JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 13, 395–411 (1991)

The Effectiveness of Task-Level Parallelism for Production Systems

WILSON HARVEY, DIRK KALP, MILIND TAMBE, DAVID MCKEOWN, AND ALLEN NEWELL

School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213-3890

Large production systems (rule-based systems) continue to suffer from extremely slow execution which limits their utility in practical applications as well as in research settings. Most investigations in speeding up these systems have focused on match parallelism. These investigations have revealed that the total speed-up available from this source is insufficient to alleviate the problem of slow execution in large-scale production system implementations. In this paper, we focus on task-level parallelism, which is obtained by a high-level decomposition of the production system. Speed-ups obtained from task-level parallelism will multiply with the speed-ups obtained from match parallelism. The vehicle for our investigation of task-level parallelism is SPAM, a high-level vision system, implemented as a production system. SPAM is a mature research system with a typical run requiring between 50,000 and 400,000 production firings. We report very encouraging speed-ups from task-level parallelism in SPAM—our parallel implementation shows near linear speed-ups of over 12-fold using 14 processors and points the way to substantial (50- to 100-fold) speed-ups. We present a characterization of task-level parallelism in production systems and describe our methodology for selecting and applying a particular approach to parallelize SPAM. Additionally, we report the speed-ups obtained from the use of *virtual shared memory*. Overall, task-level parallelism has not received much attention in the literature. Our experience illustrates that it is potentially a very important tool for speeding up large-scale production systems.

1. INTRODUCTION

Large production systems (rule-based systems) continue to suffer from extremely slow execution which limits their utility in practical applications as well as research settings. Most efforts at speeding up these systems have focused on match (or knowledge-search) parallelism in production systems [1, 5, 8, 10, 20, 27, 28]. Though good speed-ups have been achieved in this process, the total speed-up available from this source is limited and is insufficient to alleviate the problem of slow execution in large-scale production systems. Such large-scale systems are expected only to increase in the future [2, 23], which will exacerbate the problem of long run times.

In this paper, we focus on task-level parallelism, which is obtained by a high-level decomposition of the production system. Speed-ups obtained from task-level parallelism will multiply with the speed-ups obtained from match parallelism. Our vehicle for the investigation of task-level parallelism is SPAM [17–19], a high-level vision system, implemented in a production system architecture. SPAM is a mature research system having over 600 productions, with a typical scene analysis task requiring between 50,000 and 400,000 production firings and an execution time of the order of 10 to 100 cpu hours.[1] It is a multiphase system, with each phase displaying a different computational characteristic. Unlike most other production systems examined for studies in parallelism, it has embedded in it a large computational demand related to the vision task that it performs. This task-related computation is separate from the computation performed for match in the system. This is evident in the large non-match-related processing time for this system. While many production systems spend up to 90% of their time in match, SPAM spends only about 30–50% of its time there.

In this paper, we show that the opportunities for task-level parallelism in SPAM are high and provide a much larger payoff in speed-up than match parallelism. We obtain near linear speed-ups for two different phases—LCC and RTF—of SPAM. For the LCC phase, we achieve a speed-up of over 12-fold using 14 processors on a 16-processor shared-memory multiprocessor. Our results indicate that a potential speed-up of 50- to 100-fold may be achievable due to task-level parallelism. We further show that match parallelism, when used in conjunction with task-level parallelism, gives another multiplicative factor of speed-up which is proportional to the size of the match component in the overall execution time. In the SPAM system, this additional multiplicative factor is around 1.5 to 2. Note that, for these parallelization experiments, we reimplemented SPAM using an optimized, C-based OPS5 implementation. All speed-ups reported here are computed against this implementation of the SPAM system run in uniprocessor mode. This baseline system provided a 10- to 20-fold speed-up to begin with over the original Lisp-based implementation used for SPAM. The speed-ups due to parallelism are, thus, not artificially enhanced by any hidden effect due to better implementation technology in the new parallel system.

We also present a methodology to arrive at a suitable parallel decomposition of SPAM and describe a set of experi-

[1] These measurements are taken from the Lisp-based version of OPS5 running on a VAX/785 processor.

0743-7315/91 $3.00

ments and measurements on SPAM that allowed us to select an appropriate grain of decomposition. These techniques are applied to the two different phases of SPAM mentioned above. These phases differ along at least two dimensions: the type of task performed and the amount of time spent in match. LCC is a constraint satisfaction task spending the majority of its time in vision-related computation outside of match. RTF is a typical AI classification task adhering more closely to the conventional production system model in terms of its match time. We expect that our techniques should be applicable to the analysis of other large production systems for evaluating the opportunities for task-level parallelism, regardless of the type of task performed or the amount of time spent in match.

