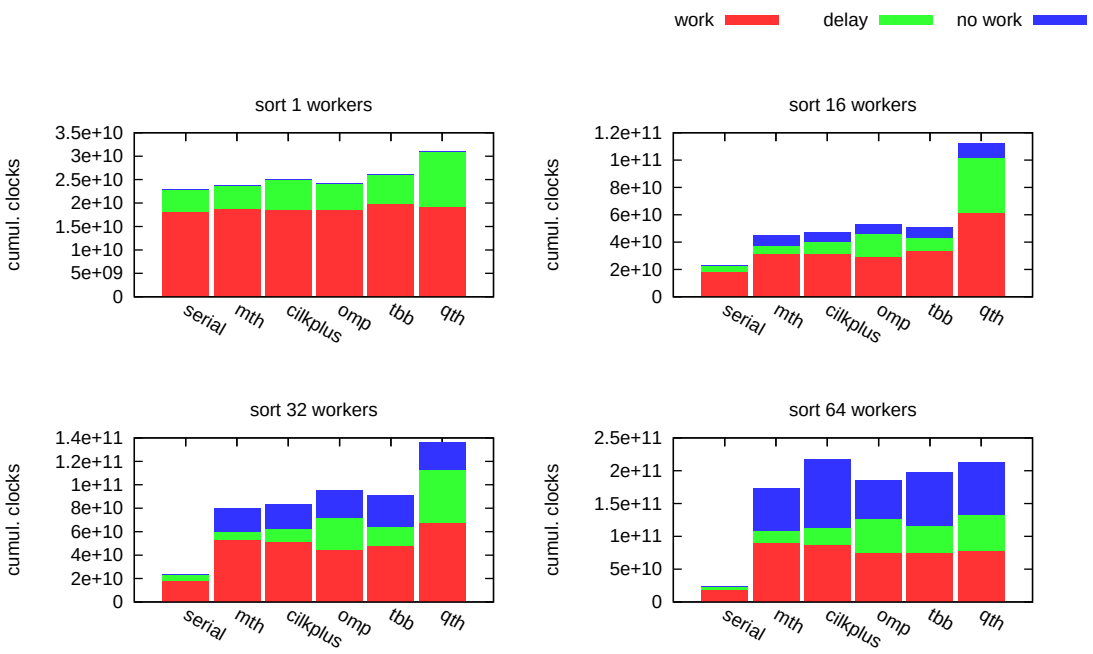


Figure 1.10. Breakdown graphs of Sort on 1, 16, 32, 64 core(s)

1.3 Motivation

DAG Recorder is useful in doing statistical analyses. Using the data it collected some interesting breakdown graphs can be produced like in Figure 1.10 and Figure 1.11. If we consider three quantities of **work**, **delay** and **nowork** as a metrics to measure the difference between task parallel systems, through Figure 1.10 we can understand that these systems perform pretty much differently. *QThreads* model has relatively large **work** and **delay** compared with other systems on 16 and 32 cores, but on 64 cores, it becomes normal. On the same number of cores, the amounts of three quantities vary on different systems. *MassiveThreads* exibits the best performance in these graphs. We would ofcourse like to reason specific causes of the breakdown quantity variation and *QThreads*'s notably bad performance, but these statistical analyses is not enough to do that.

In Figure 1.11, along with the increase in number of cores, all three quantities inflate significantly. The increase in **nowork** may be reasonable because increasing number of cores while keeping the parallelism algorithm at the same just makes the idle state of workers worse, hence **nowork** gets worse. But the inflations of **work** and **delay** are not that trivial to understand but rather sometimes seem to be very mysterious.

In these breakdown graphs, if all bars have the same height the performance is perfect. However, heights of these bars vary among systems (*type*) and rise along with high core counts (*ppn*) in fact. We name the surpassed part of **work** on high core count compared with that on one core (serial execution) as "work stretch". To say in another way, if those components of **work stretch**, **delay** and **nowork** disappear, we would have perfect performance. Therefore, the performance loss of a task parallel execution can be attributed to the 3 factors of **work stretch**, **delay** and **nowork**. To analyze the underlying causes of these 3 factors is the motivation of our work.

Figure 1.12 and Figure 1.13 gather a compact set of statistical graphs for Sort application. Figure 1.12a is the speedup graph of Sort run by *MassiveThreads*. It has fixed *app* and *type* parameters and *ppn* varying. Figure 1.12b has the same set of experiment parameters (*app*=sort, *type*=*MassiveThreads*, *ppn* varies) with the speedup graph (Figure 1.12a) but the quantity

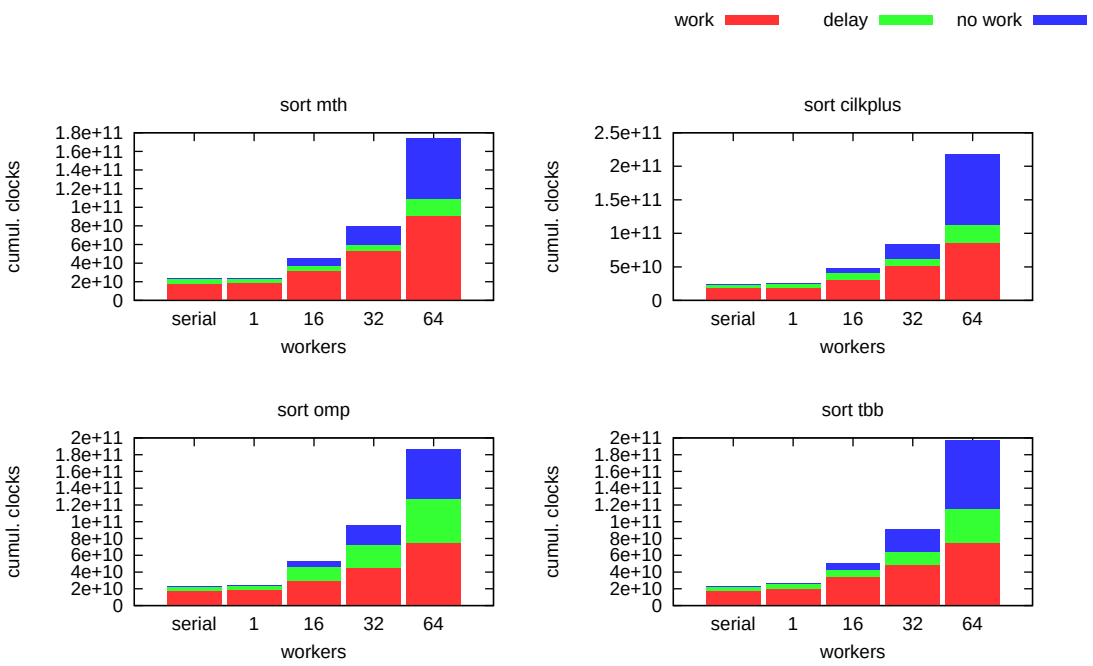


Figure 1.11. Breakdown graphs of Sort based on *MassiveThreads*, *Intel CilkPlus*, *OpenMP*, *Intel TBB*

to be plotted is different, and it also has breakdown information which is not shown in the Figure 1.12a. Figure 1.12c fixes *app* (*MassiveThreads*) and *ppn* (64) and lets *ppn* vary, hence it shows the differences between task parallel systems when executing Sort application based on the **work-delay-nowork** metrics.

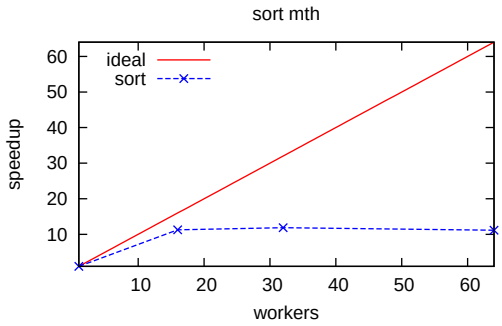
Figure 1.13 gives a closer look into a single execution of Sort application on 64 cores by *MassiveThreads* (all three parameters *app*, *ppn*, *type* fix). Parallelism profile graph expresses the actual parallelism and available parallelism of an application in the course of its execution as the x-axis represents time flow and the y-axis represents parallelism degree. The actual parallelism is the number of running tasks or working workers at a point of time. It is manifested by the red color area in the graph. Apparently actual parallelism never surpasses the number of cores on which the program is executed. The available parallelism is divided further into several kinds based on the kind of *waiting* the ready node is waiting on. Available parallelism is represented by areas of other colors on the graph. Blue *create* kind indicates *task* nodes that have been newly created but not start executing yet. Pink *create cont* kind indicates nodes that follow a CREATE_TASK primitive and are waiting for execution. Green *end* kind indicates *task* nodes that have finished execution but not synchronized yet. Cyan *wait cont* kind indicates nodes that follow a WAIT_TASKS primitive and are waiting for tasks that they synchronize to be finished.

According to Figure 1.13, it is understood that the large **nowork** factor origins from the lack of parallelism in the latter half of the execution. In its latter half, Sort was doing its merging phase merging sub-arrays that have been sorted. The parallelization of this merging phase has not been done well enough to exhibit sufficient parallelism for 64 cores. We need a tool to help us look closer into what the worker are doing during this low parallelism period so that we can know which parts in the application code to revise.

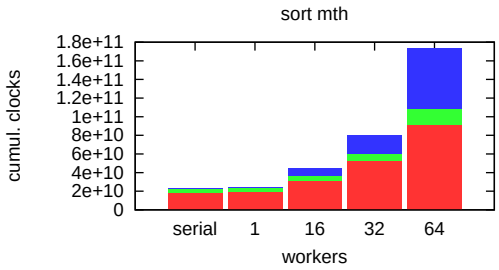
Figure 1.14 and Figure 1.15 introduce the same set of statistical graphs for Strassen application. The two factors **work** and **delay** rise along with high core counts too. Their amounts also vary among task parallel systems tool. Figure 1.15 shows the parallelism profile of Strassen running on 64 cores by *MassiveThreads*. In the first half of the execution its actual parallelism is very low as only one. We need another tool that provides a closer look into this period to see

what the worker is doing and where in the application code it is executing.

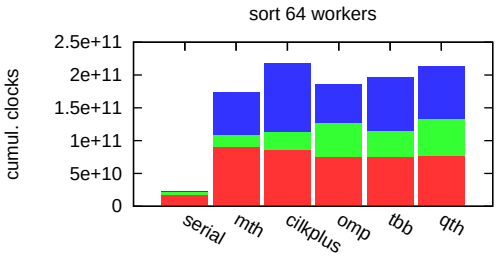
DAGViz is a tool like that. DAGViz visualizes the DAG and provides interaction functionalities to allow the user to explore the DAG visually.



