Proceedings of the First Southern Symposium on Computing
The University of Southern Mississippi, December 4-5, 1998

NEAR-CRITICAL PATH ANALYSIS: A TOOL FOR PARALLEL PROGRAM OPTIMIZATION

CEDELL A. ALEXANDER[*], ARIC B. LAMBERT[†], DONNA S. REESE[†], JAMES C. HARDEN[†]
AND RON B. BRIGHTWELL[‡]

Abstract. Program activity graphs (PAGs) can be constructed from timestamped 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 PAGs are compared. A framework for using the near-critical path information is also described. The framework includes statistical summaries and visualization capabilities that build upon the foundation of existing performance analysis tools. Within the framework, guidance is provided by the *Maximum Benefit Metric,* which uses near-critical path data to predict the maximum overall performance improvement that may be realized by optimizing particular critical path activities.

1. Introduction. Developing efficient parallel programs has proven 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 through 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. 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 initial focus of this paper is on efficient algorithms 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).

We also present a framework for using near-critical path data that encompasses both statistical summaries (patterned after IPS) and the visualization capabilities of Pablo. Guidance is provided by the *Maximum Benefit Metric,* which includes the synergistic effects of common activities on near-critical paths

[*] IBM' s Networking Hardware Division, P.O. Box 12195, Research Triangle Park, NC 27709.
[†] NSF Engineering Research Center for Computational field Simulation, Mississippi State University, P.O. Box 6176, Mississippi State, MS 39762.
[‡] Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1110.

to predict the maximum overall performance improvement associated with optimization of particular critical path activities.

In Section 2, critical path algorithms are reviewed to provide the background needed for description of near-critical path algorithms in Section 3. Probe acquisition and construction of program activity graphs are discussed in Section 4. A framework for near-critical path analysis is presented in Section 5. Section 6 contains the description of the applications and performance results from the Maximum Benefit Metric. The paper is concluded in Section 7 with a summary of key results.

2. Critical Path Algorithms.

2.1. Program Activity Graphs. A program activity graph (PAG) *is* an acyclic, directed multigraph representing the duration and precedence relationships of program activities. Edges represent execution activities, weights represent activity durations, vertices mark activity boundaries, and 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, generality requires that near-critical path algorithms accommodate multigraphs (PAG characteristics are determined by the semantics of the target system). The biggest impact of the multigraph characteristic is on data structure selection.

2.2. Longest Path Algorithm. IPS employs a modified shortest path algorithm, based on the diffusing computation paradigm [3], 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. A version of the synchronous algorithm with adaptations to accommodate multigraphs is given in [4].

2.3. Critical Path Method. The critical path method is an operational research algorithm for finding the longest path(s) through an activity-on-edge network [5]. The critical path method calculates early start and earl) 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. Table 1 defines the terms that will be used to explain the algorithm.

Notation	Definition
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)*

TABLE 1. *Critical Path Method Notation*

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 affecting 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 affecting 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. The values calculated by the critical path method for a simple example network are shown in Fig. 1.

2.4. Algorithm Comparison. The longest path algorithm is more efficient than the critical path method (since the longest path is found in a single pass through the edges). 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 0*(e),* where *e* is the number of edges in the graph. Selection of the most appropriate algorithm is dependent upon application needs.

