Visualization of Parallel Execution Graphs *

Björn Steckelbach[1], Till Bubeck[1], Ulrich Fößmeier[2], Michael Kaufmann[1], Marcus Ritt[1], Wolfgang Rosenstiel[1]

[1] Universität Tübingen, Wilhelm-Schickard-Institut, Sand 13, 72076 Tübingen, Germany,
email: steckelb/bubeck/mk/ritt/rosen @informatik.uni-tuebingen.de
[2] Tom Sawyer Software, 804 Hearst Avenue, Berkeley, CA 94710,
email: foessmei@tomsawyer.com

Abstract. Measuring and evaluating the runtime of parallel programs is a difficult task. In this paper we present tools for performance evaluation and visualization in the distributed thread system (DTS), a programming environment for portable parallel applications. We describe the visualization of a parallel trace log as an execution graph using a novel layout algorithm which has been tailored to expose the structure of multithreaded applications.

1 Introduction

The measurement and evaluation of parallel runtime is a problem where very few tools exist. We present a parallel programming environment, the *distributed threads system* DTS [1, 2], which consists of a parallelizing compiler, a parallel runtime system and evaluation tools. For evaluation of parallel applications, the runtime system generates a trace log of parallel execution, which is postprocessed to calculate the contribution of each thread to the overall runtime. Even with this detailed information it is often difficult if not impossible to extract important quantities out of the huge amount of profiling data.

Hence, the next step was to visualize the parallel execution in a call graph. The goal was to be able to see several key characteristics of the execution profile in a glance: The overall structure of the application, its load balancing as well as the critical execution path.

The problem here was to tailor the graph layout to the specific needs of call graph visualization. Each thread has to be clearly distinguishable from other threads and the execution times should be easily recognizable. Existing layout algorithms proved to be insufficient for this task. The main contribution of this paper is a novel layout algorithm suitable for the visualization of runtime graphs.

* This research is partially supported by the DFG-Grant Ka812/4-2 "Graphenzeichnen und Animation" and by the DFG within SFB 382: Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern

In Section 2 we describe the generation of execution logs and profile graphs for parallel applications. Section 3 presents the detailed criteria for the visualization of call graphs, in Section 4 the algorithms und layout techniques used are discussed. Examples of parallel algorithms and their corresponding profile graphs are given in Section 5.

2 Timing Parallel Applications in DTS

Although all modern UNIX operating systems provide threads, there is no interface for measuring the execution times of single threads. Standard UNIX timing calls like `getrusage` or `times` provide only information about the virtual process time. For multithreaded processes, this is the accumulated thread execution time. There is no information about how many CPU time has to be accounted to each thread. Measuring wall clock time is completely inadequate, since the execution times are further disturbed by other applications running on the same machine. Therefore we decided to trace parallel execution based on virtual CPU time and calculate the runtime share of each thread using our own algorithms.

The DTS runtime system has been modified to trace the execution of multithreaded parallel programs. A DTS application usually is distributed to several independent hosts. Each host executes multiple threads. On uniprocessor nodes this allows to hide communication latencies, on multiprocessor nodes we are able to use all available CPUs.

Each node produces a separate logfile, gathering timestamps for all significant events during execution. For each of the following seven events the virtual process time and additional control information is logged:

init Start of computation on local node.

exit End of computation on local node.

start Creation of a new thread. The thread id is logged.

end Termination of a thread.

fork Start of execution of a thread on local node.

bjoin Local node issued a join on a thread. This event marks the begin of the join. The caller gets suspended until the thread to be joined has terminated.

ajoin Completion of a join. The thread issuing the join continues execution.

Based on the information in the logfiles, the computation time for all threads can be computed. Each thread has to be accounted for its share of the measured virtual process time. This is accomplished using a simple recursive algorithm, which relies on some basic assumptions on the thread scheduler. Details can be found in [3].

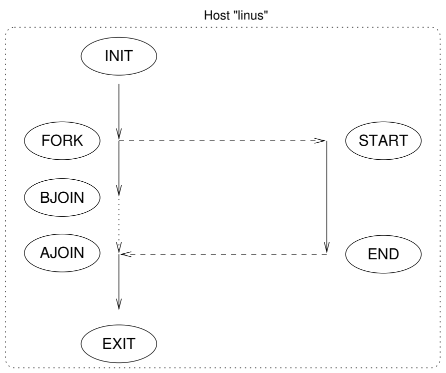


Fig. 1. Control flow for a single master and a single slave thread

3 A Graph Theoretical Formulation

Let $G = (V, E)$ be the given graph. Ignoring the init- and the exit-node (they carry no information for the graph) we can partition V in five subsets: $V = V_s \cup V_e \cup V_f \cup V_{bj} \cup V_{aj}$, namely *start-*, *end-*, *fork-*, *b-join-* and *a-join*-nodes.

