Scheduling of Dynamic Divide-and-Conquer Computations on Multicomputers [†]

Vijay Karamcheti and Benjamin W. Wah
Center for Reliable and High Performance Computing
Coordinated Science Laboratory,
1308, West Main Street,
Urbana, IL 61801.
e-mail: wah@manip.crhc.uiuc.edu

Abstract

The scheduling of tasks for applications with dynamic behavior traditionally rely on externally observable metrics such as the number of active processes. This paper presents a new scheduling strategy based on the observation that it may be possible to capture the near-term resource requirements of active tasks by expressions involving task parameters. Run-time evaluation of these expressions yields estimates of task behavior that are valid over a short, future interval of time. The heuristics proposed, which when used in conjunction with information supplied by profiling, can be used to annotate the source program with such expressions. Preliminary simulation results show that the use of near-future estimates in a dynamic scheduling strategy for divide-and-conquer algorithms consistently improves over traditional dynamic strategies. The performance of this strategy approaches that of the best-known deterministic strategy while incurring an overhead of the same order as other dynamic strategies.

1 Introduction

The *divide-and-conquer* algorithmic technique is a powerful problem solving paradigm. There are two methods for executing a divide-and-conquer algorithm in parallel. *Static* (deterministic) approaches [1] are used when the set of tasks and their precedence relationships are known a priori. *Dynamic* strategies, which are used to solve a problem when its task requirements are not known in advance, can be further divided into two subclasses. Strategies in the first subclass address scheduling in a dedicated system, where all tasks that affect system load are controllable. The second kind of dynamic strategies are used in multitasked systems whose load is influenced by tasks and factors, some of which are beyond the control of the resource-management strategy.

The paper addresses the execution of dynamic divide-and-conquer algorithms on dedicated multicomputer systems using the first kind of dynamic strategies. These strategies either (*i*) use *no information* about the resource-requirement patterns of active tasks; (*ii*) use information contained in *externally observable metrics*, such as the number of tasks [2] and the number of unexecuted requests for resident tasks [3]; or (*iii*) use some *internal measures* of tasks that are conveyed through annotations to the source code [4], or through scheduler directives [5]. The first two levels of information are appropriate only for computations that exhibit predictable patterns of resource requests. This may not be true for the applications considered; hence, dynamic strategies that use these levels of information are not expected to perform well.

The approach taken in the paper attempts to use information in the third category above. Although it is intractable to correctly predict (a priori) the behavior of an application over its entire lifetime, it may be possible to capture the *near-future behavior* of tasks by expressions of task parameters. Run-time evaluation of these expressions constructed at compile-time yields estimates of task behavior that are valid for a short, future interval of time.

We propose a set of systematic heuristics that analyze the task description, and that can be used in conjunction with profiling information about the behavior of the tasks to estimate the near-future task requirements as expressions of task parameters. The estimation heuristics abstract away variations in resource requests that are caused by local variables of the task, but preserve differences captured in the task parameters. They build on the work of Sarkar [6, 7], who uses profiling-based compile-time estimates of average task behavior to partition and schedule applicative programs. Our approach of capturing the internal behavior of applications through these estimates of near-term behavior contrasts with the annotation approach adopted by Hornig [4], who uses user-supplied complexity measures of the entire task to decide on the partitioning of Lisp-like programs; and the approach of Ngai, et al.[5] who derive estimates of measures like the message-wait time on the basis of best-case analyses.

2 Estimation of Near-Future Task Behavior

The notion of a task that we use is an indivisible unit of computation that does not have any precedence constraints after it has started execution. Consequently, we do *not* estimate factors such as communication wait times, or time spent waiting for CPU control after the precedence relationships have been satisfied. The near-future behavior of an executing task is determined primarily by its task description and the current context of the task. In imper-

[†]Research Supported in part by National Science Foundation grant MIP 92-18715, by National Aeronautics and Space Administration under Grant NAG 1-613, and by Sumitomo Electric Industries, Yokohama, Japan.

Proceedings of Computer Software and Applications Conference, November 3-5, 1993.

ative programs, uncertainty in behavior arises due to (*i*) different paths being taken at *conditional statements* in the program segment describing the task; (*ii*) different number of iterations of the *loop blocks*; and (*iii*) different invocations of *functions* resulting in different behaviors. This section presents heuristics that analyze the program segment describing the task, and which can be used in conjunction with profile information to alleviate uncertainty in task behavior. Due to the inherent intractability of program analysis, the behavior estimation of all program structures is not possible through pure analysis alone. Profile information captures the average-case behavior of the program structure, and is used to provide the base estimate that can be further improved heuristically.

2.1 Programming Model

A divide-and-conquer algorithm can be represented as a set of interacting *objects* whose *methods* are defined using an imperative language. During execution, object creation and method invocation take place by message passing. A resident scheduler object is responsible for the management of objects resident on a processor. The scheduler objects periodically exchange their estimates of the local load. On receiving a message, an object can do one or more of the following operations: (*i*) compute a function, (*ii*) create a new object, or (*iii*) invoke the method of a different object and wait for a response.