3. Near-Critical Path Algorithms.

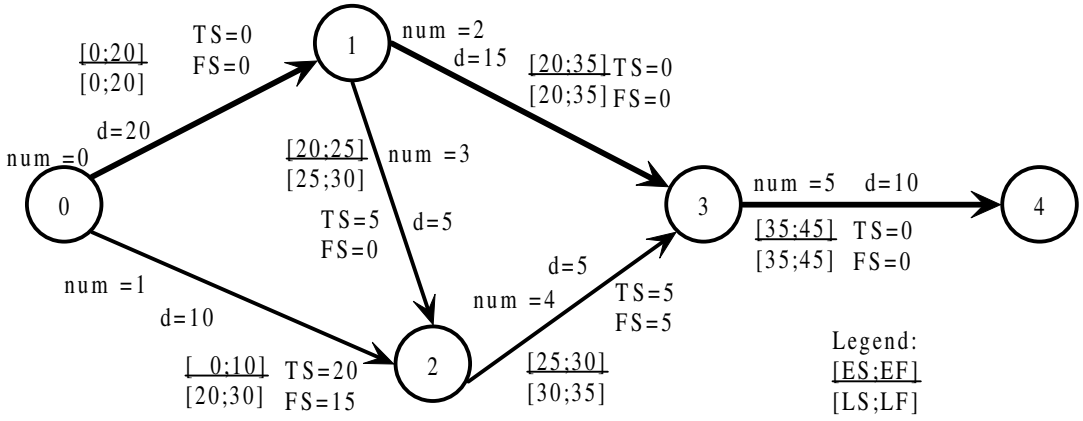


FIG. 1. *Critical path method example.*

Definition 1*:* A *near-critical path* is a path whose duration is within a certain percentage, the *near-criticality percentage,* of the critical path duration. The near-criticality percentage (denoted *nc%)* may be specified by the user or reported by the algorithm. Three near-critical path algorithm approaches are summarized in the following list:

1) Specify maximum number of longest paths to find, *k*, and report *nc%* of k^{th} longest path,
2) Specify *nc%* and find all near-critical paths.
3) Specify both *k* and *nc%* (i.e., find up to *k* longest near-critical paths).

In this section, four near-critical path algorithms are compared: the path enumeration and extended longest path algorithms are examples of approach 1); the branch-and-bound algorithm is based on approach 2); and the best-first search algorithm employs approach 3). Approach 3) can be advantageous, relative to approach 1), when the number of near-critical paths is less than *k*.

3.1. Path Enumeration and Extended Longest Path Algorithms. An algorithm for listing the *k* shortest paths between two vertices of an acyclic digraph is described in [6]. The algorithm can be easily modified to enumerate longest paths. For a multigraph containing *n* vertices and *e* edges, the worst-case time and memory requirements of the algorithm are in *O(kne)* and $O(kn^2+e)$, respectively.

A more straightforward approach is to simply extend the longest path algorithm to find the *k* longest paths as described in [4]. Since the extended algorithm maintains an array of *k* (fixed-size) path description records for each vertex, and a descriptor is required to represent each edge, the storage requirements are in *O(kn+e).* The worst-case time complexity of the algorithm is in 0*(ke).*

3.2. Brunch-and-Bound Algorithm. Brute-force depth-first searches can solve the longest path problem in linear space; however, the time complexity is exponential [7]. Branch-and-bound (BnB) is a technique that may significantly improve the efficiency of depth-first searches by eliminating unproductive search paths [8]. In this subsection, we show how the slack values calculated by the critical path method can be used as the basis for a BnB near-critical path algorithm. The notation employed to explain the algorithm is defined in Table 2.

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 *(FS_sum* = 0 for a critical path).

Notation	Definition
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 current path at any edge of a depth-first search, *critical_path_duration*-(*FS_sum* + *TS*)
max_ncp_slack	Maximum slack of near-critical path, *critical_path_duration-min_ncp_duration*

TABLE *2. Near-Critical Path Notation*

The performance of the algorithm is highly dependent upon the input PAG. In the best case, the time complexity is in $0(1)$. If we optimistically assume that only one edge exists between any two vertices and that no vertex has more than two outgoing edges (which is true for the PAGs that we generate), the worst-case complexity, based on the number of edges that must be examined, is in $0(1.62^n)$. When the critical path method is also included in the analysis, the best-case and worst-case time complexities are in $0(e)$ and $0(1.62^n+e)$, respectively.

3.3. 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 edge 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 [9]. 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. 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.