(a) Speedup graph



(b) Breakdown graph



(c) Breakdown graph on 64 cores

Figure 1.12. Sort's scalability and breakdown graphs

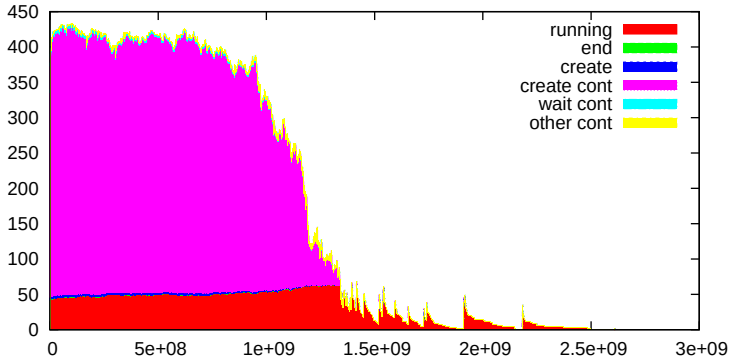
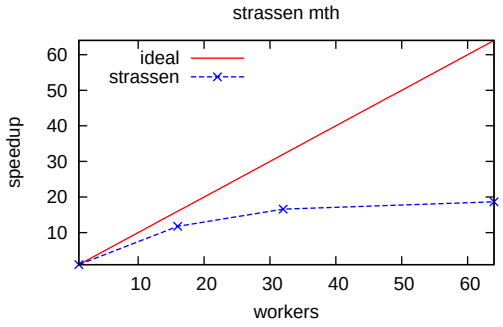
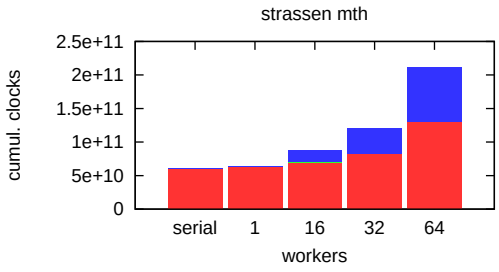


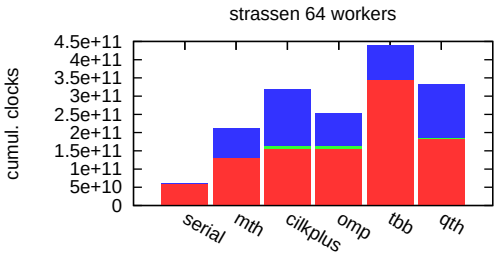
Figure 1.13. Sort's parallelism profile at 64-core execution by *MassiveThreads*



(a) Speedup graph



(b) Breakdown graph



(c) Breakdown graph on 64 cores

Figure 1.14. Strassen's scalability and breakdown graphs

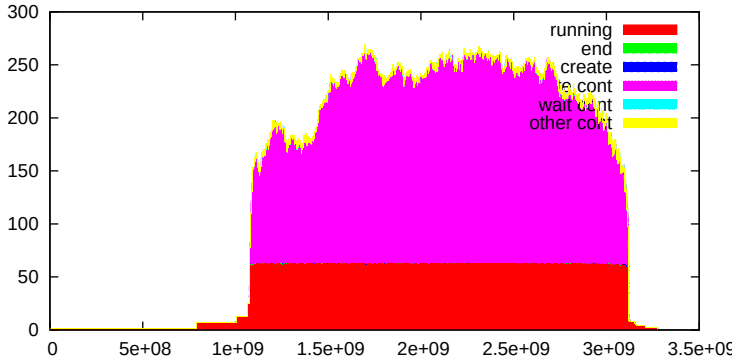


Figure 1.15. Strassen's parallelism profile at 64-core execution by *MassiveThreads*

1.4 Organization of this Thesis

The structure of this paper is organized as following: the next chapter discusses related work, the third chapter describes DAGViz's design and implementaion, the fourth chapter talks about using DAGViz in case studies of BOTS's applications. Chapter 5 describes a preliminary implementation of sampling-based measurement method. Chapter 6 discusses about evaluations of DAG Recorder and DAGViz. it describes what the profiler can do and show. Finally, chapter 7 is conclusions and future work.

Chapter 2

Related Work

2.1 Parallel Performance Analysis

Tallent et al. [13] categorized parallel execution time of a multithreaded program into 3 kinds of *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. [14] had taken a step further than [13] 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 Barcelona OpenMP Task Suite (BOTS). They demonstrated that work inflation accounted for a dominant part and proposed a locality-aware scheduler which could mitigate this factor.

There have been many tools for analyzing parallel performance. The TAU performance system [15] is an open source system that has a powerful automatic instrumentation toolset. Intel VTune Amplifier software [16] 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. [17] 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 must be more complicated.

The Cilkview Scalability Analyzer [18] describes Cilkview tool which monitors logical parallelism during an instrumented execution of the Cilk++ application on a single processor core, then analyzes logical dependencies between tasks to predict the application's performance on a machine with more cores.

2.2 Visualizations for Analyzing Performance and Graphical Tools

Visualization is an highly useful tool in doing analysis. Visual elements can convey structure of the problem at a glance, and they may ignite insights to the solution that numbers and tables merely can hardly reveal. By sticking to the analysis mindset of "overview first, zoom and filter, the details on demand" [19], a visualization tool can support effectively the analysis of complex hierarchical large datasets.

Visualization has been used as an effective tool to deal with various specific performance problems. Knowing that communication cost in massively parallel applications on large distributed systems impacts heavily their performance, the authors in [20] have combined 2D and 3D views to visualize network traffic in order to explain and then optimize the performance of large-scale applications on a supercomputer. *CommGram* [21] invented a new kind of visualization to display network traffic data. It enhances bipartite graph style by replacing thin straight arrows by fat colorful brushy curves to represent data flow between communication nodes vividly.

Vampir Vampir [22] translates a trace file of an MPI program into a variety of graphical visualizations. Its main visualization is a timeline view (Gantt chart) of the execution of the parallel program. It simultaneously provides a statistical view that displays aggregate information of a chosen time interval. It can also provide system activities at a particular point of time. Iwainsky et al. [23] have used Vampir to visulize remote socket traffic on the Intel Nehalem-EX.

Jumpshot Jumpshot [24] is a scalable tool to visualize timelines. Task intervals of all workers written in file in *sslog* log file format can be converted into slog2 format by a converting program written by Prof. Taura. *slog2* format can be read and visualized by Jumpshot. Jumpshot is really a scalable tool that can zoom into tiny intervals but it is not that easy and quick for users to perform zooming in/zooming out operations. One restriction of Jumpshot is that it can only display up to 10 different categories which have different colors. It means that, for example, the visualization can distinguish up to only 10 different task levels.

Paje Paje [25] provides timeline style visualization of parallel programs executing on multiple nodes each of which contains dynamically running multiple threads. Paje supports *click-back*, *click-forward* interaction semantics which mean that clicking visualization to show source code and clicking source code to show visualization. Paje has several filtering and zooming functionalities to help programmers to cope with large amount of trace information. These filterings give users simplified abstract view of the data (statistical graphs showing aggregate information of a chosen time slice). Users of Paje can also modify mapping between trace information entities and visual elements (arrows, boxes, triangles) which makes the visualization flexible.

Jedule Jedule [26] is a tool to visualize schedules of parallel applications in timeline style. It is built on Java. Users can adjust color style of Jedule's visualization, can zoom in by selecting a rectangular box, can export current view to images. Authors in [14] have used Jedule to visualize a timeline view for analyzing the locality of a scheduling policy.

ThreadScope Wheeler and Thain [27] in their work 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 [28]. GraphViz tool is scalable up to only hundreds of nodes and very slow with

large graphs of more than a thousand nodes because its algorithm [29] focuses on the aesthetic aspect of graphs rather than rendering speed. And most of all, GraphViz is not interactive.

Aftermath Aftermath [30] is a graphical tool that visualize traces of an OpenStream [31] 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. Aftermath is also built upon the GTK+ GUI toolkit [32] and Cairo graphics rendering library [33] like our work DAGViz.

Chapter 3

DAG Visualizer

When *dr_dump()* is called, DAG Recorder flattens the DAG which is stored hierarchically in memory and write to file. DAGViz then reads the flattened DAG from file, reconstruct its hierarchical structure but in a different way which is for the favor of a graphical representation. DAGViz lays out the DAG in memory by calculating and assigning coordinates to its nodes and then draws these nodes along with edges connecting them on screen. In this chapter, we will describe the internal data structure of DAGViz, its layout algorithms, rendering algorithms, animation mechanismsand other interesting aspects in design and implementation of DAGViz.

3.1 Internal Data Structure

A flattened DAG in file is called PIDAG (position-independent DAG) or simply **P** which is defined as *dv_pidag_t* structure in DAGViz. Because one PIDAG may be very large by holding millions of nodes and edges, it is not efficient to read the whole PIDAG into (physical) memory. Or sometimes it's even impossible to read it all at once when PIDAG's size is up to gigabytes exceeding memory's capacity. Reading via a sream of the file every time we need to get data from PIDAG is not efficient either. A better approach is that we map PIDAG into virtual memory space by *mmap()* function, then needed parts of PIDAG will later get loaded into physical memory automatically by hardware mechanisms when DAGViz accesses them. This mapping approach is especially fit with arbitrary data-accessing pattern of DAGViz when users tend to travel around the DAG toward interesting parts undeterminedly.

A **P** provides exactly information that DAG Recorder provides without anything related to graphics rendering. So another structure is needed to hold the laid-out DAG which can be rendered on screen. We call it DAG (from the perspective of DAGViz) or simply **D**. **D** is defined in DAGViz by *dv_dag_t* structure. **D** is a collection of DAGViz's nodes (as distinguished with DAG Recorder's nodes and PIDAG's nodes) each of which holds coordinate information necessary to render itself to screen. One node in **D** is associated with a node in **P** which is the content source of the **D**'s node. So a **D**'s node carries a reference to a **P**'s node. **D** is like a renderable version of **P**. **D** is the skeleton frame and **P** is the content. A **D** should have had the same number of nodes corresponding to the number of nodes existing in **P** but due to the constraint of memory capacity and also because displaying excessive number of nodes on screen could make them unseeable or make users confused, we limit the size of **D** to a fixed-size pool of nodes. Thus, a **D** does not reflect the whole **P** but only a part of it. In favor of recycling unnecessary **D**'s nodes for newly accessed nodes when the pool gets empty, DAGViz will purge unnecessary nodes in **D** and return them to the pool at runtime based on what need to be displayed on screen as the user navigates/interacts on the GUI. Adding this node pool mechanism provides us with a better control over the memory footprint of DAGViz.

A **D** mainly manages the DAG's node pool, the DAG's structure and interfaces with the contents residing in **P**. **D** is not what the user can see physically. What the user see and

conceive visually of a DAG is called *view*s of the DAG. The way a user views a DAG can be different based on different kinds of layouts that can be applied to the same DAG. Each *view* can be considered as the result of applying a layout algorithm on a DAG to materialize it to an actual form that the user can see. Therefore, we added another data structure *dv_view_t* (**V**) to represent this *view* notion. Beside layout type information, a **V** also contains rendering parameters which specify the user's preferences in drawing a DAG, and interaction parameters which stores the interaction activities that the user conducts on a DAG through that *view*.

There can be multiple *view*s based on the same DAG, as well as there can be multiple **V**s referencing the same **D**. DAGViz also supports multiple **D**s for the same **P** too, each **D** explore different parts of **P** independently. This kind of relationship between **P**, **D** and **V** are illustrated in Figure 3.1.

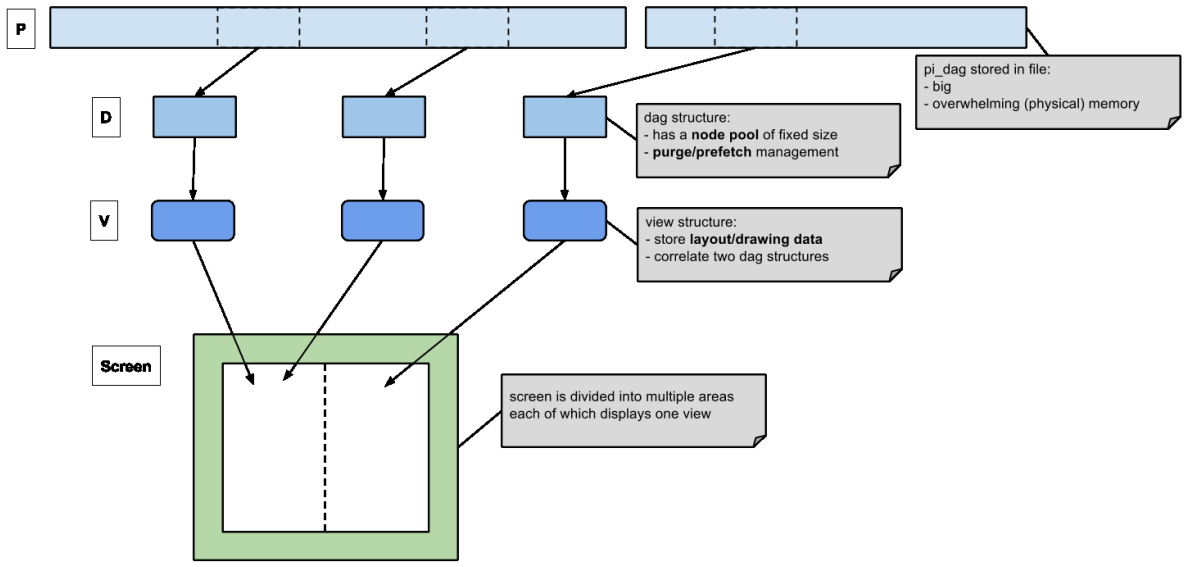


Figure 3.1. P-D-V design

In the following paragraphs, I'm going to describe detailed structures of **P**, **D**, **V**.