An object does not occupy the CPU continuously during its lifetime. The periods of inactivity in the object's lifetime correspond to the time when it is waiting for one of its requests to return with a response, or is waiting for some other object to invoke one of its methods. In the mapping of the algorithmic description to our notion of a task, we model each period of activity of the object as a separate unit of indivisible computation. Each *trigger* message (a response or a method-invocation message) corresponds to the satisfaction of a precedence constraint in the general task graph.

The dependence of a task on multiple trigger messages (as might happen when a task has multiple predecessors, all of which must finish executing before the task can commence) is modeled by negligible periods of activity after the reception of all but the last trigger message. A corresponding situation exists for the case when a task needs to send messages to all its successors.

The choice of an object-oriented representation allows the analysis to be restricted to the level of code-blocks contained in a public method of the object. Since the computation performed by the task can be represented using a wide variety of program structures, the estimation heuristics should handle any general program.

2.2 Heuristics for Behavior Estimation

For the sake of clarity, we limit the following discussion to estimating near-future *execution-times.* The initial representation of the program segment is assumed to be the traditional *control flow graph* (CFG). We are interested in CFGs only at the level of schedulable units in a method. We assume that nodes of a CFG represent basic blocks, where a mapping exists between program statements and basic blocks. We further assume that the execution times of primitive operations are available. As is in other code analysis and optimization techniques, we assume that the CFG is *reducible. Node splitting* is a standard approach for transforming an irreducible CFG into a reducible one.

A reducible CFG has a unique depth-first spanning tree and hence a unique *interval structure* that can be computed easily [8]. The intervals identify the loops in the program and are defined as below.

Interval. An interval with *header h* is a set I of nodes in the graph such that (*i*) I is a single-entry region with entry node h, and (*ii*) all circuits in the interval contain h. In other words, $I - \{h\}$ is circuit-free.

A derived graph $\mathcal{I}(G)$ can be defined by considering each interval of the original graph G to be a single node. It is now possible to repeat the process on $\mathcal{I}(G)$ to obtain $\mathcal{I}^2(G)$ and so on. For a reducible graph G, there exists some k such that $\mathcal{I}^k(G)$ is the trivial flow graph consisting of a single node. Thus, interval structuring allows a hierarchical interpretation of the program for the purpose of behavior estimation.

We also use the notion of a control-dependence graph (CDG) derived from the *control-dependence* relationship defined by Ferrante et al. [9]. A node Y in the CFG is **control-dependent** on a predicate X if X has at least two exits. Following one of the exits from X always results in Y being executed, while taking another exit may result in Y not being executed.

Our proposed method assesses the execution time for each interval in terms of estimates for the intervals contained within it. A single node is a trivial interval. Its execution-time estimate corresponds to interval $\mathcal{I}^k(G)$. Profile information provides the base estimate for each interval. Applicable heuristics from the set described below are then used to improve the accuracy of the estimate. The execution time of each interval is expressed as the sum of the execution times of each contained subinterval, weighted by the **statement count** of the subinterval. We associate a **branch probability** with each conditional statement, and a **loop count** with each loop block. The latter denotes the number of iterations of the loop body. The statement count is computed from the branch probabilities and loop counts using the CDG, by adding the contribution of each loop to the statement count. The contribution of a loop is evaluated as below.

1. Identify nodes in the CFG that lie on some path that includes the exit-predicate of the loop and the back-edge to the interval header.
2. Construct a CDG using these nodes and the edges on the path.
3. Assign contributions to statement counts based on the set of control dependencies for each node:
    - control-independent nodes: loop count.
    - control-dependent nodes: (statement count of parent)×(branch probability).

A *pseudo-edge* from each interval exit to the interval header ensures that there exists at least one loop that every node lies on. Figure 1 shows a CFG and the contribution of loop $L2$ to the statement counts of the graph nodes.

The estimation method assumes that a similar technique has been used on the CFG of the function body to

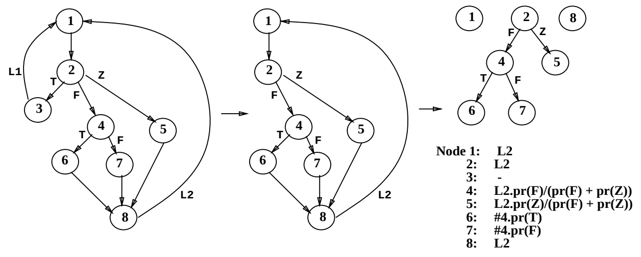

Figure 1: Example showing computation of Statement Counts.

obtain the execution-time estimate of any function calls in the interval. The information required from profiling is:

- for each conditional statement, the average probability of associated control conditions; and
- for each contained loop, the average loop count in one execution of the interval.

The following definitions are required before the estimation heuristics can be described.

