Near–Critical Path Analysis of Program Activity Graphs

Cedell Alexander, Donna Reese, and James Harden

NSF Engineering Research Center for Computational Field Simulation
Mississippi State University

Abstract

Program activity graphs can be constructed from time-stamped traces of appropriate execution events. Information about the activities on the k longest execution paths is useful in the analysis of parallel program performance. In this paper, four algorithms for finding the near–critical paths of program activity graphs are presented and compared, including an efficient new algorithm that utilizes slack values calculated by the critical path method to perform a best–first search in linear space. The worst–case time and memory requirements of the new algorithm are in O(ke) and O(k+e), where e is the number of edges in the graph. Results confirming the efficiency of the algorithm are presented for five application programs. A framework for utilizing the near–critical path information is also described. The framework includes both statistical summaries and visualization capabilities.

Index Terms—*Critical path, program activity graph, instrumentation, parallel program performance analysis*

I. Introduction

Developing efficient parallel programs has proved to be a difficult task. Substantial research has been devoted to many aspects of the problem; active work spans the computer science spectrum from algorithmic techniques, programming paradigms, advanced compilers, and operating systems to architectures and interconnection networks. Complex interactions at each of these levels have provided motivation for a suite of performance measurement and analysis tools.

Insight into a system's dynamic behavior is a prerequisite for high–productivity optimization of parallel programs. Multiple tools, offering varying perspectives, may be required to gain the necessary insight. The IPS Parallel Program Measurement System [1] and the Pablo Performance Analysis Environment [2] are two significant toolkits facilitating different viewpoints based on timestamped probe descriptions of run–time events.

IPS provides a hierarchy of statistical information based on a five layer model consisting of the whole program, machine, process, procedure, and primitive activity levels. Critical path and phase behavior analysis techniques guide the search for performance problems. Critical path analysis focuses the optimization effort by identifying the activities on the longest execution path; to improve the program's performance, the duration of activities on the critical path(s) must be shortened.

Pablo is a visualization and sonification toolkit designed to be a *de facto* standard via a philosophy of portability, scalability, and extensibility. Custom performance analysis environments are constructed by graphically interconnecting a set of analysis and display modules. The graphical programming model encourages experimental exploration of the performance data.

The utility of critical path analysis can be extended when information is available about the *k* longest paths [3]. Optimization of specific critical path activities may provide little overall performance improvement if the second, third, etc., longest paths are of similar duration and consist of independent activities. Near–critical paths can be used to further refine the analysis process by quantifying the benefit of optimizing critical path activities. The focus of this paper is on an efficient algorithm for determining the near–critical paths of program activity graphs. Efficient algorithms are important because program activity graphs can be very large (hundreds of thousands of vertices).

Additionally, we present a framework for utilizing the near–critical path information that encompasses both statistical summaries (patterned after IPS) and the visualization capabilities of Pablo. Within the framework, guidance is provided by a new performance debugging metric. The *Maximum Benefit Metric* uses near–critical path data to predict the maximum overall performance improvement that may be realized by optimizing particular critical path activities.

In Section II, critical path algorithms are reviewed to provide the background needed for description of near–critical path algorithms in Section III. Probe acquisition and construction of program activity graphs are discussed in Section IV. Algorithm performance results are presented in Section V, and Section VI contains the proposed framework for near–critical path analysis. The paper is concluded in Section VII with a summary of key results and plans for future research.

II. Critical path algorithms

A. Program activity graphs

Program activity graphs (PAGs) are a new application of the activity on edge (AOE) networks that are often employed for planning and scheduling purposes. A PAG is an acyclic, directed multigraph representing the duration and precedence relationships of program activities. The edges represent execution activities, the weights of the edges represent activity durations, and the vertices mark activity boundaries. In AOE networks, outgoing activities from a vertex cannot begin until all incoming activities have completed.

Multigraphs are distinguished by multiple edges between a given pair of vertices. Although not all PAGs are multigraphs, depending on the semantics of the target system, generality requires that near–critical path algorithms accommodate multigraphs. The biggest impact of the multigraph characteristic is on data structure selection. For example, adjacency matrices are inappropriate data structures for multigraphs.

B. Longest path algorithm

IPS employs a modified shortest path algorithm, based on the diffusing computation paradigm [4], to find the path with the longest execution duration. A diffusing computation on a graph begins at the root vertices and diffuses to all descendant vertices. In the synchronous variation, a vertex will not diffuse a computation to its descendants until all incoming computations are received. The asynchronous variation diffuses the computation as soon as it receives any new computation. The asynchronous variation trades efficiency for concurrency, which is potentially attractive in a parallel environment; however, the results of experiments reported in [5] are not promising. A sequential version of the synchronous algorithm is given below.

Longest path algorithm
Q is a vertex queue
Activity edges are represented by descriptors <*num*,*t*,*h*,*d*>, where *num* uniquely identifies the edge, *t* is the tail (preceding) vertex, *h* is the head vertex, and *d* is the duration
D[v] contains the duration of the longest path from a root vertex to vertex *v*
P[v] contains a pointer to the descriptor of the final edge on the longest path to vertex *v*

```
for each vertex v in graph     // initialization
    D[v] := 0
    Count[v] := in–degree of vertex v
    if Count[v] = 0
        insert v in Q              // v is a root vertex
while Q is not empty
    remove vertex t from head of Q
    for each outgoing edge e from t
        Count[h] := Count[h] – 1
        if D[h] < (D[t] + d)
            P[h] := address(e's descriptor)
            D[h] := D[t] + d
        if Count[h] = 0
            insert h at tail of Q
```

C. Critical path method

The critical path method is an operational research algorithm for finding the longest path(s) through an AOE network [6]. The critical path method calculates early start and early finish times for each activity in a forward pass through the network. Late start times, late finish times, and slack values are calculated in a backward pass. The following definitions will be used to explain the algorithm:

Critical path method notation
d(i): duration of activity *i*
ES(i): early start time of activity *i*
EF(i): early finish time of activity *i*, *ES[i] + d(i)*
LS(i): late start time of activity *i*
LF(i): late finish time of activity *i*, *LS(i) + d(i)*
TS(i): total slack of *i*, *LS(i) – ES(i) := LF(i) – EF(i)*
FS(i): free slack of *i*, *ES(i's* immediate successors*) – EF(i)*

The early start time of an activity is the earliest possible time the activity can begin. The late start time of an activity is the latest time the activity can start without extending the overall network completion time. The slack values are criticality measures. The total slack of an activity is the amount of time that it can be delayed without impacting the overall completion time. Activities with zero total slack are on a critical path. The free slack of an activity is the amount of time the activity can be delayed without impacting the early start time of any other activity. The total slack values of activities on a path are not independent; delaying an activity longer than its free slack reduces the slack of subsequent activities. Fig. 1 shows the values calculated by the critical path method for a simple example network.