Structure of P **P**'s definition is shown in Figure A.1. Beside number of nodes, number of edges, the start clock of the execution, number of workers, a PIDAG holds a contiguous array of PIDAG's nodes (Figure 3.3) which reference to the same *dr_dag_node_info* structures with DAG Recorder's nodes but have different linking structure between themselves as they are flattened-down version of DAG Recorder's DAG. PIDAG's nodes also have 5 kinds, they are *create*, *wait*, *end*, *section* and *task* (same as DAG Recorder's node kinds) (Figure 1.4). *section* and *task* are collective nodes which contain subgraphs inside them. All (direct) child nodes of a collective node are arranged contiguously in a sub-array somewhere behind the node in the array. So a *section* or a *task* would additionally hold two offset indexes (*subgraphs_begin_offset*, *subgraphs_end_offset*) pointing to the beginning and the end of the sub-array where their child nodes reside. A *create* node would additionally hold one offset index pointing to the *task* node that it creates. Figure 3.2 illustrates how nodes of a DAG of *fib*(3) program would be arranged in PIDAG. As we can see from the figure, child nodes of a *task* reside right behind it. Child nodes of a *section* are put after the sub-array that that *section* belongs to. *task* node created by a *create* node is inserted after the sub-array that contains that *drcreate* node.

Structure of D **D** structure's definition is shown in Figure A.2. First of all, a **D** structure holds a pointer to the **P** that it associates with. Then, the most important part of a **D** is its

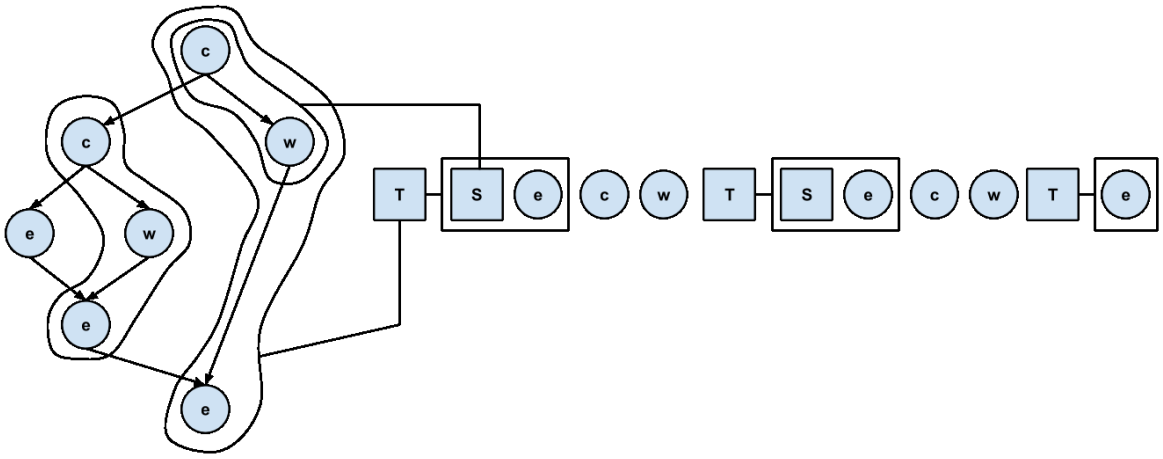


Figure 3.2. Flattened DAG Recorder's nodes in PIDAG of *fib(3) program*

```
1  struct dr_pi_dag_node {
2    dr_dag_node_info info;
3    long edges_begin; /* index of begining of edges from this node */
4    long edges_end; /* index of end of edges from this node */
5    union {
6      /* for create node */
7      long child_offset; /* offset to its created task */
8      /* for section or task node */
9      struct {
10       long subgraphs_begin_offset; /* offset to beginning of subgraphs */
11       long subgraphs_end_offset; /* offset to end of subgraphs */
12     };
13   };
14 };
```

Figure 3.3. PIDAG's node

node pool. The pool is a fixed-size array of DAGViz's nodes (Figure A.4). It consists of *T*, *To*, *Tsz*, *Tn* member variables of the **D**. Its size stored in *Tsz* is currently set at one hundred thoundsand. *To* (*T* occupied) is used to indicate if a node in the array *T* has been allocated for the DAG. So *To* has the same number of elements as *T* does. Number of currently allocated nodes of *T* is stored in *Tn*. The allocation and releasing mechanisms of node pool are provided through following interfaces:

- *dv_dag_node_pool_init()*: initialize node pool's variables to default values, allocate memory for T array and To array.
- *dv_dag_node_pool_is_empty()*: to check if the node pool is empty (occupied fully) or not by comparing Tn with Tsz.
- *dv_dag_node_pool_avail()*: return the current number of available nodes in the pool (Tsz − Tn).
- *dv_dag_node_pool_pop()*: return one pointer to an available node in the pool, set the flag of that node as occupied in To. If the pool is empty, call the function *dv_dag_clear_shrinked_nodes()* to ask for **D** to release children of shrinked nodes in the DAG which are not currently visible.
- *dv_dag_node_pool_push()*: used to return a node to the pool by reseting its occupied flag in To, and incrementing Tn.

- *dv_dag_node_pool_pop_contiguous()*: used to pop a contiguous sub-array of nodes in T. The purpose of this function is that if all child nodes of a *section* or a *task* in **P** are assigned contiguous **D**'s nodes, it makes it easier for **D** to later locate all child nodes of a particular node and release them to the pool when necessary.

Using these interfaces, **D** can acquire nodes from and return nodes to the pool in order to build its DAG in memory. Reversely, **D** also provides an interface, *dv_dag_clear_shrinked_nodes()*, for the pool to initiatively ask **D** to return unnecessary nodes for it to supply for new allocation requests when it has run out of available nodes. The mechanism in *dv_dag_clear_shrinked_nodes()* will be discussed in Section 3.2 when DAG's structure based on the linking between **D**'s nodes are described. Beside the node pool, a **D** also stores other parameters specifying its relative position inside **P** such as its current depth (*cur_d*), its current depth including nodes that are extensible to but hidden (not visible) on screen (*cur_d_ex*).

Structure of V A **V** is associated with one **D**. **V**'s full definition is shown in Figure A.3. Its first member variable is a pointer to a **D**. **V** holds *viewing* parameters for the **D** that it is associated with. These *viewing* parameters can be classified into 3 categories of appearance, drawing and interaction parameters. Appearance parameters are:

- *lt*: layout type of the DAG
- *et*: edge type specifying how to draw edges
- *edge_affix*: indicates if an affix segment of edge should be drawn at the contact of an edge and a node
- *nc*: node color mode, indicate what to be represented by colors (worker, cpu, node kind or source code location)

Drawing parameters are:

- *vpw, vph*: width and height of the main viewport that this **V** is displayed on. The notion of viewport will be discussed in Section 3.6.1.
- *x, y*: position of the coordinate origin of the view in viewport, is changed when user pan the DAG around screen.
- *zoom_ratio_x, zoom_ratio_y*: zoom ratios for cairo to magnify/shrink graphics horizontally and vertically
- *nd*: number of nodes drawn on screen
- *ndh*: number of nodes drawn on screen plus number of collective nodes not drawn on screen but needed to traverse through when drawing DAG

Interaction parameters are:

- *focused*: indicate if the view is focused or not so that hot keys would be effective to it
- *cm*: clicking mode, indicate what to do when user clicks a node
- *drag_on*: indicate if a dragging operation is being conducted or not
- *pressx, pressy*: position where the user presses (clicking down)
- *accdisx, accdisy*: accumulated moving distance since the user pressed
- *do_zoom_x, do_zoom_y*: indicate to make cairo do zooming when the user scroll on the view
- *do_scale_radix, do_scale_radius*: change the radix and/or radius then make DAGViz re-layout and re-draw the DAG. This results in magnifying/shrinking the DAG by re-laying out rather than the automatic cairo's zooming.

```
typedef struct dv_dag_node {
  ...
  /* linking structure */
  struct dv_dag_node * parent;
  struct dv_dag_node * pre;
  dv_llist_t links[1];
  struct dv_dag_node * head;
  dv_llist_t tails[1];
  ...
} dv_node_coordinate_t;
```

Figure 3.4. Node's linking variables

Common linked list data structure DAGViz implements a common linked list data structure (*dv_llist_t*). This list structure provides such operations as add operation to add a new item to the end of the list, pop operation to pop the first item out of the list, get operation to get (but not remove) any item on the list using its index. The list element can store any kind of pointer (void *) as its item. For example, this list data structure is being used to hold the list of **P**'s nodes that have info tags to draw (**P**→itl), the list of **D**'s nodes having info tag (**D**→itl), the list of DAGViz's nodes that are in the middle of collapse/expand animation.

Automatic view coordination The fact that multiple **V**s can reference the same **D** makes these **V**s coordinated automatically because any change that the user makes to the **D** through any **V** would propagate to other **V**s too. The change can be some changing to the properties that **D** structure has. It is the same that multiple **D**s referencing the same **P** would be coordinated on the properties that **P** structure holds too. Moreover, all **V**s that reference different **D**s but their **D**s reference the same **P** would be coordinated on the properties that **P** structure holds.

3.2 DAG Structure and Hierarchical Traversal Model

In this section, we will discuss about how DAG structure is constructed based on linking between **D**'s nodes. DAGViz considers "child nodes" notion of a *section* wider than DAG Recorder. Child node group of a *section* consists of not only *create*, *section*, *wait* nodes as DAG Recorder does but also *task* nodes which child *create* nodes in the group create. Child nodes of a *task* still consist of only *section* and *end*.

One node will hold five pointers or lists of pointers referencing to some other nodes related to it: *parent*, *pre*, *links*, *head* and *tails* in which *links* and *tails* are lists of nodes (Figure 3.4. These five variables can be described as below:

- parent: its parent node
- pre: the node right before it in the same group of child nodes of its parent. *pre* of a *task* and *continuation* node is the *create* node that created the *task*.
- links: is a list of nodes that it links to. A node links to its next node in the same group of child nodes. A *create* node links to the *task* that it created and its *continuation* node.
- head: the head one (or the first one) of its child nodes.
- tails: the last one of its child nodes and also all *task* nodes that its child *create* nodes create.

Because the DAG is a hierarchical structure with many levels of layers stacked upon each other, we can classify these variables as such that *parent* points to the higher layer, *pre* and *links* point to nodes next to it in the same layer, *head* and *tails* point to child nodes in lower layer (Figure 3.5).

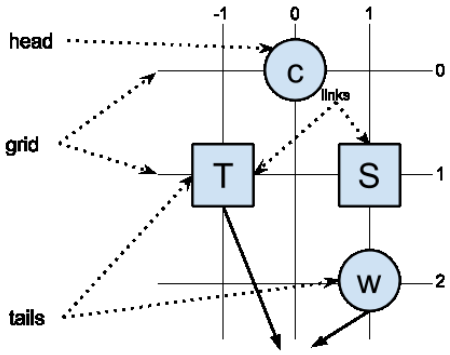


Figure 3.5. Hierarchical layout model

The "linked" relationship between nodes in DAGViz is a superset of the "contiguous" relationship between code segments. It is true that two nodes of two contiguous code segments are linked together but the reverse is not always true. For example, a *create* node and a *task* node are linked but they are not contiguous in the application code.

A DAG begins with a single *task* node representing the whole original application. Beginning from this root *task* we can traverse all nodes of the DAG recursively. A simplified version of this traversal model is shown in Figure 3.6a. At each node, after processing itself, it calls recursively its head node and then calls each of its linked (successor) nodes.

However, in some cases that simple traversal pattern is not enough because nodes sometimes require to be processed before they are traversed, or sometimes some of their processing need to be computed only after their inner subgraphs have been processed or after all their linked successors have been processed. In those cases, a more complex traversal model generalized in Figure 3.6b is needed.