Predicate function. This is a function in terms of the variables that appear in the predicate expression. It is obtained by rewriting the expression such that all variables appear on one side of the Boolean operator, and the other side has only constants.

Global variables. These refer to the task variables in whose terms the estimate is expressed. They are a subset of all the task variables that have well-defined values at the time the estimate expression is evaluated.

t-global variables. These are local variables that propagate the effect of the global variables in the CFG. They belong to one of the following types:

- an initial assignment to a global variable or a constant;
- a subsequent *use* in a loop predicate that contains only global variables and constants (in addition to this variable);
- all intervening *definitions* of the variable involving only constants, global variables, and this variable.

The estimation heuristics provide better information (than that obtained by profiling alone) about the branch probabilities, the loop counts, and the execution-time estimates of functions. Table 1 gives an overview of the heuristics along with the conditions under which they apply. A detailed description of each heuristic follows.

HEURISTIC 1 (Identifying active nodes in the CFG). Given the set of global variables, a predicate node in the CFG is said to be **evaluatable** if it contains only global variables and constants. The CDG identifies the **active** nodes given a particular control edge (the value that the predicate evaluates to) from the predicate. These are nodes that will execute if execution proceeds along the selected control edge. Thus, the active nodes corresponding to different control conditions of evaluatable predicates can be identified at the point when the task executes the expression that contains the estimate.

Table 1: Heuristics for Behavior Estimation in Programs (H1 is for conditional program structures and estimating branch probabilities; H2 and H3 are for loops based on loop counts; H4-H6 are for function-call program structures).

	Condition	*Description*
$H1$	Evaluatable predicates in CDG.	Range of control condition identifies active nodes.
$H2$	Constants, global and t-global variables in predicate; *associative* operations on t-global variables in loop.	Relate difference in predicate function for one iteration of the loop to required difference. Solve constraint equation.
$H3$	Loop-operations on t-global variables are *non-associative*.	Use profile information regarding *sequence* of execution to relate loop count with net difference in the predicate function.
$H4$		Compute estimate in terms of formal parameters of callee; transform to variables of caller.
$H5$	Recursive: depth affected only by global variables.	Set up recursive cost expression; solve using some heuristic.
$H6$	Local variables affect depth of recursion.	Use Sarkar's heuristic to obtain estimate for external invocation of function.

Statement counts of nodes corresponding to each control condition are computed by taking the branch probability of that condition to be 1, and 0 for all other values of the predicate. Since all predicate nodes in the graph are unlikely to be evaluatable, multiple estimates (in terms of different sets of global variables) may be required to capture the task behavior. In the absence of any evaluatable predicates, profile information regarding the branch probability of the predicate node is used to compute the statement counts.

HEURISTIC 2 (Computing loop counts for associative operations). If a loop predicate (predicates defining exit from the loop) contains t-global variables then the loop count is affected by the values of the global variables. An expression for the loop count can be written in terms of the global variables if all the operations in the loop body that *define* the value of t-global variables are **associative**. Associative operations are defined as below.

Associative Operations. A set of operations are defined as associative if the final value of the t-global variable on application of an arbitrary sequence of these operations is independent of the order in which the operations are applied. Examples of sets of associative operations include $\{+, -\}$, $\{\times, \div\}$ and $\{\text{pow}(-, x), \log_x(-)\}$.

Given a loop that satisfies the above conditions and has one exit predicate, the loop count is estimated by relating the difference in the predicate function for one execution of the loop to the net required difference. The required difference is evaluated by examining the initial value of the predicate function (defined on entry to the loop body) and the value of the function that would cause an exit from the loop. The loop count is estimated as follows.

$$\frac{\textit{net difference in predicate function}}{\textit{estimated difference due to one iteration of the loop}}$$

The effects of the interval nodes on the predicate function is hierarchically evaluated in a manner similar to the behavior estimation of the subinterval, and is weighted by the statement counts of the nodes that lie on the loop (a loop-count of unity is assumed for this purpose). For a loop that has multiple exits, the loop counts for each exit are estimated as above, and the overall loop count is estimated by the minimum of the individual loop counts.

HEURISTIC 3 (Estimating loop counts for non-associative operations). The above heuristic cannot be applied when the loop body contains non-associative operations on the t-global variables that appear in the loop predicate. The approach adopted in such circumstances is to estimate the change in value of the predicate function for a typical *sequence of operations* on the t-global variables. Profile information is used to derive the average number of times a particular operation occurs between the closest two occurrences of all other operations. Having obtained an estimate of the execution sequence, the predicate-function difference for the sequence is obtained by computing the difference that would result from an application of this sequence of operations. The loop count is estimated by relating the change in the predicate function for one execution of the sequence to the net difference in the predicate function.