Edges of G are either *flow-edges* $\in E_f$ which represent a part of a thread or *structural edges* $\in E_s$ which connect a subthread with its calling thread. Thus edges in E_s run from a fork-node to a start-node or from an end-node to a join-node and do not represent any sort of run time. Each flow-edge carries the information about the run time of the corresponding part of the thread.

A thread T consists of a chain $v_1, \ldots, v_t$ of nodes, v_1 being a start-node, v_t being an end-node and $v_2, \ldots v_{t-1}$ being fork- and join-nodes. Each thread is preceded by a fork-node and succeeded by an a-join-node. The nodes $v_2, \ldots v_{t-1}$ themselves are predecessors resp. successors of subthreads of T. $v_2, \ldots v_{t-1}$ can be paired into disjoint pairs (v_i, v_j) with $i < j$ such that v_i is the predecessor and v_j is the successor of the same subthread. If for any two subthreads of T defined by pairs (v_{i_1}, v_{j_1}), (v_{i_2}, v_{j_2}) holds: $i_1 < i_2 \Leftrightarrow j_2 < j_1$, then G is a series-parallel graph.

Our thread visualization tool TreVis computes drawings of runtime graphs with the following qualities:

- The five types of nodes can be easily distinguished.
- The drawing is orthogonal where every thread is represented by a chain of nodes in the same column.
- The drawing is hierarchically (top-down) such that flow-edges are drawn vertically and structural edges are drawn horizontally (with a possible vertical extension if necessary because of idle times).
- The number of edge crossings is at least locally minimal; the drawing is always planar for series-parallel graphs.

– Every thread T is balanced (if possible); i.e. T will be drawn near the barycenter of the subgraph induced by T and its subthreads.
– The run times are represented by the node positions. Here the y-axis is seen as a time axis and the y-coordinate of a node is proportional to the time when the corresponding action is performed.

Many applications show graphs where some edges have a very short length compared to other edges (e.g. Fig. 2: 0.01 vs. 3.14). We use a special scaling strategy here that sets short edges to a user- (or system-) defined minimum length; this makes it possible to distinguish the endpoints of this edge and to recognize the edge itself (see the first two join-nodes of the main thread in Fig. 2 for an example). If there is an edge of length zero between a b-join-node and the corresponding a-join-node, we do not distinguish between these nodes and draw them as a single join-node.

4 Algorithms

Since the edge routing is simple for given node positions (most of the edges are straight, some edges have one bend) the crucial part of the algorithm is to compute the node positions.

4.1 Computing y-Coordinates

Computing the y-coordinates is easy: The init-node is getting y-coordinate 0. Every start-, fork-, end-, and b-join-node is getting the value of the y-coordinate of its only predecessor plus the length of the corresponding edge. a-join-nodes have two predecessors (say u and w). W.l.o.g. u is an end-node and w is a fork- or a b-join-node. Place v at $\max(y\text{-coord}(u) + \text{length}(u,v),\ y\text{-coord}(w) + \text{length}(w,v))$. b-join-nodes have diamond shape in our drawings and indicate an idle time in this part of the program. Note that solid edges in the drawings indicate the node positions whereas dashed edges indicate idle times.

4.2 Computing x-Coordinates

We only have to compute an x-coordinate for every thread because all nodes of the thread will have the same coordinate. We distinguish whether the graph is series-parallel or not which is easy to decide (this information is often part of the input).

Series-Parallel Graphs. In series-parallel graphs a series T_i $(1 \leq i \leq t)$ with T_i is subthread of T_{i+1} $(1 \leq i \leq t-1)$ is nested. Thus they can easily be drawn without edge crossings. For balancing the drawing we use the following strategy:

We treat the subthreads of T one by one 'outermost-first' and store for every step the ranges currently used by the subthreads to the left and to the right of T. The next subthread to be drawn will be placed at the side of T with the smaller range. In the example of Fig. 2 we first choose to place the outermost subthread T_1 of the main thread T to the left of T. T_1 uses four columns there. T_2 (starting at the second fork-node of T) will be draw at the right side of T and uses two columns. Since the right range (two) is smaller than the left range (four) we draw T_3 (starting at the third fork-node of T) at the right of T.