At first, only the original *task* node is accessed in PIDAG (for it to be loaded into physical), assigned a node in **D**'s node pool and rendered on the screen. This node and other child nodes later would get accessed, loaded and rendered based on the user's demand. So state transition of a node can be systemized like this:

$$none \rightarrow set \rightarrow inner\_loaded$$

When a node has just been allocated from the pool and initialized, it has state *none*. When the function *dv_dag_node_set()* is called upon it, DAGViz would access its corresponding **P**'s node in PIDAG, causing it to be loaded into physical memory if it is not yet, and evaluate if it has any child node. If it has child node(s), it is considered a "union" node that can be expanded further into subgraphs. Even a or node which has been collapsed by DAG Recorder is not of "union" kind. A "union" node can move to the next state of *inner_loaded* which indicates that all its child nodes have been accessed and loaded into **D**. A node in *inner_loaded* state is switched between *shrinked* and *expanded* states which specifies if its inner subgraph should be drawn on screen or "collapsed" (this is visual collapse conducted by DAGViz, not the physical collapse done by DAG Recorder).

$$shrinked \leftrightarrow expanded$$

In summary, a **D**'s node would hold following flags to specify its possible state transitions above, in which two flags of *expanding* and *shrinking* are used for the collapse/expand animation mechanism (Section 3.5).

- *set*: 0 means *none* state and 1 means *set* state

```
int traverse(dv_dag_node_t * node) {
  /* Process individual */
  visit(node);
  /* Traverse inward */
  if (node->head) {
    traverse(node->head);
  }
  /* Traverse link-along */
  for (next in node->links) {
    traverse(next);
  }
}
```

(a) simple

```
int traverse(dv_dag_node_t * node) {
  /* Traverse inward */
  if (node->head) {
    /* Process head */
    ...
    /* Traverse */
    traverse(node->head);
    /* Process node with inward */
    ...
  } else {
    /* Process node without inward */
    ...
  }
  /* Traverse link-along */
  switch (node->links.size()) {
  case 0:
    /* Process node without link-along */
    ...
    break;
  case 1:
    /* Process one next */
    ...
    /* Traverse */
    traverse(next);
    /* Process node with one link-along */
    ...
    break;
  case 2:
    /* Process two nexts */
    ...
    /* Traverse */
    traverse(right_next);
    traverse(left_next);
    /* Process node with two link-alongs */
    ...
  }
}
```

(b) complex

Figure 3.6. DAG traversal model

```
1 typedef struct dv_node_coordinate {
2   /* Process individual */
3   double x, y;
4   double xpre, xp;
5   double lw, rw, dw;
6   double link_lw, link_rw, link_dw;
7 } dv_node_coordinate_t;
```

Figure 3.7. Node's coordinate variables

- *union*: specifies if this node is of *union* kind or not
- *inner_loaded*: 1 if this node is at *inner_loaded* state, its child nodes have been loaded to **D**, ready to be rendered on screen
- *shrinked*: 0 means *expanded* state, and 1 means *shrinked* state
- *expanding*: indicates this node is in transition from state *shrinked* to state *expanded*
- *shrinking*: indicates this node is in transition from state *expanded* to state *shrinked*

The purpose of the state *set* is to control the access to **P**'s nodes because every access to it would possibly cause the hardware system to fetch a page to physical memory. *union* kind is the notion of terminal and collective nodes from DAGViz's point of view. The *inner_loaded* state is for the sake of **D**'s node pool. If the pool gets empty, DAGViz would conduct a cleaning process to purge inner subgraphs of an unnecessary *union* node, return it back to *inner_loaded*=0. This cleaning process is done by the function *dv_dag_clear_shrinked_nodes()*.

Based on interactions with the GUI, the user can order DAGViz to load inner subgraph(s) and expand one or all leaf nodes which are of *union* kind in the current DAG. The user can also order DAGViz to collapse (visually) unnecessary nodes so that he can get a cleaner view on screen optionally.

3.3 Layout Algorithms and Views

A node has coordinate variables shown in Figure 3.7 in which *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 expanded child nodes. *link_lw*, *link_rw* and *link_dw* describes the bounding box covering its self, its subgraph, all successor nodes reached when traversing along the links, and their subgraphs too.

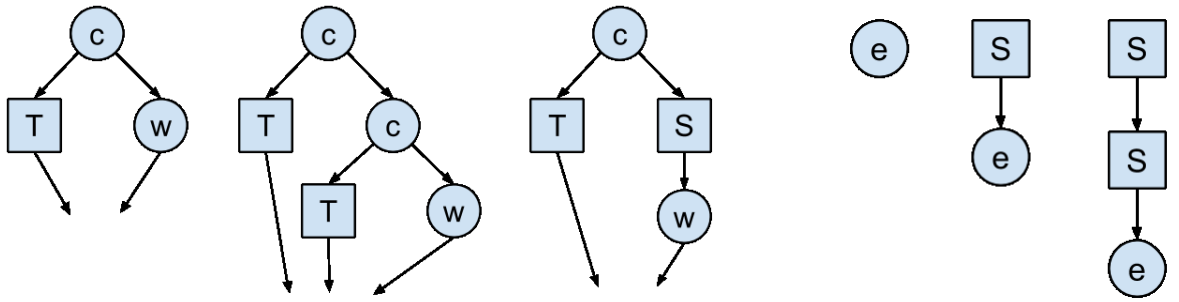
A layout algorithm traverses the DAG (with the described hierarchical traversal model) and sets values to these varibales of each node.

Basic topologies that a *task* can expand to are shown in Figure 3.8b. Basic topologies of a *section* are shown in Figure 3.8a.

We currently have implemented four layout algorithms which produce four kinds of views. They are DAG with round nodes, DAG with long nodes, timelines and parallelism histogram. The appearance of these four kinds of views applied to Sort application are shown as a demonstration in Figure 3.9.

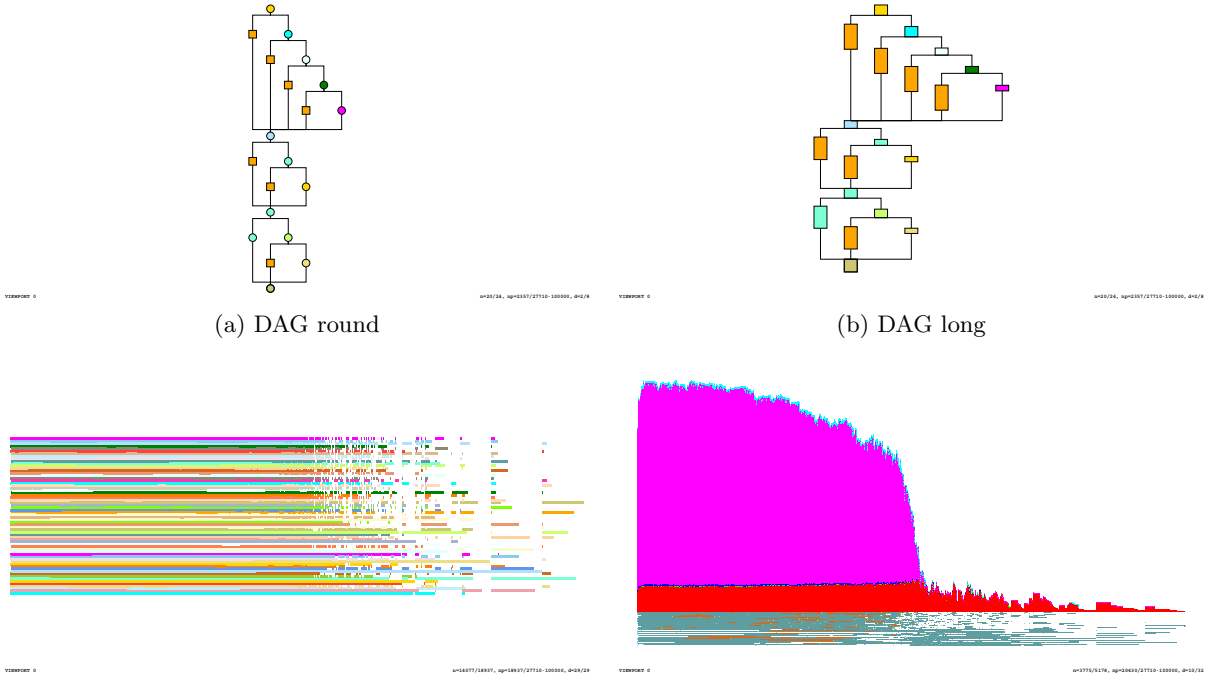
3.3.1 DAG View with Round Nodes

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 (Figure B.1). In the second phase, it sets values for xp and x of every node. In both phases, it traverses nodes based on the complex traversal model (Figure 3.6b).



(a) Section's topology (b) Task's topology

Figure 3.8. Section and Task's topology



(a) DAG round (b) DAG long

(c) Timeline (d) Parallelism histogram

Figure 3.9. Four kinds of view

3.3.2 DAG View with Long Nodes

The algorithm is similar with that of DAG view with round nodes, also including two phases, setting xpre, y, lw, rw, dw and link_lw, link_rw, link_dw first then setting xp and x, but there is a different point that it calculates dw and link_dw differently. The height of a node is not constant anymore but based on work time of the node. Because nodes' work time varies tremendously as there are nodes that are as small as some hundred of nanoseconds and nodes that are as large as some hundred of milliseconds, if node height is set proportionally with work time, one node can be millions of times longer than another. Therefore, we have implemented a scale-down functionality for this view. The scale-down degree can be of linear, logarithmic or power functions of adjustable radixes (Figure 3.10). These radixes can be changed via GUI.

3.3.3 Timeline View

Timeline view (Gantt chart) is a popular visualization adopted by many visualization tools. In timeline view, the x-axis is the time flow and y-axis includes a number of rows each of which

```
1  double
2  dv_dag_calculate_vresize(dv_dag_t * D, double val) {
3    double ret;
4    switch (D->sdt) {
5    case 0:
6      ret = log(val) / log(D->log_radix);
7      break;
8    case 1:
9      ret = pow(val, D->power_radix);
10     break;
11   case 2:
12     ret = val / D->linear_radix;
13     break;
14   default:
15     dv_check(0);
16     break;
17   }
18   return ret;
19 }
```

Figure 3.10. Scale down

corresponds to one worker thread. The rows contain boxes representing works that worker were doing at specific points of time.

There is a bug in cairo graphics library that it can not zoom into too tiny boxes. When being zoomed in too much the rendering gets failed, a part of the surface gets painted by one color. This bug origins from the 24.8 fixed-point format used by cairo's device backend. In cairo's device-space, coordinates are stored in 24.8 fixed point format (24 bits for integer number before the point, and 8 bits for real number after the point), they have a limit of maximum value at around 8 million (2^23). When zooming in too largely the coordinates passed to the device-space (= user-space coordinates $\times$ zoom ratio) will surpass this limit causing wrong drawing.

The solutions we used to overcome this bug are two as following:

- **cairo_clip()** function: this function provided by cairo instructs cairo to draw only things inside a predefined box, and ignore others outside. However, using this function alone is not enough. Because when there is a shape stretching from inside the clipping box out to the outside, crossing the limit point, rendering still malfunctions.
- **DAGViz does clipping itself**: DAGViz cuts down parts of a shape that cross the clipping box when drawing nodes.

With these two solutions, DAGViz can now zoom into the thinnest box in timeline view.

3.3.4 Parallelism Histogram View

By traversing the DAG hierarchically and drawing the according parallelism profile, DAGViz can produce more flexible parallelism profile that the statistical images rendered by Gnuplot. We also make DAGViz draw a timeline view sticking to the bottom of the histogram.

Figure 3.11 shows the parallelism profile of Sort application down to depth 1. Figure 3.12 shows that down to depth 5. And Figure 3.13 shows that down to depth 10.

There are some shortcomings of current implementation of this view:

- slow, not scalable: currently it can deal with up to 10 thoundsand nodes within 1-2 seconds but gets very slow when there are more nodes. The reason is because the time x-axis is divided into too many entries.