HEURISTIC 4 (Estimating behavior of function calls). If function-call behavior is estimated only on the basis of profile information, all invocations of the function would have identical behavior. A more informed estimate is possible when the arguments of the function are simply expressed in terms of the global variables of the caller function. First, obtain an estimate of the function behavior in terms of the formal parameters of the callee function, and then rewrite this expression so it is in terms of the global variables of the caller function. The behavior estimation of the callee function in terms of its formal parameters is done in a fashion similar to the behavior estimation of a program segment.

HEURISTIC 5 (Estimating behavior of recursive functions via analysis). Heuristic 4 used in isolation does not suffice for estimating the behavior of recursive functions. If the depth of recursion is influenced by the global variables, we can analyze the behavior of the recursive function as below.

1. Set up a recursive cost expression in terms of the global variables.
2. Map to a set of difference equations.
3. Solve the set of difference equations along with base conditions. Exact solutions may not always be possible; in which case heuristics can be used to obtain the best-case, average-case, or worst-case behaviors of the recursion.

HEURISTIC 6 (Estimating behavior of recursive functions via profiling). This uses the approach adopted by Sarkar [6], which distinguishes between the cost of **external** and **internal** calls (from mutually recursive functions) to a recursive function. In our model, since there can be no recursion at the task level, we are interested in predicting the cost of only the external calls to any member of a mutually recursive function set $\{u_1, u_2, \ldots, u_k\}$. We define the following quantities:

$\mathcal{B}(j)$:	base cost of u_j
$\mathcal{E}(j)$:	cost of external call to u_j
$\mathcal{I}(j)$:	cost of internal call
t_j:	total number of calls to u_j (profiled)
$c_{i,j}$:	contribution of function-calls to u_j in u_i (profiled)
i_j:	no. of internal calls made to u_j; $i_j = \sum_{1\le i\le k} t_i c_{i,j}$
e_j:	no. of external calls made to u_j; $e_j = t_j - i_j$

The total computation performed by all the members of the recursive function set is given by $\sum_{1\le j\le k} t_j \mathcal{B}_j$. Since all this work is requested by external calls,

$$\sum_{1\le j\le k} e_j \mathcal{E}_j = \sum_{1\le i\le k} t_i \mathcal{B}_i$$

Under the assumption that all internal calls (to any function in $\{u_1, u_2 \ldots, u_k\}$) have the same execution-time estimate, the following estimate can be derived by simple algebraic manipulation:

$$\mathcal{E}_j = \mathcal{B}_j + \mathcal{I}' \sum_{1\le h\le k} c_{j,h} \quad \text{where } \mathcal{I}' = \frac{\sum_{1\le j\le k} i_j \mathcal{B}_j}{\sum_{1\le j\le k} e_j . \sum_{1\le h\le k} c_{j,h}}$$

The above heuristics do not estimate task behavior of all program segments with the same accuracy. The tolerance of these estimates to inaccuracies will be explored in Section 4.

3 Dynamic Scheduling using Local Information

This section describes a dynamic scheduling strategy for divide-and-conquer algorithms that uses estimates of near-future behaviors of active tasks in the system. We assume that the execution-time requirements are the only task behavior of interest. In our programming model, tasks correspond to unbroken periods of object activity. Tasks with multiple predecessors decide their own schedulability after they have received all their trigger messages, while parent tasks determine the readiness of child tasks that await only one trigger message. The scheduler uses the task-behavior estimates in two ways.

- Task placement decisions can be made based on initial estimates that are conveyed to the scheduler either by the parent task or by the task itself.
- Estimates from already executing tasks are used to improve the accuracy of the estimated load on a site.

The scheduler is responsible for the relocation of an object if the selected site is not the currently occupied site. The scheduler uses the cost of migration in deciding whether or not to move the object. We assume that migration cost is low (of the order of a message transfer) since objects get relocated only at inter-method boundaries. This avoids the copying of activation records since the object is trivially checkpointed. The cost of rerouting each message to the object at its new site is reduced by having a shared object-name space that is built on top of physically distributed memory.

3.1 Execution environment

The processor system consists of a set of *processors* $\mathcal{P} = \{P_i : 1 \leq i \leq m\}$. Processor P_i has a *speed* b_i. The interpretation of *speed* is that a task T with execution-time requirement $\mu(T)$ on a processor of speed 1 requires $\frac{\mu(T)}{b_i}$ time units on processor P_i. Assume that $b_1 \geq b_2 \geq \cdots \geq b_m > 0$.