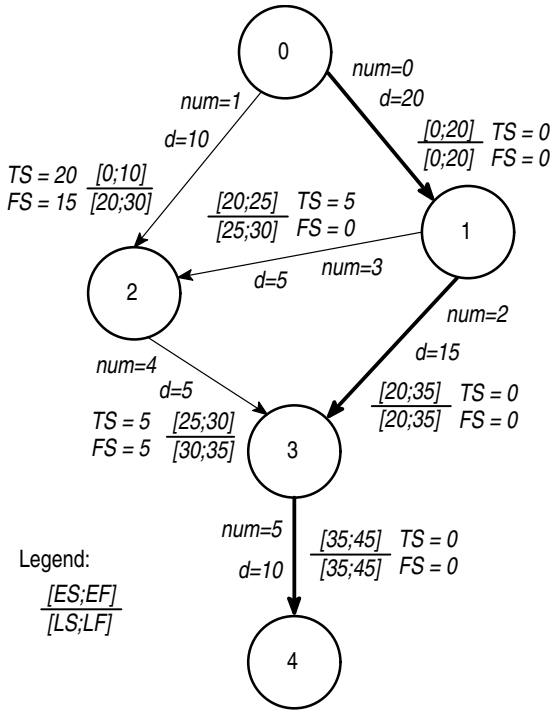


Fig. 1. Critical path method example.

The forward pass of the critical path method considers activities in topological order and the backward pass considers activities in reverse topological order. A topological ordering of activities is one in which no activity appears until all of its predecessors have appeared. Adjacency lists are an appropriate data structure for building a topological ordering [7]. An adjacency list for a vertex of a directed multigraph is a list of all outgoing edges. A count field containing the in–degree of the vertex is also maintained at the head of each list. When a multigraph with e edges is represented as a set of adjacency lists, the time required to perform a topological sort is in $O(e)$ [8] . Given a topological ordering, the forward pass of the critical path method can be performed in $O(e)$ time. In fact, the forward pass can even be performed in parallel with the topological ordering [8]. The backward pass can also be performed in $O(e)$ time. For succinctness, a topological ordering is assumed in the basic algorithm presented below.

Critical path method
A is an array of adjacency list heads
Adjacency list heads contain *ES* and *LF* fields specifying the early start time of all outgoing edges and the late finish time of all incoming edges for the associated vertex
Activity edges are represented by the 8–tuples <*num,t,h,d,EF,LS,TS,FS*>

Forward pass algorithm:
```
for each vertex v      // initialization
    A[v].ES := 0
    A[v].LF := INFINITY
critical_path_duration := 0
for each activity edge e in topological order
    EF := A[t].ES + d
    A[h].ES := max(A[h].ES,EF)
    critical_path_duration :=
                    max(critical_path_duration,EF)
```
Backward pass algorithm:
```
for each leaf vertex v
    A[v].LF := critical_path_duration
for each activity edge e in reverse topological order
    LS := A[h].LF – d
    A[t].LF := min(A[t].LF,LS)
    TS := LS – A[t].ES
    FS := A[h].ES – EF
```

D. Algorithm comparison

The longest path algorithm is more efficient than the critical path method. However, the critical path method produces more information; multiple critical paths are identified and the slack criticality measures are provided. Both algorithms have the same asymptotic time complexity, in $O(e)$. Selection of the most appropriate algorithm is dependent upon application needs.

III. Near–critical path algorithms

A. Path enumeration algorithm

An algorithm for listing the k shortest paths between two vertices, v_1 and v_n, of an acyclic digraph is described in [8]. The algorithm begins by finding the shortest path, p_1, between the two vertices. If p_1 contains m edges, denoted e_1, e_2, e_3, . . ., e_m, the second shortest path, p_2, is the minimum duration path in the set, P_2, of shortest paths that differ from p_1 in one of m ways. To form P_2, we find the shortest path from v_1 to v_n subject to each of the following m constraints: (1) the path includes e_2, e_3, . . ., e_m, but not e_1, (2) the path includes e_3, . . ., e_m, but not e_2, . . ., and (m) the path does not include e_m. The third shortest path is the shortest path in the set $(P_2 - \{p_2\}) \cup P_3$, where P_3 is generated by further partitioning of p_2. Subsequent paths are found in a similar manner. The algorithm could be easily modified to enumerate longest paths. For a multigraph containing n vertices and e edges, the worst–case time and memory requirements are in $O(kne)$ and $O(kn^2+e)$, respectively.

B. Longest path algorithm extension

A more straightforward approach is to simply extend the previously described longest path algorithm to find the k longest paths. To adapt the algorithm, we employ an array, *A*, of adjacency lists and augment the list heads with an array, *PD*, that contains k *path_descriptor* records of the form <*duration,edge_p,index*>. The *duration* field contains the duration of a preceding path from a root vertex, the *edge_p* field contains a pointer to the descriptor of the incoming edge on the path, and the *index* field contains the index of the corresponding record in the *PD* array of the immediately preceding vertex. Each *PD* array is ordered based on the *duration* fields. Conceptually, when an edge is evaluated, a new *PD* array is formed to extend the paths from the tail vertex to the head vertex. This new array is then merged with the existing *PD* array at the head vertex. The result represents the longest paths that have been found to the head vertex. After all edges have been evaluated, the *PD* arrays of leaf vertices are merged, and the result represents the longest paths in the graph. An algorithm for unraveling the paths is given below.