Figure 3.11. Sort's parallelism profile at depth 1

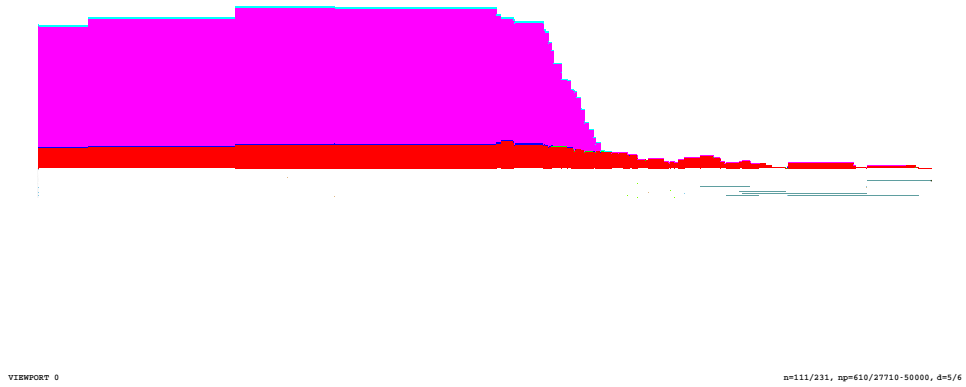


Figure 3.12. Sort's parallelism profile at depth 5

3.4 Rendering

The DAG is drawn on a drawing surface. The drawing functionality is provided by *cairo* library which is also used as the base rendering system in *gtk+*.

DAGViz traverses the DAG and draws each node based on its coordinates. It then traverses and draws edges connecting related nodes later.

3.5 Animation

DAGViz supports two kinds of animations. They are collapse/expand animation and motion animation.

3.5.1 Collapse/Expand Animation

Animation duration is set as 400 milliseconds, and animation step is set as 30 milliseconds at default.

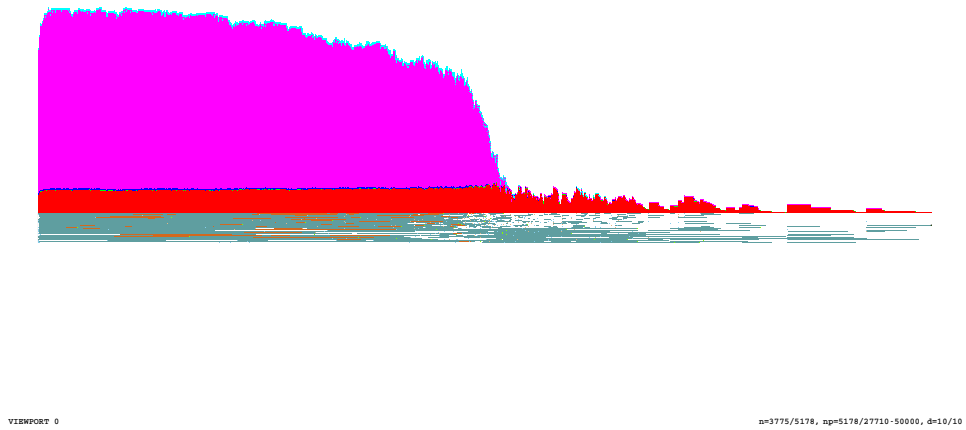


Figure 3.13. Sort's parallelism profile at depth 10

GLib is a low-level system library providing common data structures and basic mechanisms that other libraries and applications need to be built upon. Gtk+'s non-GUI-specific code is based on GLib. Among various things, GLib particularly provides a timeout mechanism that we used to make the animation in DAGViz. The mechanism works as that we register a function to GLib for it to call regularly after a predefined period of time. The first call to the function will be at the end of the first interval, GLib keeps calling the function repeatedly until it returns FALSE. The mechanism is provided through *g_timeout_add(interval, function, data)* interface in which *interval* is the time in milliseconds between two consecutive calls to the function who is pointed by the function pointer *function* and accepts variable *data* of type gpointer as the only parameter and should return a boolean value of type gboolean.

Every node who is on animation is set its start time in its member variable (*started*). Using this start time, current time and the predefined duration of the animation, we can calculate the **ratio** of the progress of the animation. But this **ratio** is linear, and linear animation does not look very natural and beautiful. So we transform this linear-progressing **ratio** into some other form like polynomial-progressing **rate** based on simple mathematical formula.

If the node is expanding:

$$rate = ratio \longrightarrow rate = 1 - (1 - ratio)^2$$

Or if the node is shrinking:

$$rate = 1 - ratio \longrightarrow rate = (1.0 - ratio)^2$$

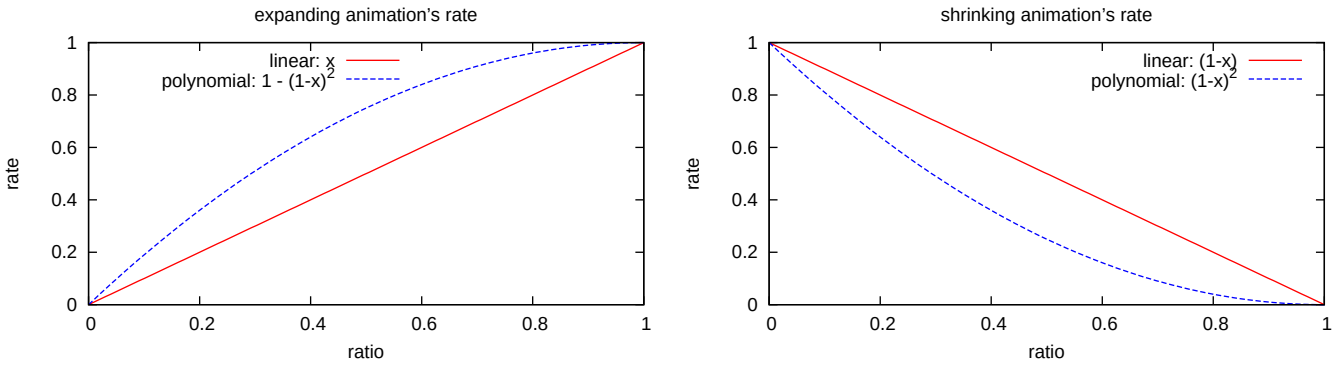
The differences between these linear and polynomial scale can be seen visually by graphs in Figure 3.14.

Following is the formula for calculating reverse rate when a node who is expanding is ordered to switch to shrinking:

$$reverse\_rate = 1 - \sqrt{rate}$$

And below is the formula for calculating reverse rate when a node who is shrinking is ordered to switch to expanding:

$$reverse\_rate = 1 - \sqrt{1 - rate}$$



(a) Expanding (b) Shrinking

Figure 3.14. Animation's rate

A shrinking/expanding *union* node would be drawn with alpha to express that it is fading in or fading out. These alphas are calculated based on following formula:

$$alpha_{fading\_out} = 1 - ratio^2$$

$$alpha_{fading\_in} = ratio^2$$

Its visual progress is shown in Figure 3.15.

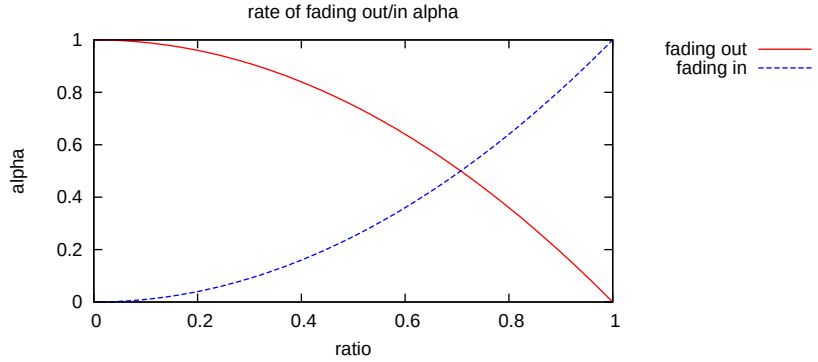


Figure 3.15. Rate of alpha for fading out/in

The interfaces to the animation mechanism are listed here:

- *dv_animation_init()*: initialize variables of the **V**'s *dv_animation_t* structure.
- *dv_animation_tick()*: is the function to be called at regular intervals to adjust the DAG's layout and request to re-draw the DAG on screen.
- *dv_animation_start()*: calls *g_timeout_add()* to initiate the timer and register *dv_animation_tick()* to be called regularly.
- *dv_animation_stop()*: simply modifies *dv_animation_t* structure as that the animation stopped. It would be called by the last occurence of *dv_animation_tick()*.
- *dv_animation_add()*: to add a new node to be on animation
- *dv_animation_remove()*: to remove a node from animation
- *dv_animation_reverse()*: to reverse the collapse/expand animation of a node

3.5.2 Motion Animation

Motion animation is like an automatic pan operation. It moves the view to a specific point gradually. This motion animation is used when user searches for a node, and the view moves to makes that node into the user's vision. Its mechanism is like that of the collapse/expand animation.

- *dv_motion_init()*: initializing function
- *dv_motion_tick()*: called regularly to adjust the rendering
- *dv_motion_start()*: start the motion animation
- *dv_motion_stop()*: stop the motion animation
- *dv_motion_reset_target()*: reset target to move towards

3.6 External Appearance

3.6.1 GUI

We use *gtk+* library to make the GUI of our visualizer. *gtk+* also provides mechanisms to control interactive actions from users such as mouse clicks, mouse moving, key pressing.

Viewport's hierarchical division

Screen is divided into nested viewports through an dialog that the user can adjust based on their preferences.

3.6.2 Interaction

Layout and draw make DAG visible on the screen. Users then must need to interact with this DAG visualization to explore details such as moving DAG around, zooming to parts of interest to view better. We have implemented some basic interaction features in this prototype:

- Pan: users drag visualization around to view hidden parts
- Zoom: users manify or shrink the visualization to specific parts to view better
- Info tag: displays full information associated with a particular node, at least for correctness checking purpose.
- Collapse/expand animation: collapse sub-DAGs to aggregate nodes (*section*, *task*) to hide details, providing a more general view.

DAGViz is implemented an ability for user to choose what it should do when the user does mouse **scrolling**: zoom horizontal & zoom vertical separately, scale radius (y-axis) & scalle radix (x-axis). This feature is convenient to change size of the visualization without distorting text.

3.6.3 Exporting Views

DAGViz is equipped with the ability to export views to png/eps file format. The produced eps image is much more beautiful than one that is converted from PrintScreen image.

Chapter 4

Case Studies

4.1 BOTS: Barcelona OpenMP Task Suite

Execution environment stack are shown in Table 4.1.

Compiler	gcc 4.4.7
Task parallel library	MassiveThreads
OS	CentOS 6.4 (Linux 2.6.32-x86_64)
#cores	64
CPU	AMD Opteron 6380 2.5GHz

Table 4.1. Environment

Benchmark applications are originally from the Barcelona OpenMP Task Suite (BOTS) which is a collection of applications used to evaluate tasking layer implementation of the runtime system (Table 4.2).

Name	*Summary*
Alignment	aligns sequences of proteins
FFT	computes a Fast Fourier Transformation
Fib	computes Fibbonaci number
Floorplan	computes the optimal placement of cells in a floorplan
Health	simulates a country health system
NQeens	finds solutions of the N Queens problem
Sort	uses a mixture of sorting algorithms to sort a vector
SparseLU	computes the LU factorization of a sparse matrix
Strassen	computes a matrix multiply with Strassen's method
UTS	computes the number of nodes in an Unbalanced Tree

Table 4.2. Benchmark applications

Experiment parameters used to run BOTS's applications are shown in Table 4.3.

Experiment results of the benchmark applications, DAG file size and numbers of nodes are shown in Table 4.4.

Running results of BOTS's ten applications by DAGViz's DAG views with round nodes long nodes are gathered in Figure 4.1, Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5, Figure 4.6, Figure 4.7, Figure 4.8, Figure 4.9, Figure 4.10.

Table 4.3. Summary of benchmarks settings. They are all the applications in Barcelona OpenMP Task Suites.