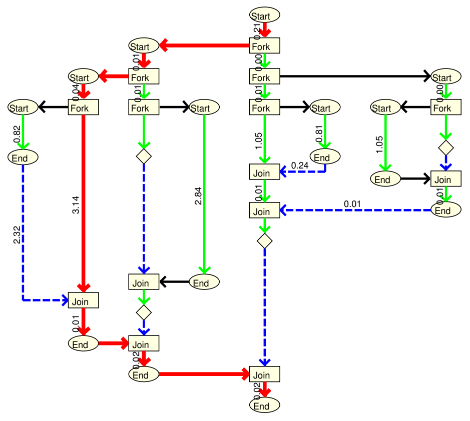


Fig. 2. A series-parallel graph.

General Planar and Nonplanar Graphs. Let T be a thread, T_1 and T_2 two subthreads of T, f_1 and f_2 the predecessors of the start-nodes of T_1 and T_2; and j_1 and j_2 the successors of the end-nodes of T_1 and T_2. If f_1 is placed above of (before) f_2 in the time-axis, but j_1 below (after) j_2, then T_1 and T_2 must cross if they are drawn at the same side of T. We call these threads intersecting. Thus we have to solve the problem of finding an assignment of the subthreads of T to the sides L and R (left and right) of T, such that no two subthreads on the same side cross or more general, we want to minimize the number of crossings.

The intersection structure of the subthreads of T can be formalized by the intersection graph $G_I = (S, I)$, where S is the set of nodes representing the direct subthreads of T and an edge $e = (T_i, T_j) \in I$ exists iff T_i, T_j are intersecting.

A non-crossing assignment of the subtreads to L and R corresponds to a subdivision of the nodes of G_I into two independent sets. This is possible only if the graph is bipartite. Hence, the tests whether the thread structure can be visualized using our drawing convention can be done greedily in linear time checking the bipartiteness property.

If the intersection graph is not bipartite, we cannot avoid all crossings. Since each remaining edge in the subgraphs induced by L and R represents a crossings, we have to minimize the number of those edges, or to maximize the number of edges between nodes in L and R. This is exactly the well-known max-cut problem [5]. Note that our intersection graphs are graphs similar to interval graphs [6]. The setting of the problem here is the same as for row routing [8, 7, 4], although in row routing the goal is to minimize the number of layers, vias and/or tracks, while we have a crossing minimization problem. To our current knowledge, it is unknown whether the problem is NP-complete or can be solved efficiently.

TreVis currently uses a greedy heuristic for that problem with a postprocessing local exchange step. That performs very well in practice (see examples in Section 5). For the future, we plan to incorporate exact methods from combinatorial optimization as well.

5 Some Examples

Fibonacci Numbers. In this example the Fibonacci numbers are calculated, using the well-known recursion formula. A slightly simplified version of the actual code and the resulting profile graph are shown in Fig. 3.

RSA Encryption. RSA encryption is an example for a regular non-recursive parallel algorithm. The main thread forks a number of slave workers, depending on the size of the input file, which each encrypt a single block of the plaintext using the RSA method.
Fig. 4 shows two call graphs of the same parallel execution using 13 threads. In the upper graph small edges are enlarged to emphasize the overall structure of the application. The calling sequence can be seen clearly. The lower graph shows the execution in true time scale. This view is of particular interest to evaluate the load balancing of the algorithm.

Bubble Merge Sort. Bubble merge sort uses a divide-and-conquer based approach to sort integer numbers. The divide step splits the array to be sorted and creates two subtasks running in parallel. Each thread uses the same algorithm recursively to sort its part. The conquer step grabs the pre-sorted subarrays and combines them using merge sort. The call graph of Bubble merge sort of 100000 Integers using 1024 threads can be seen in Fig. 5.

6 Conclusion

We presented a novel approach of visualizing the execution profiles of multi-threaded parallel applications. The visual inspection of the parallel call graphs has proved to be a very valuable tool in evaluating and tuning parallel applications. The layout algorithm presented here improved the usability and expressiveness of call graphs significantly.

```
int fib(int n) {
  dts_t dts_id1,dts_id2;
  int result=0;

  /* simple case */
  if (n<=1) return 1;

  /* recurse with two threads */
  dts_id1=fork_fib(n-1);
  dts_id2=fork_fib(n-2);
  result+= (int)join_fib(dts_id1);
  result+= (int)join_fib(dts_id2);
  return result;
}

int main(int argc, char *argv[]) {
  dts_init(argc,argv,NULL);
  for (i=0; i< 35; i++) fib(i);
  dts_leave();
}
```