Path traversal algorithm
```
i := 0
for each path_descriptor record in final PD array (up to k)
    i := i + 1
    pd := address(PD[i])
    j := 0
    while pd–>edge_p–>t is not a root vertex
        j := j + 1
        P[j] := pd–>edge_p–>num
        pd := address(A[pd–>edge_p–>t].PD[pd–>index])
    P[j], . . ., P[1] contain numbers of edges on
                        i'th longest path
```

Since the time complexity of a *PD* array merge is in $O(k)$ and a merge must be performed for each edge, the time requirement of the extended longest path algorithm is in $O(ke)$. The storage requirements are in $O(kn+e)$, since a *PD* array must be maintained at each vertex and a descriptor is required to represent each edge.

C. Branch–and–bound algorithm

Brute–force depth–first searches can solve either the shortest or longest path problem in linear space; however, the time complexity is exponential [9]. Branch–and–bound (BnB) is a technique that may significantly improve the ef-

ficiency of depth–first searches by eliminating unproductive search paths [10]. BnB may be readily applied to the shortest path problem, but is not directly applicable to the longest path problem. In this section, we show how the slack values calculated by the critical path method can be used to transform the longest paths problem of finding near–critical paths into a shortest paths problem, and thereby enable utilization of BnB. We begin with a few definitions:

Near–critical path notation

nc%: near–criticality percentage, paths whose durations are within this percentage of the critical path duration are near–critical

min_ncp_duration: minimum duration of a near–critical path, *critical_path_duration* ∗ ((100 – *nc%*) ∗ .01)

FS_sum: sum of free slack on all preceding edges of path

max_path_duration: maximum potential duration of the current path at any edge of a depth–first search, *critical_path_duration – (FS_sum + TS)*

max_ncp_slack: maximum slack of a near–critical path, *critical_path_duration – min_ncp_duration*

To find the critical and near–critical paths, depth–first searches are started at the root vertices. A search is terminated when either a leaf vertex is reached or *max_path_duration* is less than *min_ncp_duration*. If a leaf vertex is reached, then a critical or near–critical path has been found. The recursive algorithm, which assumes the critical path method data structures, is given below.

Branch–and–bound algorithm

```
FS_sum := 0
depth := 0
for each root vertex r
    for each edge e from r to a descendant vertex v
        traverseEdge(e.num,r,v,e.TS,e.FS)
procedure traverseEdge(num,t,h,ts,fs)
// num is edge number, t is tail vertex, h is head vertex,
// ts is total slack, and fs is free slack
    if (FS_sum + ts) ≤ max_ncp_slack
        depth := depth + 1
        P[depth] := num
        if h is a leaf vertex
            if FS_sum = 0      // a critical path has been found
            else               // a near–critical path has been found
            path_duration :=
                    (critical_path_duration – (FS_sum + ts))
            1st depth entries in P contain #'s of edges on path
        else
            FS_sum := FS_sum + fs
            for each edge e from h to a descendant vertex v
                traverseEdge(e.num,h,v,e.TS,e.FS)
    FS_sum := FS_sum – fs
    depth := depth – 1
    return
```

The performance of the algorithm is highly dependent upon the input PAG. In the best case, the time complexity is in *O(1)*. If we optimistically assume that only one edge exists between any two vertices and that no vertex has more than two outgoing edges (which appear to be reasonable assumptions for PAGs based on our experiences and the results reported in [11]), the worst–case complexity, based on the number of edges that must be examined, is in $O(1.62^n)$. When the critical path method is also included in the analysis, the best–case and worst–case time complexities are in *O(e)* and $O(1.62^n+e)$, respectively.

D. Best–first search algorithm

The slack values provided by the critical path method can also be used as the basis for a best–first search (BFS) algorithm that traverses the *k* longest near–critical paths in order of nonincreasing duration. The algorithm begins by evaluating all outgoing edges from root vertices. The edge with minimum total slack is selected. The critical path method guarantees that at least one of these edges will be on a critical path and have zero total slack. Once a path has been selected, traversal is an iterative process of following the edges with minimum total slack at each descendant vertex. When a leaf vertex is reached, the next longest path is selected for traversal.

Traditionally, the applicability of BFS has been limited by an exponential memory requirement [12]. The memory is needed to save the state of all partially explored paths so that optimal selections can be made. Slack values provide the information needed to overcome this limitation. Since slack is a global criticality measure, storage can be constrained to maintaining state for the *k* longest near–critical paths that have been found. To maintain this state information, partial paths encountered during near–critical path traversal must be evaluated. Partial paths are formed by edges that are not on the current near–critical path. Partial path evaluation is based on the cost function (*FS_sum + TS)*, and state is maintained for the minimum cost near–critical paths.

To minimize path evaluation overhead, path costs are maintained in a max–heap data structure (see [13] for a discussion of heaps). This allows direct access to the maximum cost partial path and a new (lower) maximum can be established in logarithmic time. To minimize the overhead of selecting the next longest path, path costs are also maintained in a min–heap. When the max–heap is modified by sifting down a new entry, the associated min–heap entry is percolated up to maintain the integrity of the dual heaps. Thus, the minimum cost partial path is always available at the top of the min–heap. Advanced data structures such as min–max heaps [14] and deaps [15] can provide similar double–ended functionality without duplicating the storage. We opted to trade a 17% increase in the storage component dependent upon *k* for the simplicity and lower overhead of traditional heaps.

Path state information is preserved in *path_descriptor* records. Pointers to the descriptors of edges on near–critical paths are recorded in *path_entry* records. Paths consist of two segments. The first segment of a path contains edges shared with the (parent) near–critical path that was being traversed when the partial path was formed. These edges begin at a root vertex. When a partial path is formed, information about the preceding segment is saved in the *path_descriptor.* This information includes a count indi-

cating the number of edges on the first path segment, *path_1_cnt*, and a pointer to the *path_descriptor* of the parent path, *path_1_p*. The second path segment consists of a linked–list of *path_entry* records. The first *path_entry* record for the second path segment, *path_2*, is also contained in the *path_descriptor.* The second path segment is constructed during near–critical path traversal and terminates at a leaf vertex.