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`

Name	*DAG file size*	*#nodes (materialized)*
Alignment	3.9M	9,904
FFT	18.0M	47,438
Fib	4.0M	10,636
Floorplan	8.6M	22,415
Health	6.7G	17,796,837
NQeens	8.2M	21,360
Sort	11M	27,710
SparseLU	117M	302,920
Strassen	6.0M	15,692
UTS	1.1G	2,761,694

Table 4.4. Experiment results

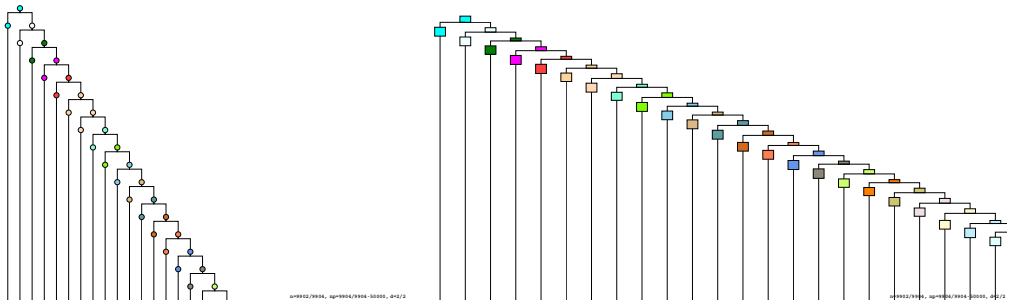


Figure 4.1. Alignment

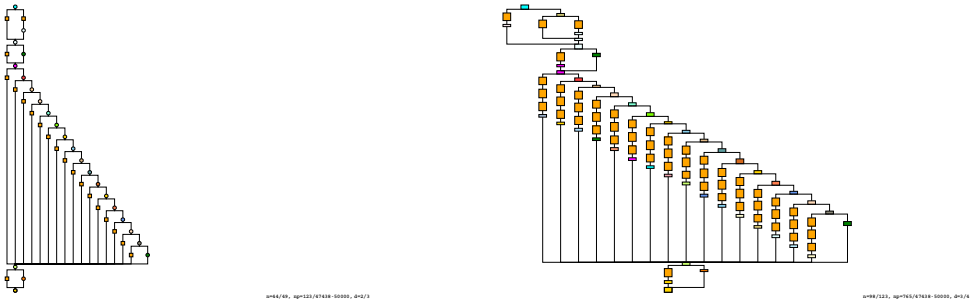


Figure 4.2. FFT

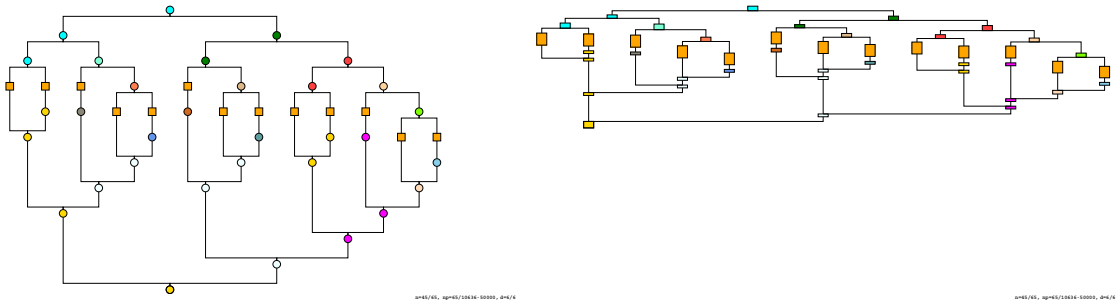


Figure 4.3. Fib

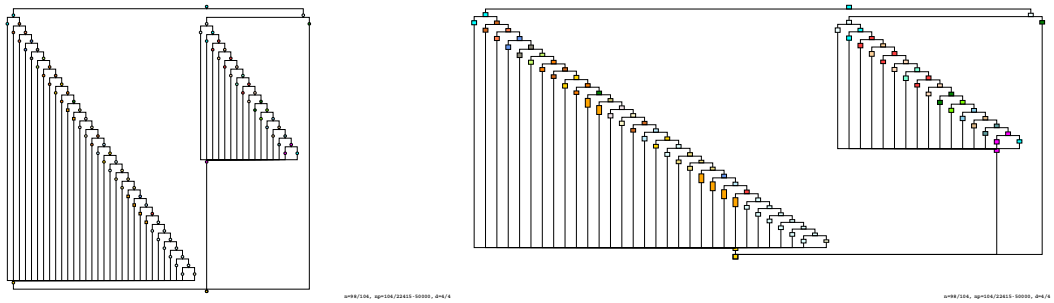


Figure 4.4. Floorplan

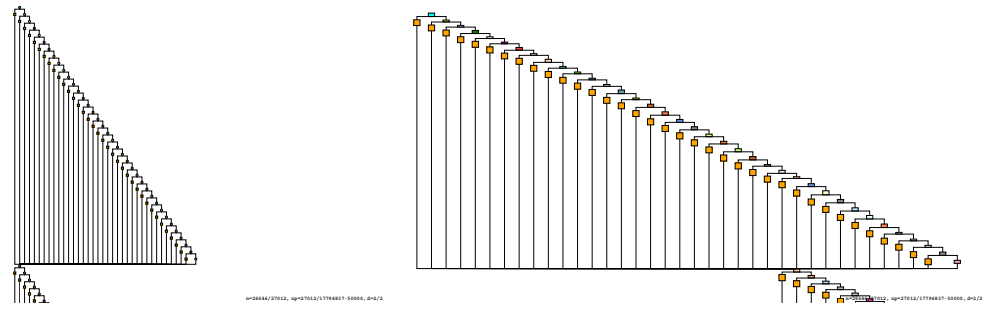


Figure 4.5. Health

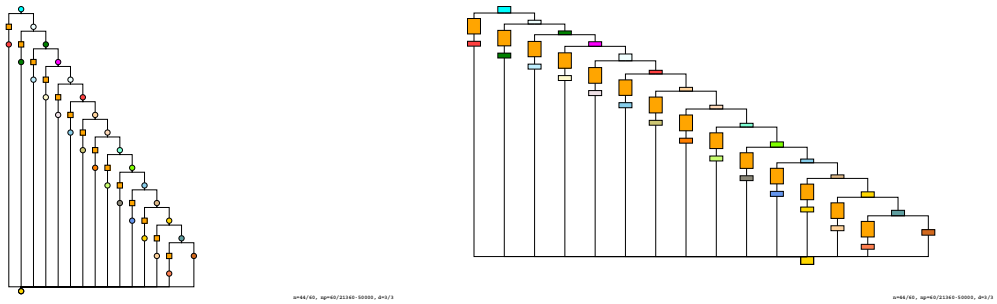


Figure 4.6. NQueens

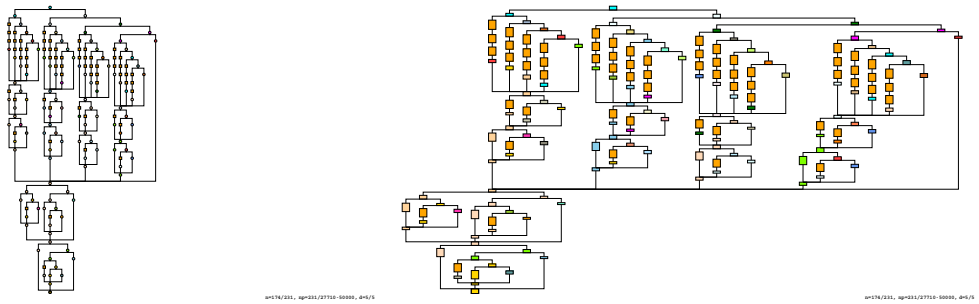


Figure 4.7. Sort

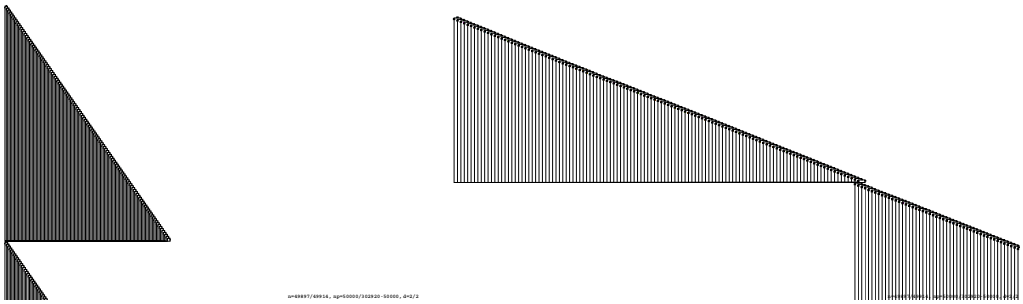


Figure 4.8. SparseLU

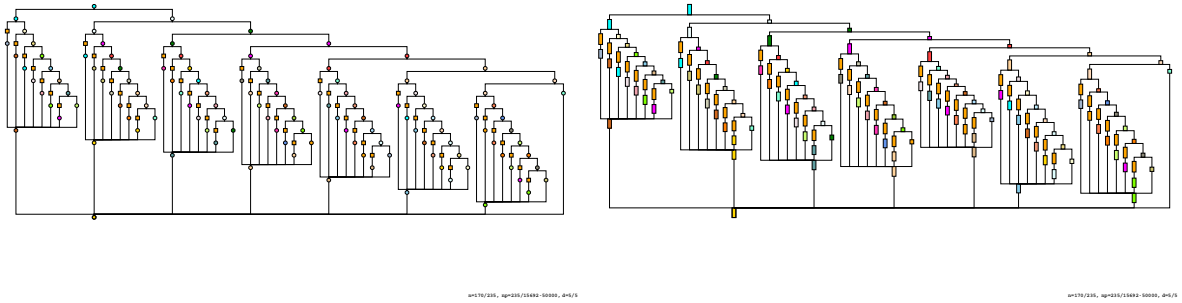


Figure 4.9. Strassen

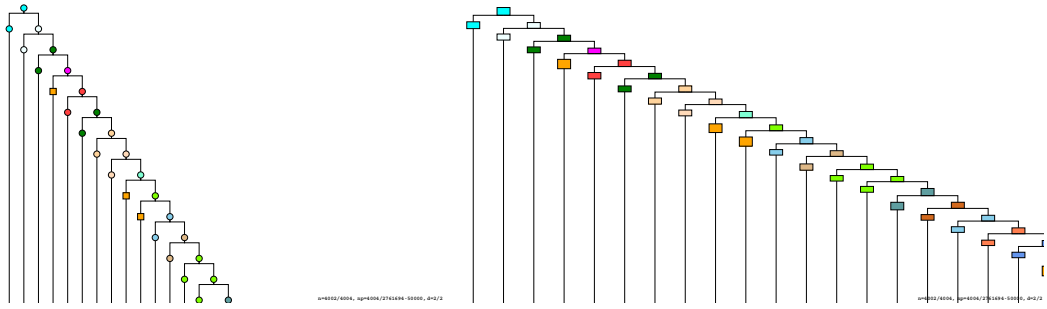


Figure 4.10. UTS

Chapter 5

Evaluation

5.1 DAG Recorder

Figure 5.1 shows overhead of DAG Recorder. Except for particular cases of Health and UTS programs, for all others, DAG Recorder is feasible.

5.2 DAGViz's Scalabilty

Memory-surpassing sizes Practical task-parallel programs can produce very big DAGs which do not fit into the memory. If the visualizer ignores this problem and behave as if there was no memory overload, thrashing, the phenomenon where the OS constantly exchanging data in memory for data on disk, would occur, causing unstable state for the visualizer application and even the whole machine.

Figure 5.2 illustrates an easy-to-view picture of size ranges that a DAG file can have. "Memory-overwhelming" indicates these cases of sizes that surpass physical memory size. Obviously, a countermeasure for this situation is to make the visualizer not to read all data at once but to read only necessary parts that are needed for displaying currently visible sub-DAG on screen. Considering a big DAG file as a very long stick, for one time the memory can hold only one continual fragment of that stick. The parts of DAG that are in that fragment can be displayed and explored very quickly. But if the user navigates out of these parts, which requires loading another fragment, the speed would get much slower because it involves exchanging memory pages.