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 indicating 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 a vertex is saved at the first visit to each vertex to 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. 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. 1 provides an illustration of the path description data structures for the graph in Fig. 2.

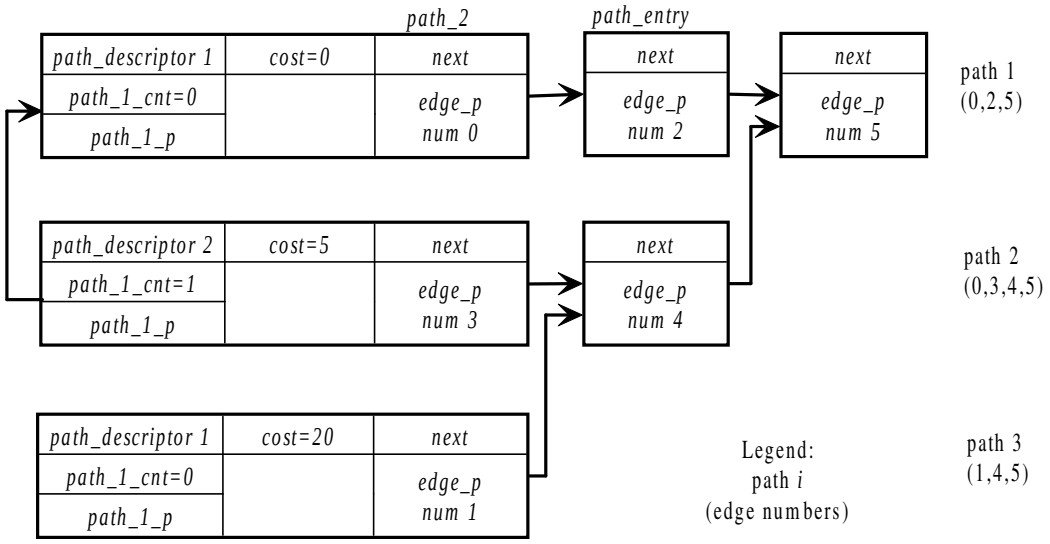


FIG. 2. *BFS path description data structures.*

The worst-case time complexity of the algorithm is in 0*(ke),* with the dominant factor being that 0*(e)* edges may need to be examined during each of the *k* near-critical path traversals. A detailed analysis of the algorithm, along with proofs of correctness and worst-case optimality can be found in [10l (worst-case optimality is established in terms of both time and space for the problem of enumerating the *k* longest paths of acyclic, directed multigraphs).

Algorithm Comparison. Asymptotic upper bounds on the worst-case time and memory requirements for the four near-critical path algorithms are summarized in Table 3.

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)*

TABLE 3. *Worst-Case Complexities Of Near-Critical Path Algorithms*

One advantage of the path enumeration algorithm is the capability to incrementally explore the next longest path until sufficient data is available, which is potentially useful in an interactive environment. The BFS algorithm can be used similarly, but is constrained to a maximum of *k* paths. Memory requirements limit the utility of the extended longest path algorithm. Uncertainty differentiates the BnB and BFS algorithms. With BnB, the uncertainty is associated with execution time; with BFS, the uncertainty is associated with the near-criticality percentage of the k^{th} longest path. The significance of the BFS algorithm is in the combination of time and memory requirements.

4. Probe Acquisition and PAG Construction.

4.1. SuperMSPARC Multicomputer and Instrumentation System. The traces used in this study were collected with the instrumentation facilities of the SuperMSPARC multicomputer [11]. The SuperMSPARC is a 32-processor machine based on the SPARCstation 10 multiprocessor. There are eight SPARCstations, each of which contains four 90 MHz Ross hyperSPARC processors. SuperMSPARC has three types of interconnection communication networks: Ethernet, ATM, and Myrinet. Each node is

equipped with an intelligent performance monitor adapter that provides an interface to a separate data collection network.