A pointer to the *path_entry* record corresponding to the minimum cost path from each vertex is maintained in the associated adjacency list head. This *longest_path* pointer is saved at the first visit to each vertex. The *longest_path* pointers allow additional *path_entry* record sharing. If, during near–critical path traversal, a vertex is reached that has already been visited by an earlier traversal, then all succeeding edges are shared with the earlier path. In such a case, no further *path_entry* records are allocated; instead, the *longest_path* pointer is used as a link to the existing *path_entry* records. Duplicate *path_entry* records are required only when the same edge begins the second segment of near–critical paths, which can occur a maximum of *k/2* times. Therefore, the worst–case memory requirement for the algorithm is in *O(k+e)*. Fig. 2 provides an illustration of the path description data structures for the graph in Fig. 1.

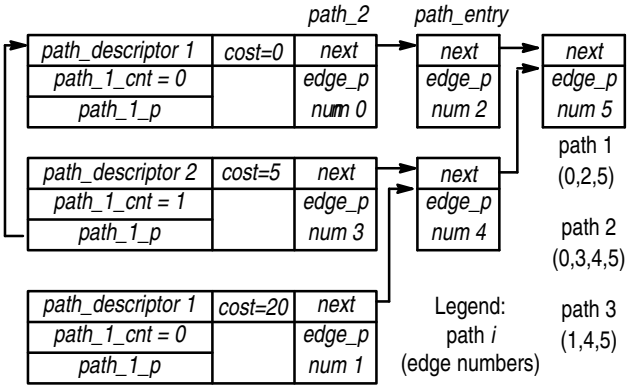


Fig. 2. BFS path description data structures.

Pseudo–code for the algorithm, assuming the augmented critical path method data structures, is given below.

Best–first search algorithm

```
Sort adjacency lists so that entries have nondecreasing total
slack values (to expedite traversals)
path_cnt := 0        // incremented when paths are evaluated
complete_paths := 0
FS_sum := 0
for each root vertex v
  effective_out_degree := 0
  for each outgoing edge from v
    evaluatePath(edge,effective_out_degree)
    effective_out_degree := effective_out_degree + 1
while (path_cnt – complete_paths) ≠ 0
  Remove path_descriptor of next longest path from
  root of min–heap
  traversePath(path_descriptor)
  complete_paths := complete_paths + 1
```

```
procedure traversePath(path_descriptor)
  Record edges on first path segment in edge_num_buf
  by following parent path_descriptor chain
  Initialize FS_sum from path_descriptor
  path_entry_p := address(path_descriptor.path_2)
  edge_p := path_entry_p–>edge_p
  path_complete := FALSE
  while path_complete ≠ TRUE
    if edge_p–>t has not been visited
      A[edge_p–>t].longest_path := path_entry_p
    if edge_p–>h is a leaf vertex
      path_complete := TRUE
    else
      if edge_p–>h has already been visited
        path_entry_p–>next :=
            A[edge_p–>h].longest_path  // set link to edges
        Traverse remainder of path to record edges #'s and
        evaluate partial paths, but do not allocate additional
        path_entry records
        path_complete := TRUE
      else
        FS_sum := FS_sum + edge_p–>FS
        edge_p :=
          address(min cost outgoing edge from edge_p–>h)
        effective_out_degree := 1
        for all but minimum cost outgoing edge
          evaluatePath(edge,effective_out_degree)
          effective_out_degree := effective_out_degree + 1
        path_entry_p–>next :=
                              address(allocated path_entry)
        path_entry_p–>edge_p := edge_p       // insert edge
        Store edge_p–>num in edge_num_buf
  path_duration :=
            critical_path_duration – path_descriptor.cost
  Output path (using contents of edge_num_buf)
  return

macro evaluatePath(edge,effective_out_degree)
// edge is descriptor of leading edge on second path segment
  Evaluate cost of partial path, (FS_sum + edge.TS),
  relative to max_ncp_slack & costs of previous paths
  if path is among k longest near–critical paths found
    initialize path_descriptor record & update heaps
  else
    A[edge.t].out_degree :=
        effective_out_degree  // reduce effective out–degree
    terminate path evaluation loop
```

Factors related to sorting the adjacency lists, traversing the paths, and maintaining the heaps must be considered for a worst–case performance analysis. If the maximum out–degree is bounded by a constant upper limit, the time required for the sort is in *O(n)*. In the worst case, the number of edges that will have to be examined during each of the *k* near–critical path traversals is in *O(e)*, yielding a term in *O(ke)*. The worst–case requirement associated with the heaps is in $O(k\lg(k)+e\lg(k))$; the first term is due to construction/maintenance of the min–heap, and the second term is due to maintenance of the max–heap. The max–heap is built, in *O(k)* time, when the *k'th* near–critical path is identified; thereafter, insertion of a new partial path requires ejection of an existing partial path entry. When a partial path is ejected, no further paths with the same lead-

ing edge on the second path segment will be inserted, which implies that the maximum number of replacements is in *O(e)*. If $\lg(k) \leq e$, the worst–case time complexity of the algorithm is in *O(ke)*.

Another BFS algorithm for finding the *k* longest paths of acyclic digraphs was described in [16] and subsequently revised in [17]. The algorithms were developed in the context of semiconductor timing analysis and would need to be modified for use with multigraphs. The worst–case time and memory complexities of the revised algorithm are in *O(ke*lg*(e))* and *O(ke)*, respectively.

E. Algorithm comparison

Asymptotic upper bounds on the worst–case time and memory requirements for the four near–critical path algorithms are summarized in Table I.

TABLE I
Worst–case complexities of near–critical path algorithms

Algorithm	Enumeration	Longest Paths	BnB	BFS
Time	*O(kne)*	*O(ke)*	$O(1.62^n e)$	*O(ke)*
Memory	$O(kn^2+e)$	*O(kn+e)*	*O(e)*	*O(k+e)*

One advantage of the path enumeration algorithm is the capability to incrementally explore the next longest path until sufficient data is available. The BFS algorithm can be used similarly, but is constrained to a maximum of *k* paths. This capability is potentially useful in an interactive performance analysis environment. The utility of the extended longest path algorithm is limited by memory requirements; however, parallel implementations may be able to exploit the increased granularity. Uncertainty differentiates the BnB and BFS algorithms. With BnB, the uncertainty is associated with algorithm execution time; with BFS, the uncertainty is associated with the near–criticality percentage of the *k'th* longest path. The BFS algorithm represents a good compromise between efficiency and functionality for the analysis of program activity graphs. The significance of the BFS algorithm is in the combination of time and memory requirements. For the problem of finding the *k* longest paths of acyclic, directed multigraphs, the algorithm is worst–case asymptotically optimal in terms of both execution time and memory usage (see [18] for the proof).