Each processor maintains a queue of ready tasks and associates an estimate $\mathcal{E}(T)$ for every active task T. In addition, each processor maintains a 2-tuple $(\mathcal{R}, t_s)$, where $\mathcal{R}$ is the *remaining busy time* and is the sum of the estimates $\mathcal{E}(T)$ of all tasks T that are waiting for CPU control. If H indicates the task that is at the head of the processor queue and is currently occupying the CPU, then $(\mathcal{R}(P) + \mathcal{E}(H))$ represents the current estimate of how long the processor is going to remain busy. The *start time* $t_s(P)$ maintains the last time instant at which the estimate of the header task $\mathcal{E}(H)$ was updated. A function *update_estimate(T, P, ρ)* operates on $(\mathcal{R}, t_s)$ and is defined as below:

```
if ($T == H$) then
    $(\mathcal{E}(T), \mathcal{R}) = (\frac{\rho}{b}, \mathcal{R} + \frac{\rho}{b})$
else if ($\rho \neq 0$) then
        $(\mathcal{E}(H), t_s) = (\frac{\rho}{b}, current\_time)$
    else
        $(H, \mathcal{R}, t_s) = (first(T), \mathcal{R} - first(T), current\_time)$
```

ρ is an estimate of the amount of time that task T would take if it were executing on a processor of speed 1. Storing only the ratio with respect to the processor speed allows comparisons to be made across heterogeneous processors using the absolute values. The third assignment above corresponds to the completion of the task at the head of the queue, in which case the first element of the queue of waiting tasks is permitted to next occupy the CPU.

The **compensated load** measure, $\mathcal{L}(P, t)$, of processor P is defined as

$$\mathcal{L}(P, t) = \max(0, \mathcal{E}(H) - (t - t_s(P))) + \mathcal{R}(P)$$

$\mathcal{L}(P, t)$ captures the change in load as a result of the additional processing of the task, which is at the head of the queue, since the last estimate. The expression also accounts for underestimates.

The true load, $\mathcal{W}(P, t)$, on processor P is a function of many parameters like the CPU requirements of tasks, the traffic to secondary memory, system daemons, background load, etc. Thus, there is an inaccuracy in considering the system load as being due to the CPU requirement alone. The actual processor load can be modeled by the following expression:

$$\mathcal{W}(P) = \mathcal{L}(P) + \mathcal{X}_{load}$$

where $\mathcal{X}_{load}$ is an arbitrary random variable that models the effect of the other parameters.

For a divide-and-conquer algorithm executed on the parallel system defined, its partial order $<$ is in the form of an *outin tree* [1] (composed of an outtree and an intree with shared leaves). The *Level Algorithm* (**LA**) [10] achieves an optimal solution in the case of identical processors when preemption is allowed. The problem Q/pmtn, tree/C_{max} remains an open problem, with **LA** having the best-known performance-ratio bound (when compared with the theoretical optimal) of $\sqrt{m} + \frac{1}{2}$, where m is the number of processors.

For task T, $\rho(T)$ is the execution-time estimate obtained by applying the heuristics discussed in the previous section, which is available at the time of creation of T. The error in the estimate is problem dependent and is a function of the complexity of the program structures, number of branches in the CFG that depend on local variable values, etc.

In short, the following assumptions are made in our simulations.

1. The CPU is considered the only schedulable resource.
2. All processors are capable of executing any task. In particular, we ignore the cost of communication with such entities as files.
3. Tasks obtain control of the processor according to a *first-come first-served non-preemptive* policy.
4. All processors in the system are connected. The hardware cost of communication between any pair of processors is assumed to be negligible when compared to the average task granularity. The software latency is explicitly accounted for when estimating the task behavior in terms of primitive operations.
5. Load updates are assumed to propagate to other processors with negligible delay.
6. The overhead of the resource-allocation decisions that include load-queries is amortized among the resource requirements of the participating tasks.

3.2 The D_LPT Heuristic

The scheduling heuristic, **D_LPT**, is adapted from deterministic scheduling strategies of independent tasks, and makes scheduling decisions for tasks with **L***argest*-**P***rocessing*-**T***ime* estimates first. Tasks get allocated to processors where they are likely to complete the earliest. The scheduling heuristic interprets the behavior estimates obtained from tasks as an estimate of the execution time of the task.

The heuristic is described in terms of *precondition:action* pairs. It is assumed that $\mathcal{L}(P, t)$ is the best available estimate of the load, $\mathcal{W}(P, t)$, on a processor.

When a set of tasks is ready to be scheduled at time t and $\{T_i\}$ is non-empty:

1. Pick $T = \max_{T_k \in \{T_i\}}(\rho(T_k))$, $P_{new} = \min_{1 \leq j \leq m} (\mathcal{L}(P_j, t) + \frac{\rho(T)}{b_j})$. This chooses the processor on which the task with the largest processing-time estimate is likely to complete the earliest.
2. If T does not currently reside on any processor, assign task T to processor P_{new}, and *update_estimate($T, P_{new}, \rho(T)$)*.

3. If T currently resides on processor P_{cur}, migrate T to processor P_{new} if the following condition holds:

$$(\mathcal{L}(P_{cur},t)+\frac{\rho(T)}{b_{cur}})-(\mathcal{L}(P_{new},t)+\frac{\rho(T)}{b_{new}})\geq f(\theta),$$

where $f(\theta)$ is an arbitrary, non-decreasing function of migration overhead θ and signifies the migration threshold of the strategy. Update the load information on whichever processor the task gets assigned to.