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 [12], The SuperMSPARC instrumentation system implements a hybrid approach. Special hardware on the performance monitor adapter collects and timestamps information written by software probes from the MPI environment. All processing of probes is done by the instrumentation processor, so the only obtrusiveness comes from the actual writing of the probe data, which has been measured to be ~2 microseconds per probe.

The SuperMSPARC instrumentation system records performance data 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 and visualization using a Pablo display.

4.2. Message Passing Environment. The defacto message passing standard Message Passing Interface (MPI) was chosen as the vehicle for implementation of the construction of the PAG for near-critical path analysis. The MPI standard is independent of any particular machine architecture and allows the programmer to write portable programs that can be run without changes to the underlying communication protocol [13]. Since the most important events a performance monitoring systems needs to analyze are communication events, acquisition of probe information will be done primarily within the MPI function calls.

An MPI probe library was designed with probe function calls placed at the beginning and end of each MPI function call. This allows a timestamp of the beginning and end of the MPI call to be taken so the interval of execution time of the function can be obtained. These probes were inserted by using the MPI profiling interface. The MPI profiling interface allows MPI function calls to be replaced by user-defined functions that can perform performance monitoring activities and then invoke the true MPI functions. The programmer can easily link the probe library with the application to obtain probe data without source code modification. Table 4 shows the types of MPI and additional probes that are implemented on the SuperMSPARC.

Probe Type	Overview
All Gather	All the processes distribute data to all the other processes.
All Reduce	An operation is performed on the data from all the processes. The result of the operation is obtained by all the processes.
Barrier	Each process is blocked until all the processes have called the barrier function.
Broadcast	A root process distributes data to all the other processes.
Message IRecv	A process attempts to receive data without blocking the task's execution.
Message ISend	A process attempts to send data without blocking the task's execution.
Message Receive	A process receives data while blocking the task's execution.
Message Send	A process sends data while blocking the task's execution.
Reduce	An operation is performed on the data from all the processes and result is obtained by the root.
Wait	Blocks a process until a non-blocking call is completed
Additional Probes	
Computation	Computation work being performed by the processes.
Idle Time	Idle period for processes waiting for a message

TABLE *4. SuperMSPARC Probe Types. MPI Routines Instrumented*

4.3. Construction of Program Activity Graphs. PAGs from a message passing environment 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 being sent. These vertices could have two or more children. One child is associated with the following event on the same node, and the other children mark the ending of the associated receive edge 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 (e.g. the beginning and ending of a receive call). However, the entire construction process, which is described in [4], can be performed in linear time. A sample PAG is shown in Fig. 3.

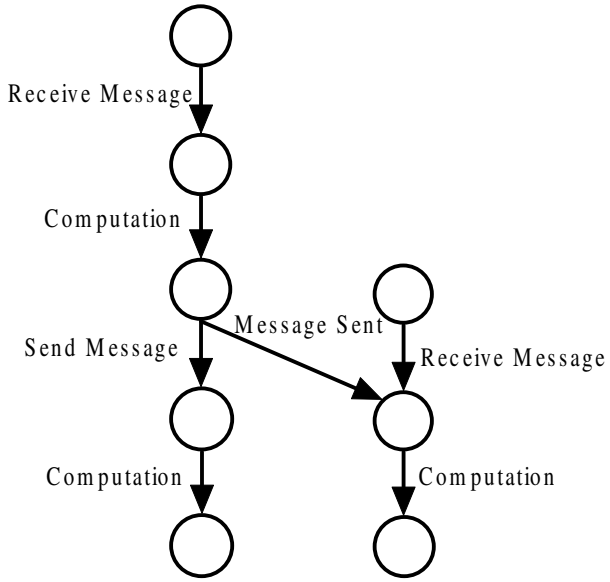


FIG. 3. *Sample program activity graph.*

5. Near-critical Path Analysis Framework. The output from the near-critical path program 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 are not known. In any case, a list of all the program activities on the near-critical paths would most likely contain too much information to be useful. Near-critical path analysis will attempt to provide both guidance through 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, the critical paths are analyzed. Classical metrics such as computation and communication percentages is provided. Activities may be viewed from a processor perspective or broken down by function. Near-critical path activity classes are represented by a new performance metric that considers contributions across all paths found. 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. More importantly, the broader perspective allows guidance to be offered regarding the relative merits of tuning specific activities.