This paper is organized as follows: Section 2 provides some background about production systems and SPAM, the image interpretation system that is the focus of our analysis of task-level parallelism. Section 3 discusses match parallelism and task-level parallelism in production systems. Section 4 exhibits our design methodology used to determine the task-level parallelism in SPAM. A new system, SPAM/PSM, resulted from the application of this methodology and its implementation is described in Section 5. Section 6 presents detailed results of experiments with match and task-level parallelism (with varying grain sizes). Section 7 presents the results of our experiments with virtual shared memory (or network shared memory). Section 8 presents a summary of our research results. Finally, Section 9 discusses some issues for future work.

2. BACKGROUND

In this section we provide a brief overview of OPS5 and SPAM. SPAM is implemented in OPS5, hence the description of OPS5 will be useful in understanding some of the issues in how SPAM represents knowledge about spatial and structural constraints used in computer vision. Besides providing background information, this section introduces the terminology that will be used in the rest of this paper.

2.1. *OPS5*

An OPS5 [4] production system is composed of a set of *if–then* rules, called *productions,* that make up the *production memory,* and a database of temporary data structures, called the *working memory.* The individual data structures are called working memory elements (WMEs), and are lists of attribute-value pairs. Each production consists of a conjunction of condition elements (CEs) corresponding to the *if* part of the rule (also called the left-hand side or LHS), and a set of actions corresponding to the *then* part of the rule (also called the right-hand side or RHS).

The CEs in a production consist of attribute-value tests, where some attributes may contain variables as values. The attribute-value tests of a CE must all be matched by a WME for the CE to match; the variables in the condition element may match any value, but if the variable occurs in more than one CE of a production, then all occurrences of the variable must match identical values. When all the CEs of a production are matched, the production is satisfied, and an instantiation of the production (a list of WMEs that matched it), is created and entered into the *conflict set.* The production system uses a selection procedure called *conflict-resolution* to choose a production from the conflict set, which is then *fired.* When a production fires, the RHS actions associated with that production are executed. The RHS actions can add, remove, or modify WMEs, or perform I/O.

The production system is executed by an interpreter that repeatedly cycles through three steps: (1) Match, (2) Conflict-resolution, (3) Act. The match procedure determines the set of satisfied productions, the conflict-resolution procedure selects a single instantiation, and the act procedure executes its RHS. These three steps are collectively called the *recognize–act cycle.*

2.2. *SPAM: A Production System Architecture for Scene Interpretation*

SPAM [17–19] is a production system architecture for the interpretation of aerial imagery with applications to automated cartography and digital mapping. It tests the hypothesis that the interpretation of aerial imagery requires substantial knowledge about the scene under consideration. Knowledge about the type of scene—airport, suburban housing development, urban city—aids in low-level and intermediate-level image analysis, and will drive high-level interpretation by constraining search for plausible consistent scene models. SPAM has been applied in two task areas: airport and suburban housing scene analyses. The remainder of this section describes the SPAM architecture, and gives run-time statistics that lead us to focus on two of its phases for parallelization.

The SPAM system architecture uses the production system framework to provide control in sequencing vision-related tasks as it performs feature recognition and model building within an image analysis. SPAM is distinctive from many typical production system models in which knowledge search is the only significant component. As mentioned in Section 1, SPAM spends more than 50% of its time in vision-related tasks *outside* of the OPS5 framework. Such vision-related tasks as image segmentation and the calculation of geometric constraints can be represented and performed more efficiently and effectively outside of OPS5. In addition, a large body of vision system software already exists to handle these types of processing.

The heterogeneous organization characterized here by SPAM may be appropriate to other domains where the mapping of complex analysis tasks is not suited to the representation provided by the production system framework. In this

type of system organization, the RHS actions become essentially external calls for information, specialized processing, or database access that are necessarily best represented outside of the production system.

As with many vision systems, SPAM attempts to interpret the 2-dimensional image of a 3-dimensional scene. A typical input image is shown in Fig. 1. The image is segmented into a collection of delineated objects in the scene. The goal of SPAM is to produce an interpretation for the segmented objects as a collection of real-world objects in the particular domain. Knowledge about the domain drives the interpretation process. For example, a final output for the image in Fig. 1 would be a model of the airport scene, describing where the runway, taxiways, terminal-building(s), etc., are all located. The overlay on the image in Fig. 1 graphically depicts this automatically generated airport model.

SPAM uses four basic types of scene interpretation primitives—*regions, fragments, functional areas,* and *models*—each of which is associated with a particular step in the interpretation process. SPAM performs scene interpretation by transforming image *regions* into scene *fragment* interpretations. It then aggregates these fragments into consistent and compatible collections called *functional areas.* Finally, it selects sets of functional areas to form *models* of the scene.