When a task T provides an estimate $\rho(T)$ at time t and P is the processor on which T is currently executing:

1. *Update_estimate($T, P, \rho(T)$).*
2. If $\rho(T)$ is 0 (T has completed execution), obtain the next ready task from the task queue using the local scheduling policy. (*update_estimate()* takes care of the changes that need to be made to the load estimates).

When a node becomes idle at time t:

1. Choose the most loaded processor, $P_{max} = \max_{1\leq j\leq m}\ (\mathcal{L}(P_j,t))$.
2. Move the ready task T that has the least priority if the following condition is satisfied

$$\mathcal{L}(P_{max},t)-\frac{\rho(T)}{b_{max}}\geq f(\theta).$$

This ensures that a task does not migrate if it would start executing later at the new location that it would at the original one.

Note that **D_LPT** is a list schedule (with no unforced idleness) in the case of identical processors. This allows us to bound the worst-case performance of **D_LPT** in the case of identical processors to be within two times the theoretical best-length schedule. Our scheduling heuristic differs in terms of overhead from traditional strategies in that it keeps track of the estimates from different tasks instead of aggregating them into a single load measure.

4 Experiments and Results

4.1 Experimentation System

The execution system was simulated in CSIM with *event-trees* driving the simulations. The event-trees were generated from sequential traces of instrumented application programs. The application used to drive the simulation was QuickSort. Several instances of the application program for a set of problem sizes were randomly constructed. The number of objects in a particular event-tree varied from 100 to 2000. The number of tasks (in the divide-and-conquer task graph) varied from 200 to 5000.

The execution-time estimates were derived using the method in Section 2 and a set of primitive-operation timings. The estimation expressions were stored in terms of the data parameters recorded in the event tree and were used to compute execution-time estimates of active tasks in our simulations. An input to the simulator, *annotation overhead*, allows variation in the time taken to evaluate an estimation expression.

In our simulations, we assume that $\mathcal{L}(P,t)$ is the best available estimate of the actual processor load, $\mathcal{W}(P,t)$, which in turn is assumed to be entirely due to execution-time requirements of tasks. Errors in the load estimate are, thus, entirely due to errors in the behavior estimates. Instead of using a random (and arbitrary) error model to capture the inaccuracies, we have chosen to rely on selective application of some heuristics to produce varying levels of accuracy in the estimation expression.

Inputs to the simulator include the number of processors, the speed factors of the processors, the threshold functions for different scheduling strategies, and the annotation overhead. The migration overhead is taken to be a constant, θ, independent of the object size. This constant is supposed to signify an upper bound of the overhead that an object-migration in the system would involve.

To compare with **D_LPT**, we also implemented the following scheduling heuristics which schedule ready tasks to:

- a randomly chosen processor;
- a round-robin selection of processors;
- a processor with the largest number of objects weighted by its speed factor [2];
- a processor with the largest number of unexecuted messages weighted by its speed factor [3].

Due to space limitation, we do not show the performance for the case of identical processors and the performance comparison with respect to Horvath et al.'s algorithm [10].

The performance of a strategy was observed with respect to its completion time. An experiment was stopped when either its sample mean value was within 10% of its actual mean value with a confidence of more than 90%, or when the number of samples tested exceeded 50.

4.2 Application-program estimates

Figure 2 contains the code for QuickSort with profiling counters shown at the beginning of the block whose execution frequency they measure. To simplify the description, we assume that only pointers to the subarray are passed. The entry point in the above program segment is the *Array::sort()* method. The creation of the child objects in lines 6 and 7 are blocking calls when the parent object needs to relinquish CPU control. We can, thus, define two tasks in this program segment: **SPLIT** consists of the execution of either the *localSort* or *partition* private functions; and **COMBINE** starts off at line 8 and models the return of the sorted subarray from the child objects.

In order to estimate the execution-time requirements of these two tasks, we need to identify structures that bring about the uncertainty in their behavior. The branches in lines 2 and 11, as well as the loops in lines 19-25, 20, 21 capture all the points that would affect estimation. Profiling the program segment with displayed counters yields the values of branch probabilities and loop counts as shown in Table 2. Profiling was based on observed object behavior during 100 runs of QuickSort on random sequences varying from 1000 to 3000 elements in length.

In order to simplify the process of writing down the estimation expressions, we classify the primitive operations into five categories: a) T_1: assign, cmp, push, inc; b) T_2: jmp, call; c) T_3: ret; d) T_4: send, recv; and e) T_5: block. *send, recv* refer to the constant hardware latencies

```
void Array::sort(int m, int n) {       sr_f
  /* public interface */
  if ((n-m) < GS)
    localSort(m,n);
  else {
    int p = partition(m,n);            sr_p
    Array *leftArr = new Array;
    Array *rightArr = new Array;
    future(leftArr->sort(m,p));
    rightArr->sort(p+1,n); }  }
void Array::localSort(int m, int n) { ls_f
  /* local work */
  if (m < n){
    int p = partition(m, n);
    localSort(m, p);
    localSort(p+1, n); } }
int Array::partition(int m, int n) {  pt_f
  /* in-place partitioning */
  int x = data[m];
  int i = m - 1;
  int j = n + 1;
  while (1) {                          pt_3
    while (data[++i] < x);             pt_1
    while (data[--j] > x);             pt_2
    if (i < j)
      swap(i,j);
    else
      return(j); } }
```