IV. Probe acquisition and PAG construction

A. MSPARC multicomputer

The traces used in this study were collected with the instrumentation facilities of the MSPARC multicomputer [19]. The MSPARC is an 8–node multicomputer based on Sun SPARCstation 2 processing boards. The nodes are organized as a mesh, and the interconnection is via wormhole routers. Each node is equipped with an intelligent performance monitor adapter that provides an interface to a separate data collection network.

B. Object–Oriented Fortran environment

The Object–Oriented Fortran (OOF) programming environment [20] is used to develop parallel applications for the MSPARC multicomputer. OOF was developed to provide a parallel environment familiar to applications engineers that would support some level of data encapsulation and be portable across a range of MIMD architectures. OOF provides extensions to standard Fortran77 that support declaration of object classes with their associated data and methods. Specific instances of these object classes may then be created dynamically at run time. Objects can be created on different processors to support parallel execution. Messages are passed between objects using a remote procedure call type syntax, but execution of the invoking object continues immediately after the message is sent. This technique isolates the user from the underlying message passing primitives. OOF is currently implemented on networks of workstations, the Intel iPSC/860 and Delta, the Silicon Graphics Power Iris, and the MSPARC multicomputer.

C. MSPARC instrumentation system

Hardware, software, and hybrid measurement systems have been used to record event traces. Hardware instrumentation is unobtrusive and delivers useful low–level information, but is costly and provides information with limited context. Software instrumentation is simple and flexible, but can perturb the execution characteristics of the program being measured. Hybrid measurement systems combine software with hardware support and provide an attractive compromise [21]. The MSPARC instrumentation system implements a hybrid approach. Special hardware on the performance monitor adapter collects and timestamps information written by software probes from the OOF environment. All processing of probes is done by the instrumentation processor, so the only obtrusiveness comes from the actual writing of the probe data. This overhead has been measured to be approximately two microseconds per probe. Table II shows the types of probes that are implemented on the MSPARC.

The MSPARC instrumentation system offers two modes of operation: performance data may be viewed via a suite of real–time graphical displays, or recorded to disk for postmortem analysis. A global timestamp clock shared by the performance monitor adapters allows for a total ordering of events collected from all nodes. Recorded probes are converted to the Pablo Self–Defining Data Format (SDDF) for the purpose of PAG generation.

D. Construction of program activity graphs

The primary probes of interest in PAG generation are *operator execution*, *message send*, *end send*, *message receive* and *receive overhead*. PAGs contain one root vertex for each node involved in the program execution. All vertices have a single child except those that mark the beginning of a remote *message send*, which have two children. One child is associated with the following event on the same node, and the other child marks the beginning of the

associated *operator execution* on the destination node. The duration of the edge to the remote node is the difference between the end of message reception time at the destination node and the start of message transmission time at the source node, and thus takes into account effects such as network congestion.

To construct PAGs, several types of probes must be matched. For example, each *message send* must be matched with the corresponding *message receive* on the destination node, and each *message receive* must be matched with the corresponding *operator execution* that results from the message receipt. The graph construction program matches corresponding events by managing three types of event queues. The first is the operator execution list. There is one operator execution list per node, and each is ordered by time. The second type of list is the message receive list. Two different types of message receive lists are kept. The first type of list is used to match *message receives* with the corresponding *operator executions*; therefore, one list for each node is kept in order by timestamp. The second type of receive list is used to match *message sends* and *message receives*; to reduce the complexity of searching for matching *message receives*, a separate list is maintained for each source–destination pair. The third type of event queue contains all *receive overhead*, *message send*, and *end send* events. Again, there is one such list per node, and the lists are ordered by timestamp. Each of the lists can be constructed in linear time as the probes from the MSPARC are in order by time for a particular node and type of event.

Once the event queues have been built, the *message receives* and *operator executions* on each node are matched and numbered so that subsequently, when a *message send* is encountered, the numbers of its children vertices are known. The graph is then generated by processing the operator execution lists and inserting *message send* and *receive overhead* events as they occur in time. This construction process can also be accomplished in linear time with one pass through the event lists. Edge information is output as it is generated. A sample PAG for a two node case is shown in Fig. 3.

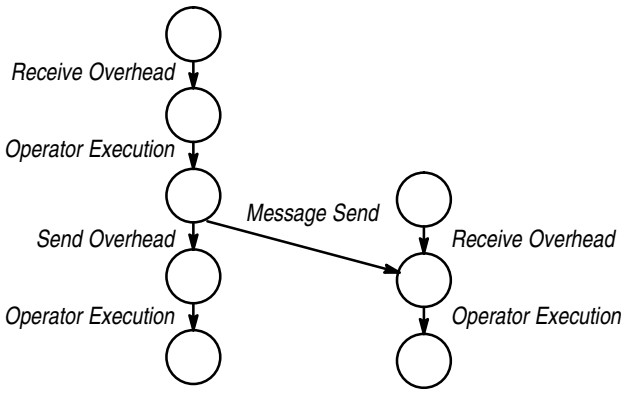


Fig. 3. Sample program activity graph.

TABLE II
MSPARC probe types

Probe Type	Meaning
Program Start	Start of program execution
Object Creation	Creation of an object
Operator Execution	Indicates start/end times for method execution
Message Send	Beginning of message transmission, identifies operator to be invoked as result of the message
Message Receive	Completion of message reception, the message is then queued for the destination object
Idle Time	Beginning and end of idle periods for a node
Object Destruction	Destroy command executed for specified object
Program End	End of program execution on all nodes
Receive Overhead	Indicates start and end times for execution of asynchronous message receive routine
Send End	Completion of message transmission overhead at source node

V. Algorithm performance results

A. Application programs

Five application programs were traced: a Quicksort of 1000 integers (QSORT), two of the NAS parallel benchmarks [22], and two computational field simulation (CFS) codes. The NAS benchmarks were the Embarrassingly Parallel kernel and the Pentadiagonal Solver application. The CFS codes were Julianne [23] and Turbomachinery [24], both of which are mature applications. All of the traces are from 8–node executions.