Users commonly want to navigate the visualization geographically along four directions of up, down, right and left. If those parts that are geographically nearby are placed near to each

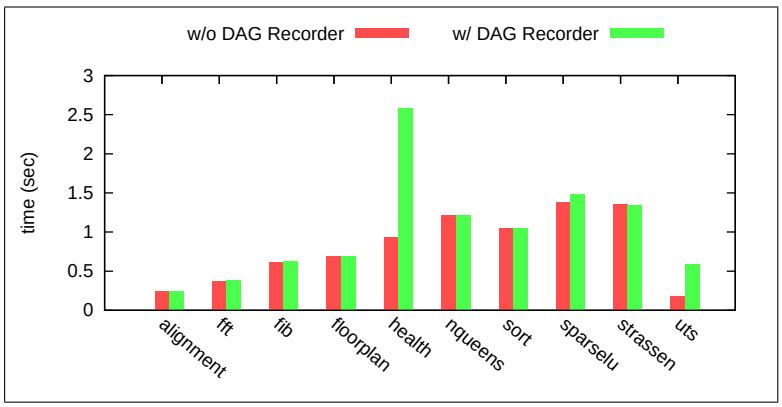


Figure 5.1. dr overhead of mth on 64 workers

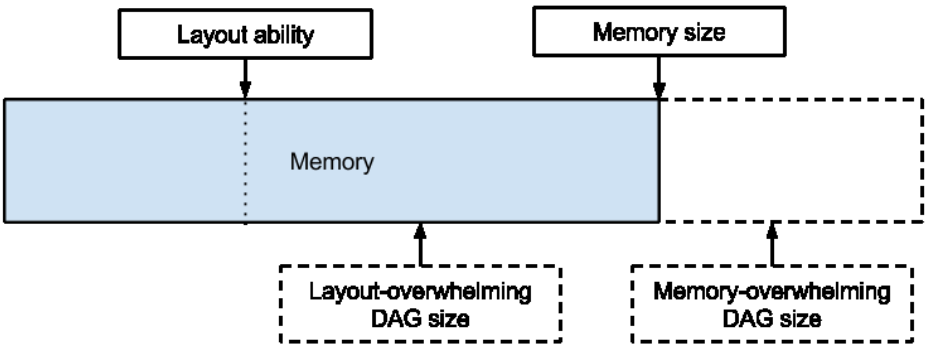


Figure 5.2. DAG size's ranges

other in the file so that they can be likely loaded together to memory by one read, it would be helpful for the visualizer's performance.

DAG has a hierarchical structure which enables users to navigate semantically vertical direction by collapsing/expanding sub-DAGs. This is convenient because users can get general information of a branch of the DAG before going to its detail, as well as they can compare two branches before deciding which one to explore further. Obviously aggregate data of nearby sub-DAGs should be placed near to each other, and their detailed data can be far from each other.

All of the work of in-file placement of DAG data we discussed so far can be summarized to one word of **file format**. A DAG file can be of binary type which stores original in-memory data structures of nodes/edges. The thing that matters here is the way these structures are placed in the file. They must be placed so that the arrangement supports the best for the visualizer's operations, resulting in the best possible performance.

Layout-surpassing sizes For DAG sizes that are sustainable for the memory there is another problem. Layout is an essential component of a visualizer, which determines the positions of nodes and edges of a graph so that latter components of the visualizer can draw them onto screen. Because the layout computation is usually the most costly part of a visualizer, it can be said that a visualizer performs fast or slow depending on how efficient its layout process can work.

Moreover, to maintain the responsiveness of a visualizer, its layout process must not run longer than a desired time limit. If not, users would get uncomfortatble and stick to the feeling that the tool is uncomplete, which is obviously not what we want.

In Figure 5.2, "layout ability" denotes the DAG size upper bound within which the layout process can finish no longer than the desired time limit. All DAG sizes that are under the memory size but above the "layout ability" bar are considered as "layout-overwhelming". To address DAGs with layout-overwhelming sizes, we can have the visualizer do layout and drawing work on only a part of the DAG loaded in the memory. One intuitive way to limit parts to do the layout is to draw only the part that are currently visible to the user. This ability involves the work of organizing the DAG's internal data structure so that the size and position of a sub-DAG can be understood rightaway when the traversal reaches that sub-DAG without going deeper into it.

Raising layout-ability bar One more point we want to discuss here is techniques to raise the layout ability bar in Figure 5.2. They are **recursion elimination** and **parallelization**. For traversals of all nodes of the DAG whose data structure is quite complicated, recursion must

be the first best choice. However, when the visualizer's layout computation has been proved to be correct, some efforts should be given in replacing recursion by other more efficient methods such as using self-organized stack instead. At least, a self-organized stack can help to avoid the overhead of a large number of function calls incurred in the traversal with recursive manner.

As of the era of multicore where even a commodity computer can have more than one processing cores, parallelizing an application can expose the potential to accelerate its performance up to several times according to the available number of cores. More specifically, traversing and processing each node of the hierarchical DAG structure are fit to task-parallel models. Besides, by separating layout process from the visualizer's main thread, its responsiveness can be improved considerably. The visualizer can inform users about current state, or terminate the on-going layout process if it takes too long.

Chapter 6

Conclusions and Future Work

Conclusions

We have built a visualizer that displays the DAG on screen, provides interaction functionalities for the user to explore the DAG. The DAG can be visualized by many kinds of views such as DAG form with round nodes which is useful in helping users conceive structure of the task parallel program, DAG form with long nodes which helps users in comparing size of nodes at a glance, timeline view which is a traditional and popular visualization in distinguishing worker threads and parallelism profile view which is new to us and useful for showing available and actual parallelism in the course of the execution time.

Future Work

In future work, we would like to combine the sampling method with DAG Recorder to get a more complete observation of long running intervals. DAG Recorder currently records only time metrics, we intend to enhance it to recorder other hardware performance counters [34] as well in order to get more thorough measures to reason about the performance. DAGViz is an indispensable tool to convey insights to the users, so we also need to develop it to display new data about performance counters, sampling samples that DAG Recorder provides. Besides, to compare two isomorphic DAGs to analyze the **work stretch** factor is a very potential direction of DAGViz.

Acknowledgement

I would like to say a big thank to my advisor, Professor Taura for his close, thorough and thoughtful guidance on my study and research in the couse of three years I have been in his laboratory. While the University of Tokyo created an excellent study environment in which I have had opportunities to extend my general knowledge by attending lectures of various topics and fields, Taura laboratory provides me with a closer and more focused environment so that I could train and improve in depth my specialized capability. Professor Taura makes me a perfect target to aim to by his precise thinking, brilliant programming skill, superb presentation skill and many more. I appreciate very much his patience on teaching me, explaining things to me multiple times and even writing email to explain them in written words when I do not understand due to my poor ability in communication, Japanese language and doing research. I would also like to thank all members in Taura laboratory for being around and always be there whenever I need help. Talking and doing research together with them were meaningful experience to me. I have enjoyed so much the time we hung out for food and drinking at yakitori restaurants.

It is really deficient if I do not mention the University of Tokyo's Global Creative Leader (GCL) graduate program. GCL program gathers students of various fields together for working towards interdisciplinary innovations. If it is not the GCL, I would not have had chance to get acquainted and talk with that many students from that various fields. GCL has also funded me to go for an abroad internship at the University of Notre Dame in the United States. This was a precious exprerience to me that helps widen my vision and enhance my decision-making instinct that is hardly described by words. I really appreciate Professor Thain for accepting me to his research group. This has enabled me to have opportunities talking with his students and other students at the University of Notre Dame.

Last but not least, I would like to say thanks to my family and friends who have always been by my side all the time, especially when I feel desperate. Without them I could not have made this far in the road that I chose.

Publications

- An Huynh, 中島潤, 田浦健次朗, "A Performance Analyzer for Task Parallel Applications based on Execution Time Stretches", Summer United Workshops on Parallel, Distributed and Cooperative Processing (SWoPP) 2013
- An Huynh, 中島潤, 田浦健次朗, "A Performance Analyzer for Task Parallel Applications based on Execution Time Stretches", Symposium on Advanced Computing Systems and Infrastructures (SACSIS) 2013
- S. Sigg, Y. Ji, N. Nguyen, A. Huynh, "AdhocPairing: Spontaneous audio based secure device pairing for Android mobile devices", International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use (IWSSI) in conjunction with Pervasive 2012, June 2012
- Ngu Nguyen; Sigg, S.; Huynh, A.; Yusheng Ji, "Pattern-Based Alignment of Audio Data for Ad Hoc Secure Device Pairing," Wearable Computers (ISWC), 2012 16th International Symposium on , vol., no., pp.88,91, 18-22 June 2012
- Ngu Nguyen; Sigg, S.; Huynh, A.; Yusheng Ji, "Using ambient audio in secure mobile phone communication," Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on , vol., no., pp.431,434, 19-23 March 2012

Bibliography

[1] Garcia, F. and Fernandez, J.: POSIX Thread Libraries, *Linux J.*, Vol. 2000, No. 70es (online), 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 (2011).

[3] Frigo, M., Leiserson, C. E. and Randall, K. H.: The Implementation of the Cilk-5 Multithreaded Language, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, New York, NY, USA, ACM, pp. 212–223 (online), 10.1145/277650.277725 (1998).

[4] Leiserson, C. E.: The Cilk++ concurrency platform, *Proceedings of the 46th Annual Design Automation Conference DAC '09*, New York, New York, USA, ACM Press, p. 522 (online), 10.1145/1629911.1630048 (2009).

[5] Reinders, J.: *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, O'Reilly Media (2007).

[6] 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), 10.1109/IPDPS.2008.4536359 (2008).

[7] 中島潤, .: 高効率な I/O と軽量性を両立させるマルチスレッド処理系, Vol. 4, No. 1, pp. 13–26 (2011).

[8] 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), 10.1145/2491661.2481433 (2013).

[9] Teruel, X., Martorell, X., Duran, A., Ferrer, R. and Ayguadé, E.: Support for OpenMP tasks in Nanos v4, *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research - CASCON '07*, New York, New York, USA, ACM Press, p. 256 (online), 10.1145/1321211.1321241 (2007).

[10] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *Journal of the ACM*, Vol. 46, No. 5, pp. 720–748 (online), 10.1145/324133.324234 (1999).

[11] Mohr, E., Kranz, D. A. and Halstead, Jr., R. H.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs, *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, New York, NY, USA, ACM, pp. 185–197 (online), 10.1145/91556.91631 (1990).

[12] 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), 10.1109/ICPP.2009.64 (2009).

[13] 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), 10.1145/1504176.1504210 (2009).

[14] 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).

[15] 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), 10.1177/1094342006064482 (2006).

[16] Intel: Intel VTune Amplifier, Intel Inc. (online), http://software.intel.com/en-us/intel-vtune-amplifier-xe 2013.

[17] 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), 10.1145/2555243.2555271 (2014).

[18] He, Y., Leiserson, C. E. and Leiserson, W. M.: The Cilkview Scalability Analyzer, *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, New York, NY, USA, ACM, pp. 145–156 (online), 10.1145/1810479.1810509 (2010).

[19] Shneiderman, B.: The eyes have it: a task by data type taxonomy for information visualizations, *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pp. 336–343 (online), 10.1109/VL.1996.545307 (1996).

[20] Landge, A., Levine, J., Bhatele, A., Isaacs, K., Gamblin, T., Schulz, M., Langer, S., Bremer, P.-T. and Pascucci, V.: Visualizing Network Traffic to Understand the Performance of Massively Parallel Simulations, *Visualization and Computer Graphics, IEEE Transactions on*, Vol. 18, No. 12, pp. 2467–2476 (online), 10.1109/TVCG.2012.286 (2012).

[21] Wu, J., Zeng, J., Yu, H. and Kenny, J. P.: CommGram: A New Visual Analytics Tool for Large Communication Trace Data, *Proceedings of the First Workshop on Visual Performance Analysis*, VPA '14, Piscataway, NJ, USA, IEEE Press, pp. 28–35 (online), 10.1109/VPA.2014.8 (2014).

[22] 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).

[23] 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), 10.1007/978-3-642-21878-$1_6$4(2011).

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

[25] Kergommeaux, J. C. D., Stein, B. D. O. and Martin, M. S.: Paje: An Extensible Environment for Visualizing Multi-Threaded Program Executions, *Proc. Euro-Par 2000, Springer-Verlag, LNCS*, pp. 133–144 (1900).

[26] 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), 10.1109/ICPPW.2010.34 (2010).

[27] 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), 10.1002/cpe.1469 (2010).