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 facilitates selection of the most promising activities for *what if* scenarios. The analysis framework supports rapid experimentation by allowing the

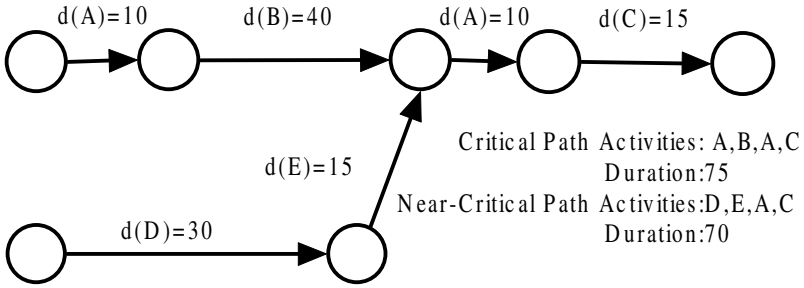
durations of selected PAG activities to be adjusted. The potential effects are then quickly ascertained by analysis of the modified PAG. While near-critical path guidance is based on a limited number of paths, *what if* scenarios extend the analysis to all execution paths.

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. 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.

The goal of performance debugging metrics is to rank the importance of improving specific program activities. Six parallel program performance metrics were compared in [14], and although no single metric was universally superior, the Critical Path Metric (CPM) provided the best overall guidance. 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:

$MBM_k(i) = min(d(i)j + (d_{cp} - d_j))$, *for* $j = i$ 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. 4 is a simple example that illustrates how optimizing the largest component on the critical path may not yield the most overall improvement. Unless all the paths are considered, which is usually not practical, the impact of the activities on the $(k+l)^{th}$ longest path are not known. Thus, the metric represents a prediction of the maximum overall benefit associated with particular critical path activities.



Activity i	Duration d(i)	Critical Path Metric MBM_1	Maximum Benefit Metric MBM_2
A	10	20	10
B	40	40	5
C	15	15	15
D	30	0	0
E	15	0	0

FIG. 4. *Communication and computation MBMs.*

Fig. 5 illustrates the aggregate MBMs for communication and computation activities of a parallel quicksort of 1000 integers. This information reveals additional clues to the application' s characteristics and behavior. MBM information indicates the need to look at as many as 100 near-critical paths to help predict the actual optimization benefit that could be obtained by optimizing communication activities. Note that the actual benefit that can be achieved is much lower than what was deduced by the critical path.

Once the MBMs over a set of near-critical paths have identified program activities of interest for optimization, *what if* scenarios can be used to recalculate the MBMs over all paths. This is accomplished by zeroing the duration of an operator in the PAG and recalculating the critical path. The MBM for activity *i* over all paths is computed as follows:

$MBM_{all}(i) = d_{cp} - d(i_0)_{cp}$,

where $d(i_0)_{cp}$ is the duration of the critical path with activity *i* zeroed, and d_{cp} is the original critical path duration.