The Embarrassingly Parallel (EP) kernel provides an estimate of the upper achievable limits for floating–point performance. Minimal communication is involved. The kernel generates pairs of Gaussian random deviates and tallies the number of pairs in successive square annuli.

The Pentadiagonal Solver (SP) application produces solutions for coupled non–linear partial differential equations. In each pseudo–time stepping iteration, xi, eta and zeta directional sweeps are performed. The xi–directional sweep requires communication among different objects. Twenty–five iterations of the SP application were traced.

The Julianne (JUL) code solves inviscid fluid dynamics problems using an implicit finite–volume scheme (first, second or third order) with ROE averaged flux computation, either local or minimum time stepping, and numerical jacobians. Parallelization is achieved via domain decomposition and passing the Q variables across block boundaries after each iteration. The results shown here are from 100 iterations of a run on a m6c onera wing with a grid size of 49x9x9.

The Turbomachinery (TURBO) code solves for the unsteady flow about turbomachines using the three–dimensional time–dependent Euler equations. This application also uses domain decomposition for parallelization; how-

ever, the decomposition is complicated by the fact that there is relative motion between domains and therefore, the communication pattern changes throughout execution, leading to dynamic decisions about the distribution of data between iterations. Two hundred iterations of the TURBO code were traced.

Table III summarizes the application–related statistics. Note that the programs provide a broad range of execution times and communication characteristics, as well as marked variations in the number of edges and vertices in the PAGs.

TABLE III
Application–related statistics

Program	Execution Time (s)	No. of Edges	No. of Vertices	Probes/ Second	CP_{com}[a]
QSORT	0.274	1097	953	5,945	84.3%
EP	8.170	198	175	34	0.3%
SP	14.838	138,854	115,909	12,276	57.6%
JUL	382.939	20,203	18,567	65	2.7%
TURBO	1,822.800	261,776	243,447	210	1.5%

[a]Percent of critical path duration devoted to communication.

B. Performance data

The BnB and BFS algorithms were implemented in the C programming language and compiled by the Sun C compiler at optimization level 4. Reported algorithm performance results are from experiments conducted on a SPARCstation 2. The SPARCstation was equipped with 16 Mbytes of memory and running SunOS 4.1.3. No other user jobs were active during the tests.

To quantify the effectiveness of the BnB technique, the pruning percentage performance metric was defined to be the percentage of edges that are not examined due to BnB. Pruning percentages for Quicksort and EP are plotted in Fig. 4. Although the BnB algorithm is clearly effective in pruning the searches, large numbers of near–critical paths were found, leading to long execution times. Fig. 5 shows how the number of near–critical paths dramatically increases as a function of near–criticality percentage for the QSORT PAG, which is relatively small. The situation is even worse for larger, more realistic PAGs. For example, the BnB algorithm was executed on a Cray Y–MP using the SP PAG and a near–criticality percentage of 1, and the program was terminated without completing after 24 hours of CPU time. Another experiment with the QSORT PAG showed comparable performance for the BnB and BFS algorithms when finding the same number of near–critical paths. The key is to know the appropriate near–criticality percentage. The BFS algorithm eliminates this uncertainty.

The graph in Fig. 6 shows the relationship between execution time of the BFS algorithm and the number of near–critical paths found. This graph is plotted on a log–log scale because of the disparity in the times for the different applications. The similar shape of the curves is significant; a simple execution time model of the form c_1e+c_2ke was able to predict the algorithm's performance within ~25%. The first term in the model represents the time to read the input multigraph and perform the critical path method, the second term represents the BFS, and the constants are implementation dependent. The near–criticality percentage associated with the 100,000'th longest path was 99.99% for the SP, JUL, and TURBO PAGs.

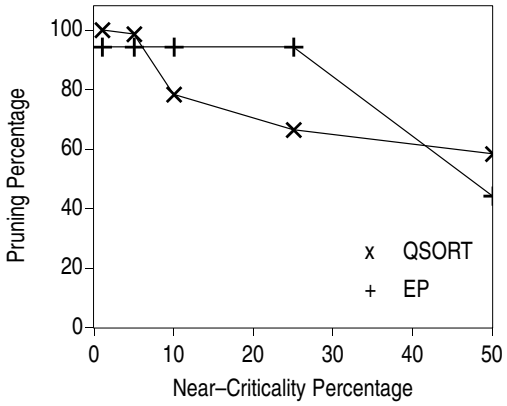


Fig. 4. Effectiveness of BnB technique.

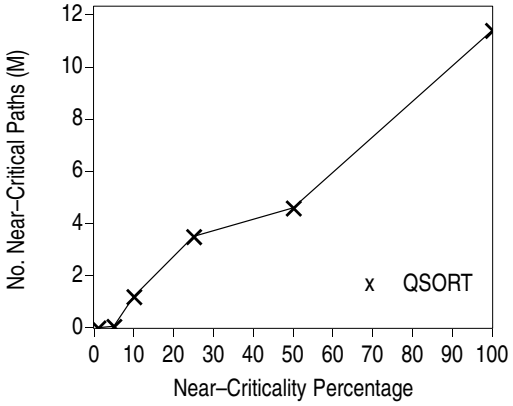


Fig. 5. Near–critical path explosion.

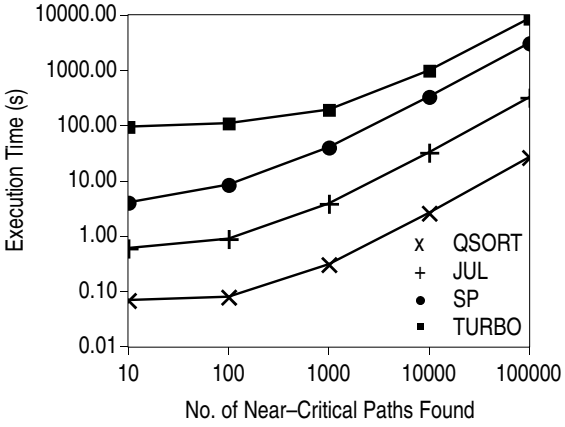


Fig. 6. Execution time of BFS algorithm.