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

[29] Sugiyama, K., Tagawa, S. and Toda, M.: Methods for Visual Understanding of Hierarchical System Structures, *Systems, Man and Cybernetics, IEEE Transactions on*, Vol. 11, No. 2, pp. 109–125 (online), 10.1109/TSMC.1981.4308636 (1981).

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

[31] 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), 10.1145/2400682.2400712 (2013).

[32] : GTK+ 3, The GTK+ Project (online), http://www.gtk.org/ 2014.

[33] : Cairo, Cairo Graphics Project (online), http://cairographics.org/ 2014.

[34] 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).

Appendices

Appendix A

DAGViz's Data Structures

```
1  typedef struct dv_pidag {
2    long n;   /* length of T */
3    long m;   /* length of E */
4    long start_clock; /* absolute clock time of start */
5    long num_workers;  /* number of workers */
6    dr_pi_dag_node * T;  /* all nodes in a contiguous array */
7    dr_pi_dag_edge * E;  /* all edges in a contiguous array */
8    dr_pi_string_table S[1];
9    char * fn; /* dag file name */
10   struct stat stat[1]; /* file stat structure */
11   dv_llist_t itl[1]; /* list of pii's of nodes that have info tag */
12 } dv_pidag_t;
```

Figure A.1. **P** data structure

```
1  typedef struct dv_dag {
2    /* PIDAG */
3    dv_pidag_t * P;
4
5    /* DAG's skeleton */
6    dv_dag_node_t * T; /* array of all nodes */
7    char * To;
8    long Tsz;
9    long Tn;
10
11   /* DAG's content */
12   dv_dag_node_t * rt; /* root task */
13   int dmax; /* depth max */
14   double bt; /* begin time */
15   double et; /* end time */
16
17   /* expansion state */
18   int cur_d; /* current depth */
19   int cur_d_ex; /* current depth of extensible union nodes */
20
21   /* layout parameters */
22   int sdt; /* scale down type: 0 (log), 1 (power), 2 (linear) */
23   double log_radix;
24   double power_radix;
25   double linear_radix;
26   int frombt;
27   double radius;
28
29   /* other */
30   dv_llist_t itl[1]; /* list of nodes that have info tag */
31   dv_histogram_t * H; /* structure for the paraprof view (5th) */
32   char tolayout[DV_NUM_LAYOUT_TYPES];
33 } dv_dag_t;
```

Figure A.2. **D** data structure

```
typedef struct dv_view_status {
  // Drag animation
  char drag_on; /* currently dragged or not */
  double pressx, pressy; /* currently pressed position */
  double accdisx, accdisy; /* accumulated dragged distance */
  // Node color
  int nc; /* node color: 0->worker, 1->cpu, 2->kind, 3->last */
  // Window's size
  double vpw, vph; /* viewport's size */
  // Shrink/Expand animation
  dv_animation_t a[1]; /* animation struct */
  long nd; /* number of nodes drawn */
  int lt; /* layout type */
  int et; /* edge type */
  int edge_affix; /* edge affix length */
  int cm; /* click mode */
  long ndh; /* number of nodes including hidden ones */
  int focused;

  /* drawing parameters */
  char do_zoomfit; /* flag to do zoomfit when drawing view */
  double x, y; /* current coordinates of the central point */
  double basex, basey;
  double zoom_ratio_x; /* horizontal zoom ratio */
  double zoom_ratio_y; /* vertical zoom ratio */
  int do_zoom_x;
  int do_zoom_y;
  int do_scale_radix;
  int do_scale_radius;

  /* moving animation */
  dv_motion_t m[1];
} dv_view_status_t;

typedef struct dv_view {
  dv_dag_t * D; /* DV DAG */
  dv_view_status_t S[1]; /* layout/drawing attributes */
  dv_view_interface_t * I[DV_MAX_VIEWPORT]; /* interfaces to viewports */
  dv_viewport_t * mainVP; /* main VP that this V is assosiated with */
} dv_view_t;
```

Figure A.3. **V** data structure

```
1  typedef struct dv_dag_node {
2
3    /* task-parallel data */
4    //dr_pi_dag_node * pi;
5    long pii;
6
7    /* state data */
8    char f[1]; /* node flags, 0x0: single, 0x01: union/collapsed, 0x11:
         union/expanded */
9    int d; /* depth */
10
11   /* linking structure */
12   struct dv_dag_node * parent;
13   struct dv_dag_node * pre;
14   dv_llist_t links[1]; /* linked nodes */
15   struct dv_dag_node * head; /* inner head node */
16   dv_llist_t tails[1]; /* list of inner tail nodes */
17
18   /* layout */
19   dv_node_coordinate_t c[DV_NUM_LAYOUT_TYPES]; /* 0:grid, 1:bbox, 2:
         timeline, 3:timeline2 */
20
21   /* animation */
22   double started; /* started time of animation */
23
24 } dv_dag_node_t;
```

Figure A.4. DAGViz's node

Appendix B

Layout Algorithms

```
1  void dv_view_layout_glike_node(dv_dag_node_t * node) {
2    if (node->head) {
3      node->head->xpre = 0.0;
4      node->head->y = node->y;
5      /* Traverse inward */
6      dv_view_layout_glike_node(node->head);
7      node->lw = node->head->link_lw;
8      node->rw = node->head->link_rw;
9      node->dw = node->head->link_dw;
10   } else {
11     node->lw = RADIUS;
12     node->rw = RADIUS;
13     node->dw = 2 * RADIUS;
14   }
15   /* Traverse link-along */
16   switch (node->links.size()) {
17   case 0:
18     node->link_lw = node->lw;
19     node->link_rw = node->rw;
20     node->link_dw = node->dw;
21     break;
22   case 1:
23     next->xpre = 0.0;
24     next->y = node->y + (node->dw + DV_VDIS);
25     dv_view_layout_glike_node(next);
26     node->link_lw = max(node->lw, next->link_lw);
27     node->link_rw = max(node->rw, next->link_rw);
28     node->link_dw = (node->dw + DV_VDIS) + next->link_dw;
29     break;
30   case 2:
31     right_next->y = node->y + (node->dw + DV_VDIS);
32     left_next->y = node->y + (node->dw + DV_VDIS);
33     dv_view_layout_glike_node(right_next);
34     dv_view_layout_glike_node(left_next);
35     right_next->xpre = right_next->link_lw - RADIUS + DV_HDIS;
36     if (right_next->links.size() == 2)
37       right_next->xpre = - right_next->links[left_next]->xpre;
38     left_next->xpre = right_next->link_lw - RADIUS + DV_HDIS;
39     node->link_lw = - left_next->xpre + left_next->link_lw;
40     node->link_rw = right_next->xpre + right_next->link_rw;
41     node->link_dw = (node->dw + DV_HDIS) + max(right_next->link_dw,
           left_next->link_dw);
42     break;
43   }
44 }
```

(a)

Figure B.1. DAG with round nodes's layout algo phase 1

```
1  void dv_view_layout_glike_node_2nd(dv_dag_node_t * node) {
2    if (node->head) {
3      node->head->xp = 0.0;
4      node->head->x = node->x;
5      /* Traverse inward */
6      dv_view_layout_glike_node_2nd(node->head);
7    }
8    /* Traverse link-along */
9    switch (node->links.size()) {
10   case 0:
11     break;
12   case 1:
13     next->xp = next->xpre + node->xp;
14     next->x = next->xp + next->parent->x;
15     dv_view_layout_glike_node_2nd(next);
16     break;
17   case 2:
18     right_next->xp = right_next->xpre + node->xp;
19     right_next->x = right_next->xp + right_next->parent->x;
20     left_next->xp = left_next->xpre + node->xp;
21     left_next->x = left_next->xp + left_next->parent->x;
22     dv_view_layout_glike_node(right_next);
23     dv_view_layout_glike_node(left_next);
24     break;
25   }
26 }
```

(a)

Figure B.2. DAG with round nodes's layout algo phase 2

```
1  void dv_view_layout_bbox_node(dv_dag_node_t * node) {
2    if (node->head) {
3      node->head->xpre = 0.0;
4      node->head->y = node->y;
5      /* Traverse inward */
6      dv_view_layout_glike_node(node->head);
7      node->lw = node->head->link_lw;
8      node->rw = node->head->link_rw;
9      node->dw = node->head->link_dw;
10     /* some processes for enhancing expand/collapse animation*/
11     ...
12   } else {
13     node->lw = RADIUS;
14     node->rw = RADIUS;
15     node->dw = dv_view_calculate_vsize(node);
16   }
17   /* Traverse link-along */
18   switch (node->links.size()) {
19   case 0:
20     node->link_lw = node->lw;
21     node->link_rw = node->rw;
22     node->link_dw = node->dw;
23     break;
24   case 1:
25     ugap = dv_view_calculate_vgap(node->parent, node, next);
26     next->xpre = dv_layout_node_get_last_tail_xp_r(V, node);
27     next->y = node->y + node->dw + ugap;
28     dv_view_layout_bbox_node(next);
29     node->link_lw = max(node->lw, next->link_lw - next->xpre);
30     node->link_rw = max(node->rw, next->link_rw + next->xpre);
31     node->link_dw = node->dw + ugap + next->link_dw;
32     break;
33   case 2:
34     ugap = dv_view_calculate_vgap(node->parent, node, u);
35     vgap = dv_view_calculate_vgap(node->parent, node, v);
36     u->y = node->y + node->dw + ugap;
37     v->y = node->y + node->dw + vgap;
38     dv_view_layout_glike_node(u);
39     dv_view_layout_glike_node(v);
40     u->xpre = u->link_lw - RADIUS + DV_HDIS;
41     if (u->links.size() == 2)
42       u->xpre = - u->links[1]->xpre;
43     v->xpre = u->link_lw - RADIUS + DV_HDIS;
44     if (v->links.size() == 2)
45       v->xpre += u->link_lw - RADIUS - u->xpre;
46     node->link_lw = - v->xpre + v->link_lw;
47     node->link_rw = u->xpre + u->link_rw;
48     node->link_dw = node->dw + max(ugap + u->link_dw, vgap + v->
           link_dw);
49     break;
50   }
51 }
```


Figure B.3. DAG with long nodes's layout algo phase 1

```
1  void dv_view_layout_bbox_node_2nd(dv_dag_node_t * node) {
2    if (node->head) {
3      node->head->xp = 0.0;
4      node->head->x = node->x;
5      /* Traverse inward */
6      dv_view_layout_bbox_node_2nd(node->head);
7    }
8    /* Traverse link-along */
9    switch (node->links.size()) {
10   case 0:
11     break;
12   case 1:
13     u->xp = u->xpre + node->xp;
14     u->x = u->xp + u->parent->x;
15     dv_view_layout_bbox_node_2nd(u);
16     break;
17   case 2:
18     u->xp = u->xpre + node->xp;
19     u->x = u->xp + u->parent->x;
20     v->xp = v->xpre + node->xp;
21     v->x = v->xp + v->parent->x;
22     dv_view_layout_glike_node(u);
23     dv_view_layout_glike_node(v);
24     break;
25   }
26 }
```

(a)

Figure B.4. DAG with long nodes's layout algo phase 2

```
1  void dv_view_layout_timeline_node(dv_dag_node_t * node) {
2    node->lw = RADIUS;
3    node->rw = RADIUS;
4    node->dw = dv_view_calculate_vresize(V, pi->info.end.t - D->bt) -
         dv_view_calculate_vresize(V, pi->info.start.t - D->bt);
5    int worker = pi->info.worker;
6    node->x = V->D->radius + worker * (2 * V->D->radius + DV_HDIS);
7    node->y = dv_view_calculate_vresize(V, pi->info.start.t - D->bt);
8    if (node->head) {
9      /* Traverse inward */
10     dv_view_layout_bbox_node_2nd(node->head);
11   }
12   /* Traverse link-along */
13   switch (node->links.size()) {
14   case 0:
15     break;
16   case 1:
17     dv_view_layout_bbox_node_2nd(u);
18     break;
19   case 2:
20     dv_view_layout_glike_node(u);
21     dv_view_layout_glike_node(v);
22     break;
23   }
24 }
```

(a)

Figure B.5. Timeline's layout algo