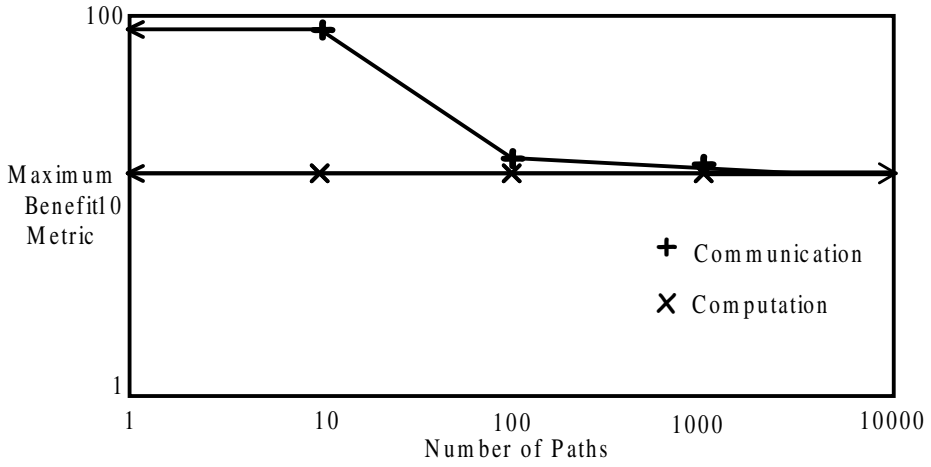


FIG. 5. *Communication and computation MBMs.*

6. Algorithms and Performance Results.

6.1. Algorithms. Algorithm performance was assessed with PAGs from five application programs: an N-body simulation application (NBODY), a Monte Carlo application (MONTE), and a ray-tracing application (ZSNOOP). NBODY simulates the evolution of a system of N bodies where the force on each body arises due to its interaction with all other bodies in the system. NBODY was designed by David W. Walker from Oak Ridge National Laboratory in Tennessee [15]. MONTE is a simple parallel implementation of an Auxiliary-Field Monte Carlo algorithm designed by Carey Huscroft at the Department of Physics, University of California at Davis [16]. ZSNOOP is a parallel ray-tracing program that uses a global combine to merge all of the images computed by the individual processors into one rendering. Lance Burton designed it at the Engineering Research Center at Mississippi State University [17]. Table 5 summarizes the application-related statistics.

Program	Execution Time (s)	No. of Edges	No. of Vertices	CP_{com}^{*}	No. of Processors
MONTE	30.301	226	285	61.2%	16
NBODY	29.294	557	448	55.2%	8
ZSNOOP	3,757.882	2131	1908	4.3%	8

TABLE 5. *Application-Related Statistics. (*Percent of critical path duration devoted to communication.)*

6.2. Performance Results. Computational performance is measured by execution time of relevant tasks. To obtain this information, probe calls are placed in delimiting points of the functional areas. Probe calls are assigned meaningful label names labels. These labels are used to identify computational performance for individual functional area.

The MBM results for MONTE in Table 6 show that function MonteCarlo has a lower percentage of possible performance benefit than was originally suggested by the critical path analysis, but that the true percentage of performance benefit is not significantly different. MBM confirmed that the critical path analysis was accurate for function Main, and since this function has a higher potential for performance benefit, the user should focus on optimizing that function.

Function Label	Critical Path %	MBM_{all} Path %
Main	21.98%	21.98%
MonteCarlo	15.20%	14.60%

TABLE 6. *MONTE Results*

The NBODY MBM analysis in Table 7 shows that possible performance benefit from function Timing is only 19.73%, where the critical path analysis showed a higher percentage of possible performance benefit of 22.97%. This indicates that the optimization of functions Timing and ParticlesInput have nearly identical expected benefits. Depending on the complexity of the individual functions, this information can enable the user to better determine which function to optimize.

Function Label	Critical Path %	MBM_{all} Path %
Timing	22.97%	19.73%
ParticlesInput	18.04%	18.02%

TABLE 7. *NBODY*

For ZSNOOP, MBM results in Table 8 show a more comprehensive estimation of expected improvement than does critical path analysis. In the original critical path analysis, the results indicated that function Zprintf accounted for a bigger percentage of the critical path than did DrawImage. The MBM indicates that optimization of function DrawImage actually has more potential benefit than that of function Zprintf, The MBM also shows a more refined percentage of possible performance gains of function Zprintf, which is 14.48 compared to 17.40 from critical path analysis.