As shown in Fig. 2, each interpretation phase is executed in the order given. SPAM drives from a local, low-level set of interpretations to a more global, high-level, scene interpretation. There is a set of hard-wired productions for each phase that controls the order of rule executions, the processing of geometric computations, and other domain-independent tasks. However, this "bottom-up" organization does not preclude interactions between phases. For example, prediction of a fragment interpretation in the *functional-area (FA)* phase will automatically cause SPAM to reenter the *local-consistency check (LCC)* phase for that fragment. Other forms of top-down activity include stereo verification to disambiguate conflicting hypotheses in the *model-generation (MODEL)* phase and to perform linear alignment in the *region-to-fragment (RTF)* phase.

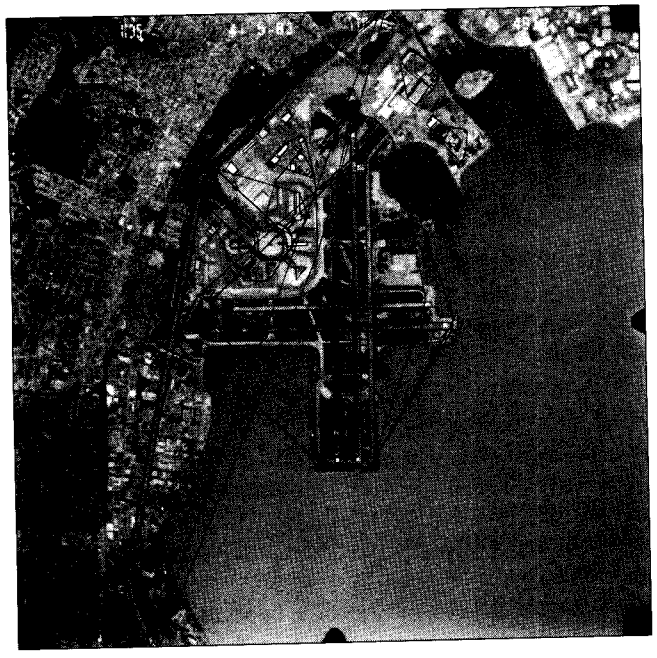

FIG. 1. Aerial image of San Francisco Airport with final SPAM results superimposed.

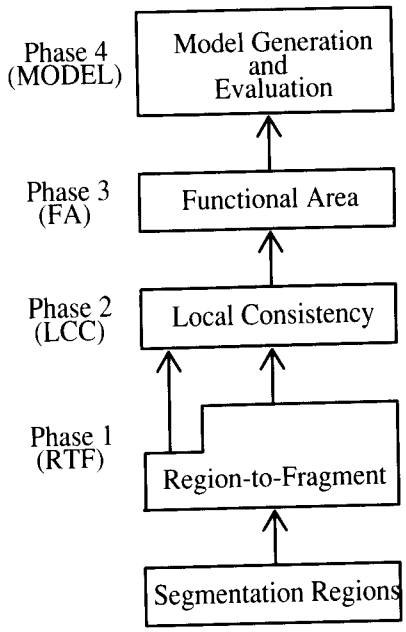

FIG. 2. Interpretation phases in SPAM.

Another way to view the flow of processing in SPAM is that knowledge is used to check for consistency among hypotheses; contexts are created based on collections of consistent hypotheses, and are then used to predict missing components. A collection of hypotheses must combine to create a context from which a prediction can be made. These contexts are refinements or spatial aggregations in the scene. For example, a collection of mutually consistent runways and taxiways might combine to generate a runway functional area. The context of a runway functional area then predicts that certain subareas within that functional area are good candidates in which grassy areas or tarmac regions may be found. However, an isolated runway or taxiway hypothesis cannot directly make these predictions. In SPAM the context determines the prediction. This serves to decrease the combinatorics of hypothesis generation and to allow the system to focus on those areas with strong support at each level of the interpretation.

Tables I, II, and III give statistics for the run-time and the number of production firings for each interpretation phase in SPAM for each of the three airports used in this study:

TABLE I

San Francisco Airport (Log No. 63)

SPAM phase	RTF	LCC	FA	MODEL	Total
Total CPU time (hours)	1.5	144.5	7.3	0.71	154.01
Total No. firings	11,274	185,950	10,447	3085	210,756
Effective productions/second	2.08	0.357	0.397	1.20	0.380
Total hypotheses	466	N/A	44	1	N/A

TABLE III

NASA Ames Moffett Field (Log No. 415)

SPAM phase	RTF	LCC	FA	MODEL	Total
Total CPU time (hours)	0.25	4.12	2.33	0.33	7.03
Total No. firings	4713	36,949	1503	3774	46,939
Effective productions/second	5.24	2.30	0.160	3.02	1.85
Total hypotheses	199	N/A	27	1	N/A

San Francisco International (SF), *Washington National (DC)*, and *NASA Ames Moffett Field (MOFF)*. It is interesting to note that the LCC and FA phases account for most of the overall time in a complete run. Further, within these two phases much of the RHS evaluation is performed outside OPS5 using external processes. For example, FA spends much of its time doing RHS evaluation outside of OPS5. RTF, on the other hand, spends most of its time within the traditional OPS5 evaluation model and consumes less time than FA, even though it executes a comparable number of productions.