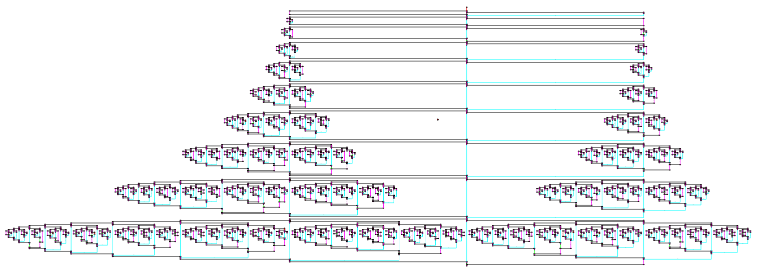
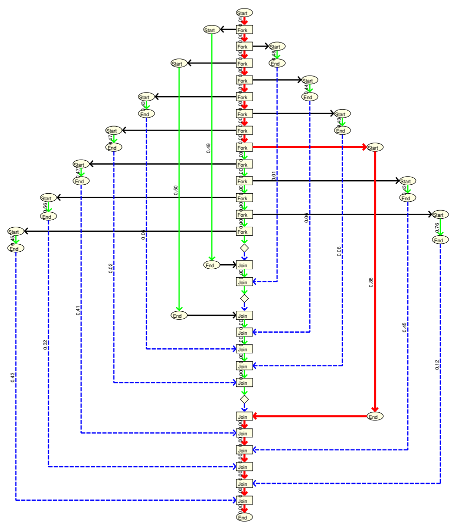


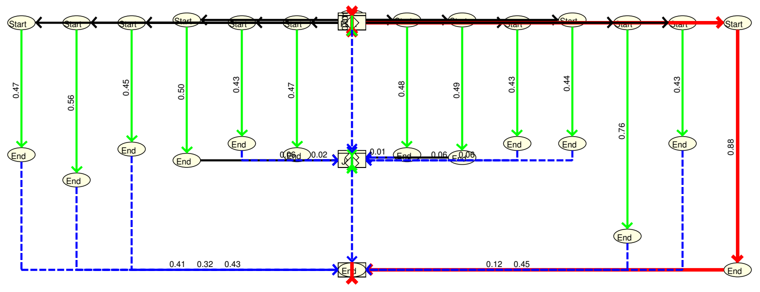
Fig. 3. Source code and call graph of Fibonacci calculation using 729 threads.

References

1. T. Bubeck. Eine Systemumgebung zum verteilten funktionalen Rechnen. Technical Report WSI-93-8, Eberhard-Karls-Universität Tübingen, August 1993.
2. T. Bubeck. *Distributed Threads System DTS User's Guide.* SFB 382/C6, Universität Tübingen, Sep 1995.
3. T. Bubeck, J. Schreiner, and W. Rosenstiel. Timing multi-threaded Message-Passing Programs. In C. A. Héritier, editor, *SIPAR Workshop 96*, pages 15–18, University of Geneva, Oct 1996.
4. A. Dingle and H. Sudborough. The complexity of single row routing problems. volume LNCS 382, pages 529–540, 1989.
5. M. Garey and D. Johnson. *Computers and Intractability.* Freeman, 1979.
6. M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academ. Press, 1980.
7. R. Raghavan and S. Sahni. Single row routing. *IEEE Transactions on Computers*, C-32:209–220, 1983.
8. T. Tarng, M. Marek-Sadowska, and E. Kuh. An efficient single row routing algorithm. *IEEE Transactions on CAD*, CAD-3:178–183, 1984.



A



B

Fig. 4. Call graphs of RSA encryption. In Fig. A small edges are enlarged to emphasize the calling structure, Fig. B shows the call graph in original time scale.

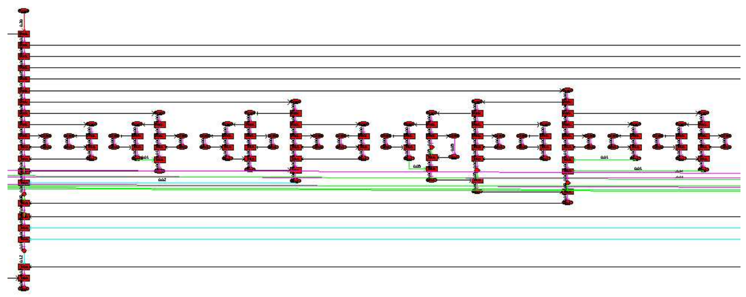


Fig. 5. A partial view of the call graph of bubble merge sort with 1024 threads.