Function Label	Critical Path %	MBM_{all} Path %
LoadTriangle	51.31%	51.31%
Zprintf	17.40%	14.48%
DrawImage	17.32%	17.25%

TABLE 8. *ZSNOOP*

7. 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, the near-critical path concept has been introduced, and an efficient new algorithm for finding the k longest paths of directed, acyclic multigraphs has been presented. The algorithm performs a best-first search in linear space. Best-first is the optimal search strategy, and the new algorithm allows best-first solution of much larger problems than previously possible. 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. We have described a multiperspective framework for near-critical path analysis of program activity graphs that builds upon the foundation of existing performance analysis tools. Near-critical path data has been

shown to complement proven techniques by revealing additional characteristics of parallel program performance.

REFERENCES

[1] B. P. MILLER AND C.-Q. YANG, "IPS: An interactive and automatic performance measurement tool for parallel and distributed programs," in *Proceedings of the 7th International Conference on Distributed Computing Systems,* IEEE Computer Society, Sept. 21-25, 1987, pp. 482-489.

[2] D. A. REED, R. A. AYDT, T. M. MADHYASTHA, R. J. NOE, K. A. SHIELDS, AND B. M. SCHWARTE, "The Pablo performance analysis environment," Tech. Rep., Department of Computer Science, Univ. of Illinois, Nov. 1992.

[3] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letter,* vol. II, no. 1, pp. 1-4, Aug. 1980.

[4] C. A. ALEXANDER, D. S. REESE, AND J. C, HARDEN, "Near-critical path analysis of program activity graphs," in *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems,* IEEE Computer Society, Jan. 31-Feb. 2, 1994, pp. 308-317.

[5] J. D. WIEST AND F. K. LEVY, *A Management Guide to PERT/CPM.* Englewood Cliffs, NJ: Prentice-Hall, 1977.

[6] E. HOROWITZ AND S. SAHNI, *Fundamentals of Data Structures,* Rockville, MD: Computer Science Press, 1983.

[7] E. RICH, *Artificial intelligence.* New York: McGraw-Hill, 1983.

[8] W. ZHANG AND R. E. KORF, "An average-case analysis of branch-and-bound with applications: Summary of results," in *Proceedings of the 10th National Conference on Al,* AAAI Press, July 12-16, 1992, pp. 545-550.

[9] R. E. KORF, "Linear-space best-first search: Summary of results," in *Proceedings of the 10th National Conference on Al,* AAAI Press, July 12-16, 1992, pp. 533-538.

[10] C. A. ALEXANDER, "Near-critical path algorithms for program activity graphs," Ph.D. dissertation, Department of Computer Engineering, Mississippi State Univ., May 1994.

[11] J. HARDEN, D. REESE, C. ALEXANDER, M. EVANS, S. KADAMBI, G. HENLEY, AND C. HUDNALL, "In Search of a Standards-Based Approach to Hybrid Performance Monitoring," in *Evaluation Tools for Parallel and Distributed Systems of Institute of Electrical and Electronics Engineers (IEEE) Parallel and Distributed Technology and Computer",* Fall, 1995.

[12] A. MINK, R. CARPENTER, G. NACHT, AND J. ROBERTS, "Multiprocessor performance measurement instrumentation," *IIIEE Computer,* pp. 63-75, Sept. 1990.

[13] W. GROPP, E. LUSK AND A, SKJELLUM, *Using MPl: Portable Parallel Programming with the Message-Passing Interface.* Massachusetts: The MIT Press, 1994.

[14] J. K. HOLLINGSWORTH AND B. P. MILLER, "Parallel program performance metrics: A comparison and validation," in *Proceedings ofSupercomputing'92,* IEEE Computer Society, Nov. 16-20, 1992, pp. 4-13.

[15] DAVID W. WALKER, Oak Ridge National Laboratory.

[16] CAREY HUSCROFT, Physics Dept., University of California, Davis.

[17] LANCE BURTON, Engineering Research Center for Computational Field Simulation, Mississippi State University.