VI. Near–critical path analysis framework

The output from the near–critical path programs consists of a list of all the critical and near–critical paths found. Each path consists of a duration and an edge list. This information by itself is not meaningful to the user as the relationships between the edges listed and program activities is not known. In any case, a list of all the program activities on the near–critical paths would probably contain too much information to be useful. Near–critical path analysis will attempt to provide both guidance via hierarchical summaries expressed in terms of logical events within the application program, and capabilities flexible enough to support detailed exploration of small–scale behavior.

At the highest level, only the critical paths are analyzed. Classical metrics such as computation and communication percentages are provided. Activities may be viewed from a processor perspective or broken down by operator. Near–critical path activity classes are represented by a new performance metric that considers contributions across all found paths. The availability of near–critical path data permits prediction of the maximum performance improvement that may be achieved by optimizing a particular critical path activity. But more importantly, the broader perspective allows guidance to be offered regarding the relative merits of tuning specific activities.

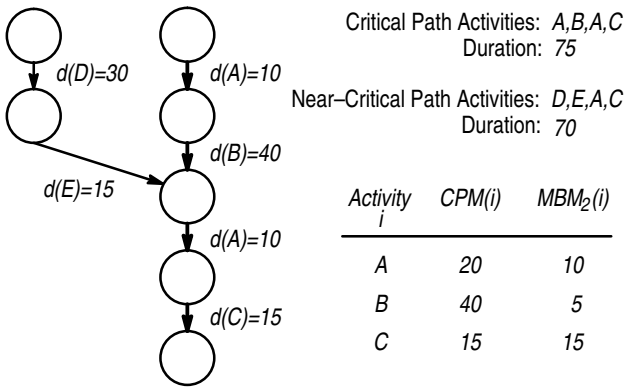
The goal of performance debugging metrics is to rank the importance of improving specific program activities. The Critical Path Metric (CPM) ranks activities according to the magnitude of their durations on the critical path. The *Maximum Benefit Metric (MBM)* is an extension of the Critical Path Metric that includes the synergistic effects of common activities on near–critical paths. The Maximum Benefit Metric for activity i over the k longest paths is computed as follows:

$$MBM_k(i) = \min(d(i)_j + (d_{cp} - d_j)) \quad \text{for } j = 1 \text{ to } k,$$

where

$d(i)_j$ = aggregate duration of activity i on j*'th* longest path,
d_{cp} = duration of the critical path, and
d_j = duration of the j*'th* longest path.

Fig. 7 contains a simple example that illustrates how optimizing the largest component on the critical path may not yield the most overall improvement. The *MBMs* for the computation and communication components of the five benchmark applications are plotted in Fig. 8 and Fig. 9. The metrics are expressed as percentages of the critical path duration. Near–critical path data clearly reveals additional performance characteristics. Several of these applications show that little additional information is gained by using more than 100 near–critical paths.



Activity i	CPM(i)	$MBM_2(i)$
A	20	10
B	40	5
C	15	15

Fig. 7. Example of MBM calculation.

The computation to communication ratio can be used to assess the appropriateness of the application decomposition. A high communications contribution to the critical path could indicate an inappropriate, or too finely grained decomposition. Near–critical path data can also be used as an architecture evaluation tool. A high communications contribution on all critical and near–critical paths can indicate that increased interconnection network performance would result in improved application performance.

The availability of PAGs facilitates speculation about the effects of reducing the time associated with a particular activity. The availability of near–critical path data facili-

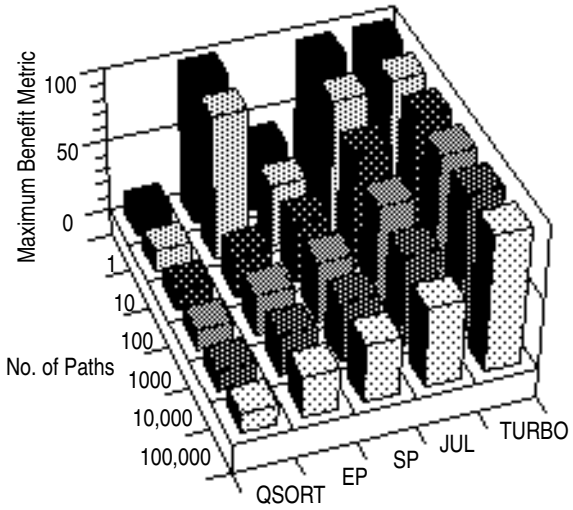


Fig. 8. Computation MBMs.

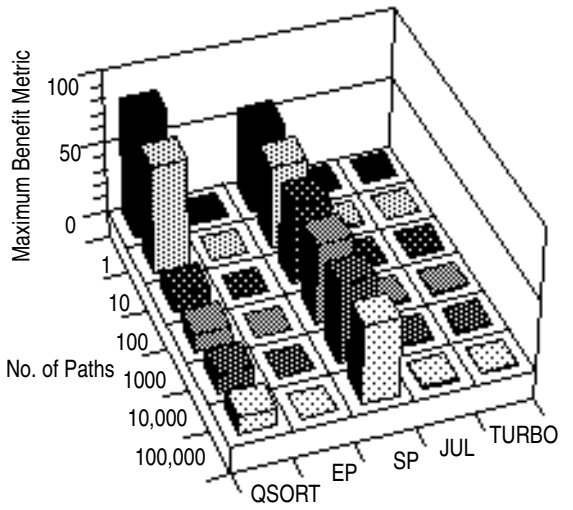


Fig. 9. Communication MBMs.

tates selection of the most promising activities for *what if* scenarios. The envisioned environment supports rapid experimentation by allowing the durations of selected PAG activities to be adjusted. The potential effects are then quickly ascertained via near–critical path analysis of the modified PAG. While near–critical path guidance is based on a limited number of paths, *what if* scenarios extend the analysis to all execution paths.