Figure 2: Code for QuickSort algorithm.

of message communication. The software latency is accounted for in terms of other primitive operations. Note that *send*, *recv*, and *block* operations are required for the object-management operations like sending and receiving messages, and context-switching to another object. Thus, by varying the relative costs of these five kind of operations, we can create different **grain sizes** of the computation, where grain size refers to the ratio of useful work to the overhead of object management.

Table 3 presents 6 different estimates for the **SPLIT** task for one choice of the relative costs of primitive operations, corresponding to medium-grain operations. (Results for small-grain and large-grain operations are not shown due to space limitation.) The estimate level, `accurate`, assumes the execution-time estimate to be completely accurate, while the estimate level, `average`, refers to the estimate being solely profile-based. We obtain the accurate execution time of the program segment by using the counter values that record the operations taking place during actual task execution. The four intermediate estimates correspond to selective application of the estimation heuristics described in Section 2. The names of the intermediate estimates indicate the function (*partition* or *localSort*) evaluated using either the estimation heuristics (`evl-`) or estimations based on profile information (`est-`). Thus, `evl-part,est-sort` corresponds to an estimation level where the *partition* function is evaluated using the heuristics, and the *localSort* function is estimated using Sarkar's heuristic. All the estimates are expressions in terms of the parameter $x = (n - m + 1)$.

The *COMBINE* task is quite straightforward (2 *recv* + 1 *send* + 1 *block*), and the estimate is always assumed to be accurate.

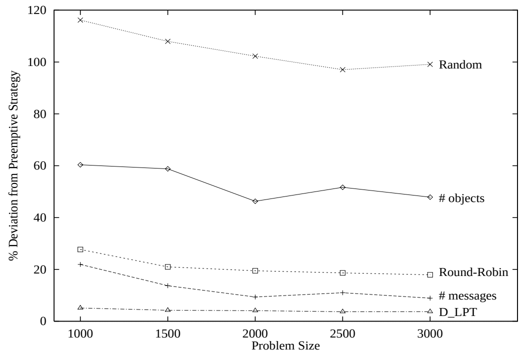

Figure 3: Variations in Problem Size (4 procs (4:1:1:1); mig_ovhd = 100; ann_ovhd = 50; MEDIUM grain; (evl-part,evl-sort)).

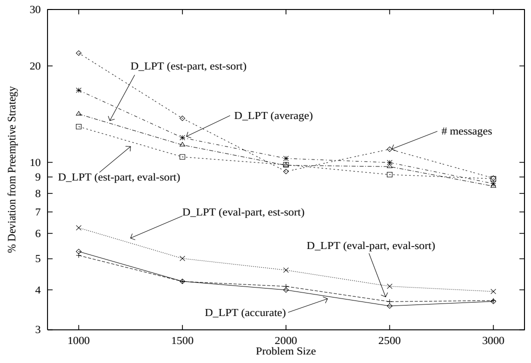

Figure 4: Variations in Estimation Accuracy (4 procs (4:1:1:1); mig_ovhd = 100; ann_ovhd = 50; MEDIUM grain).

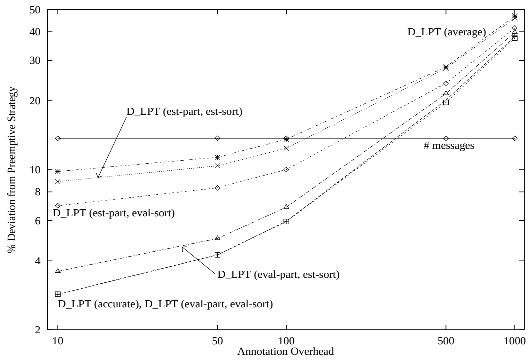

Figure 5: Variations in Annotation Overhead (4 procs (4:1:1:1); 1500 elmts; mig_ovhd = 100; MEDIUM grain).

Table 2: Profiling information from QuickSort algorithm.

Parameter	*Profile*
$P_{sr} = \frac{sr\_p}{sr\_f}$ = probability of branch (Lines 4–9) being taken	0.50
$L_1 = \frac{pt\_1}{pt\_3}$ = average number of times inner loop (Line 20) executes for one execution of outer loop (Lines 19–25)	2.15
$L_2 = \frac{pt\_2}{pt\_3}$ = average number of times inner loop (Line 21) executes for one execution of outer loop (Lines 19–25)	2.65
$L_3 = \frac{pt\_3}{pt\_f}$ = average number of times loop (Lines 19–25) executes for one execution of Array::partition	3.60
$P_{ls} = \frac{ls\_p}{ls\_f}$ = probability of branch (Lines 11–14) being taken	0.46

Table 3: QuickSort (**SPLIT**) behavior using Estimation Heuristics for medium-grain primitive operations.

Estimate	MEDIUM GRAIN $T_1, T_2, T_3 = T; T_4, T_5 = 5T$		
accurate	$(x \leq GS)$:		$(22 + 6ls\_f + 13ls\_p + 4pt\_f + 6pt\_3 + 4pt\_1 + 4pt\_2)T$
	else		$(32 + 4pt\_f + 6pt\_3 + 4pt\_1 + 4pt\_2)T$
evl-part, evl-sort ($H1, H2, H4, H5$)	$(x \leq GS)$:		$(28.25x - 0.25 + 17.44xlogx)T$
	else		$(41.25 + 5.25x)T$
est-part, evl-sort ($H1, H4, H5$)	$(x \leq GS)$:		$(113.61x - 85.61)T$
	else		$126.61T$
evl-part, est-sort ($H1, H2, H4, H6$)	$(x \leq GS)$:	$(x > 0)$:	$764.28T$
		else	$28T$
	else		$(41.25 + 5.25x)T$
est-part, est-sort ($H1, H6$)	$(x \leq GS)$:	$(x > 0)$:	$764.28T$
		else	$28T$
	else		$126.61T$
average	$445.45T$		

4.3 Performance Results

The performance of all strategies is measured in terms of percentage deviation from the *Level algorithm*. The behavior of the strategies was recorded for the following.

Variations in Problem Size. Figure 3 shows the performance of the different strategies on 5 problem sizes of QuickSort under realistic conditions using the best estimate generated using the heuristics of Section 2. **D_LPT** performs consistently better than any of the other heuristics and exhibits a problem-size independent behavior. It also adapts itself to a non-uniform configuration better than a simple strategy such as the number of messages.

Variations in Estimation Accuracy. Figure 4 shows the effect of using different estimates in **D_LPT**. As is expected, using accurate information results in good performance, as does using estimates derived using all the estimation heuristics. The significant difference in the behavior of the `evl-part, est-sort` and `est-part,evl-sort` emphasizes the selection of which heuristic to apply. The *partition* function exhibits wider variance (in terms of the data values it handles), and consequently its performance is more dependent on better estimation of its behavior. **D_LPT** performs the worst when using number of messages as a load indicator. In this case, the strategy treats each task as having the same estimated execution time (which is not true in this application).

Variations in Annotation Overhead. **D_LPT** with different estimates and the strategy, which uses the number of messages as a load indicator, are plotted in Figure 5 for variations in the annotation overhead from 10 to 1000. An annotation of 10 units corresponds to the complexity of a single assignment statement. This plot shows the permissible annotation overhead that can be tolerated by a strategy that uses a particular level of behavior estimation before its performance reaches that of the strategy using the number of messages.

Though our results are reported only based on recursive patterns of task generation, we expect the observed trends to be similar even if the algorithm is written differently, say in an iterative form. This is true because work can be captured accurately by the behavior estimates.

References

[1] B. W. Wah and G.-J. Li, “Optimal parallel evaluation of AND trees,” *Journal of Parallel and Distributed Computing*, vol. 10, pp. 1–17, Oct. 1990.

[2] P. Hudak, “Distributed applicative processing systems: Project goals, motivation, and status report,” Tech. Rep. YALEU/DCS/TR-317, Department of Computer Science, Yale Univ., May 1984.

[3] W. Shu and L.V.Kale, “A dynamic scheduling strategy for the Chare-kernel system,” *Proceedings of SUPERCOMPUTING*, pp. 389–398, IEEE and ACM SIGARCH, 1989.

[4] D. A. Hornig, *Automatic Partitioning and Scheduling on a Network of Personal Computers.* PhD thesis, Carnegie-Mellon University, Nov. 1984.

[5] T.-F. Ngai, S. Lundstrom, and M. Flynn, “Automated run-time scheduling of unstructured scientific computation on scalable multiprocessors,” in *Unstructured Scientific Computation on Scalable Multiprocessors* (P. Mehrotra, J. Saltz, and R. Voigt, eds.), Cambridge, MA: The MIT Press, 1991.

[6] V. Sarkar and J. Hennessy, “Partitioning parallel programs for macro-dataflow,” *Proceedings Conf. on Lisp and Functional Programming*, pp. 202–211, ACM, Aug. 1986.

[7] V. Sarkar, “Determining average program execution times and their variance,” *ACM SIGPLAN Notices*, vol. 24, pp. 298–312, July 1989.

[8] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Communications of the ACM*, vol. 19, pp. 137–146, Mar. 1976.

[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, 1987.

[10] E. C. Horvath, S. Lam, and R. Sethi, “A level algorithm for preemptive scheduling,” *Journal of the ACM*, vol. 24, pp. 32–43, Jan. 1977.