Visualization complements the statistical perspective by revealing the dynamics of *when* performance determining activities occurred. Rather than attempt the impossible task of predicting and satisfying all potential visualization needs, we have opted to simply output Pablo SDDF records corresponding to critical and near–critical path activities. The records contain event and node identifiers, event specific descriptors, timestamps, and counts indicating the number of critical and near–critical path occurrences. In this manner, the full capabilities of the Pablo environment may be invoked to explore critical and near–critical path activities from the most appropriate perspectives.

VII. Conclusion

As the availability of parallel computing resources becomes more common, the practical importance of effective program optimization techniques increases. Consequently, the suite of available performance analysis tools is evolving, and algorithmic advances are an important component of the emerging solutions. In this paper, we have described an efficient new algorithm for finding the k longest paths of directed, acyclic multigraphs. The algorithm utilizes slack values calculated by the critical path method to perform a best–first search in linear space, and for a specific k, the time complexity of the algorithm is also linear in the problem size. The algorithm can be used in conjunction with program activity graphs constructed from timestamped traces to identify the activities on the k longest execution paths. This information forms the basis for a new addition to the suite of available performance analysis tools by offering a broader perspective for focusing optimization efforts. Experiments with five parallel applications have verified the efficiency and practicality of the algorithm.

We have also described a framework for near–critical path analysis of program activity graphs that builds upon the foundation of existing performance analysis tools. Near–critical path data complements proven techniques by providing additional insight into the dynamic behavior of the system. Further research is needed to validate and refine the proposed framework through actual experience. Our plans include development of a parallel near–critical path algorithm as a case study.

References

[1] B. P. Miller and C.–Q. Yang, "IPS: An interactive and automatic performance measurement tool for parallel and distributed programs," in *Proc. 7th Int. Conf. Distrib. Computing Syst.*, IEEE Comput. Soc., Sept. 21–25, 1987, pp. 482–489.

[2] D. A. Reed *et al*., "The Pablo performance analysis environment," Tech. Rep., Dep. of Comput. Sci., Univ. of Illinois, Nov. 1992.

[3] B. P. Miller *et al*., "IPS–2: The second generation of a parallel program measurement system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 2, pp. 206–217, Apr. 1990.

[4] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Inform. Processing Lett*., vol. 11, no. 1, pp. 1–4, Aug. 1980.

[5] C. Q. Yang and B. P. Miller, "Performance measurement of parallel and distributed programs: A structural and automatic approach," *IEEE Trans. Software Eng*., vol. 15, no. 12, pp. 1615–1629, Dec. 1989.

[6] J. D. Wiest and F. K. Levy, *A Management Guide to PERT/CPM*. Englewood Cliffs, NJ: Prentice–Hall, 1977.

[7] J. E. Hopcroft and R. E. Tarjan, "Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jan. 1973.

[8] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Rockville, MD: Computer Science Press, 1983.

[9] E. Rich, *Artificial Intelligence*. New York: McGraw–Hill, 1983.

[10] W. Zhang and R. E. Korf, "An average–case analysis of branch–and–bound with applications: Summary of results," in *Proc. 10th Nat. Conf. AI*, AAAI Press, July 12–16, 1992, pp. 545–550.

[11] C.–Q. Yang and B. P. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int. Conf. Distrib. Computing Syst.*, IEEE Comput. Soc., June 1988, pp. 366–375.

[12] R. E. Korf, "Linear–space best–first search: Summary of results," in *Proc. 10th Nat. Conf. AI*, AAAI Press, July 12–16, 1992, pp. 533–538.

[13] G. Brassard and P. Bratley, *Algorithmics Theory and Practice*. Englewood Cliffs, NJ: Prentice–Hall, 1988.

[14] M.D. Atkinson, J.–R. Sack, N. Santoro, and T. Strothotte, "Min–max heaps and generalized priority queues," *Commun. ACM*, vol. 29, no. 10, pp. 996–1000, Oct. 1986.

[15] S. Carlsson, "Deap – A double–ended heap to implement double–ended priority queues," *Inform. Processing Lett*., vol. 26, no. 1, pp. 33–36, Sept. 15, 1987.

[16] S.H. Yen, D. H. Du, and S. Ghanta, "Efficient algorithms for extracting the k most critical paths in timing analysis," in *Proc. 26th ACM/IEEE Design Automation Conf.*, June 25–29, 1989, pp. 649–654.

[17] Y.–C. Ju and R. A. Saleh, "Incremental techniques for the identification of statically sensitizable critical paths," in *Proc. 28th ACM/IEEE Design Automation Conf.*, June 17–21, 1991, pp. 541–546.

[18] C. A. Alexander, "Near–critical path algorithms for program activity graphs," Ph.D. dissertation, Dept. of Comput. Engr., Mississippi State Univ., May 1994.

[19] J. C. Harden *et al*., "A performance monitor for the MSPARC multicomputer," in *Proc. IEEE Southeastcon'92*, Apr. 12–15, 1992, pp. 724–729.

[20] D. Reese and E. Luke, "Object oriented fortran for development of portable parallel programs," in *Proc. 3rd IEEE Symp. Parallel and Distrib. Syst.*, Dec. 2–5, 1991, pp. 608–615.

[21] A. Mink, R. Carpenter, G. Nacht, and J. Roberts, "Multiprocessor performance measurement instrumentation," *IEEE Computer*, pp. 63–75, Sept. 1990.

[22] D. Bailey, J. Barton, T. Lasinski, and H. Simon, ed., "The NAS parallel benchmarks," Tech. Rep. RNR–91–002, NASA Ames Research Center, Aug. 1991.

[23] D. L. Whitfield and J. M. Janus, "Three dimensional unsteady euler equations solutions using flux vector splitting," *17th Fluid Dynamics, Plasma Dynamics, and Lasers Conf.*, AIAA paper no. 84–1552, June 25–27, 1984.

[24] G. J. Henley and J. M. Janus, "Parallelization and convergence of a 3D implicit unsteady turbomachinery flow code," in *Proc. 5th SIAM Conf. Parallel Processing for Scientific Computing*, March 25–27, 1991, pp. 238–245.