As a result of this analysis, we decided to focus on parallelizing the LCC and RTF phases. The choice of LCC was motivated by the observation from the tables presented earlier, that LCC is by far the most expensive phase in terms of amount of time spent, number of productions, as well as number of production firings. Another rationale for this approach is the observation that this phase has the largest potential for growth. We believe that as new knowledge is added to the existing SPAM system, the proportion of time can only increase in the LCC phase. The RTF phase was selected for parallelization since it fits the framework of a traditional OPS5 system more closely than the other phases of SPAM—it thus contrasts with the computation in LCC, providing generality to the results presented.

It is interesting to consider the nature of the computations performed in the LCC and RTF phases and understand the differences among them. The RTF phase performs a traditional heuristic classification (or analysis) task [4]. It classifies/subclassifies regions in a scene as fragments based upon intrinsic properties of the region, such as shape, reflectance, and texture. For example, it may classify linear regions in the scene as taxiways or runways.

The LCC phase performs a constraint-satisfaction task. In this phase, knowledge of the structure or layout of the task domain (i.e., airports or suburban housing developments) is used to provide spatial constraints for evaluating consistency among fragment hypotheses. For example, *runways intersect taxiways* and *terminal buildings are adjacent to parking aprons* are examples of the kinds of constraints that are applied to the airport scene segmentation. It is important to assemble a large collection of such consistency knowledge since the results of these tests are used to assemble fragment hypotheses found to be mutually consistent as contexts for further interpretation within the functional area phase.

TABLE II

Washington National Airport (Log No. 405)

SPAM phase	RTF	LCC	FA	MODEL	Total
Total CPU time (hours)	2.5	17.9	7.3	0.33	28.03
Total No. firings	18,319	32,751	1483	1516	54,069
Effective productions/second	2.03	0.508	0.056	1.27	0.536
Total hypotheses	247	N/A	57	1	N/A

3. SOURCES OF PARALLELISM IN PRODUCTION SYSTEMS

We can identify two sources of parallelism in production systems: match parallelism and task-level parallelism (*TLP*). In this section we first discuss existing results in match parallelism. We then discuss task-level parallelism and introduce a taxonomy for describing various approaches to achieving effective speed-ups.

3.1. *Match Parallelism*

In general, production systems spend most of their time (more than 90%) in the match phase of the recognize–act cycle. This makes it imperative that the match phase be speeded up as much as possible. In the past few years, an increasing number of researchers have explored many alternative ways to speed up the match in production systems using parallelism [5, 8, 10, 20, 24, 27, 28].

Our own efforts in speeding up the match have culminated in ParaOPS5 [10, 14], an optimized C-based parallel implementation of OPS5 for shared-memory multiprocessors. ParaOPS5 represents our current technology for achieving match parallelism within systems such as SPAM. This implementation parallelizes the highly efficient Rete [6] match algorithm. ParaOPS5 exploits parallelism at a fine granularity: subtasks execute only about 100 instructions. ParaOPS5

has been able to provide significant speed-ups for OPS5 systems that are match-intensive. Figure 3 shows the speed-ups achieved with our current implementation for three different *match-intensive* systems: Rubik, Weaver, and Tourney. The speed-ups are for an implementation on the Encore Multimax and are reproduced from [10]. Though good speed-ups are achieved in Rubik and Weaver, the speed-up in Tourney is quite low. The speed-ups are a function of the characteristics of the productions in the production system (see [9, 10]).

Although systems such as ParaOPS5 have achieved good speed-ups, the total possible speed-up via match parallelism in current production systems is limited (only 20- to 40-fold [8]). This limit is imposed by:

1. *The recognize–act cycle of OPS5:* The OPS5 model requires a synchronization in its resolve phase. Thus match parallelism is limited to individual cycles; we cannot extract match parallelism across cycles.
2. *Limited match effort per cycle:* In every recognize–act cycle, only a limited number of productions are *affected;* i.e., the match effort per cycle is also quite limited.

Furthermore, the effectiveness of match parallelism is based on the assumption that the match phase dominates the entire computation. However, it is possible that the system under consideration is embedded in some other computationally demanding environment. In such cases, it is necessary to parallelize the rest of the computation besides match. Consider a system that spends only 50% of its time in match. Even if the match is made infinitely fast, the total speed-up possible will be only a factor of 2, as described by Amdahl's law.

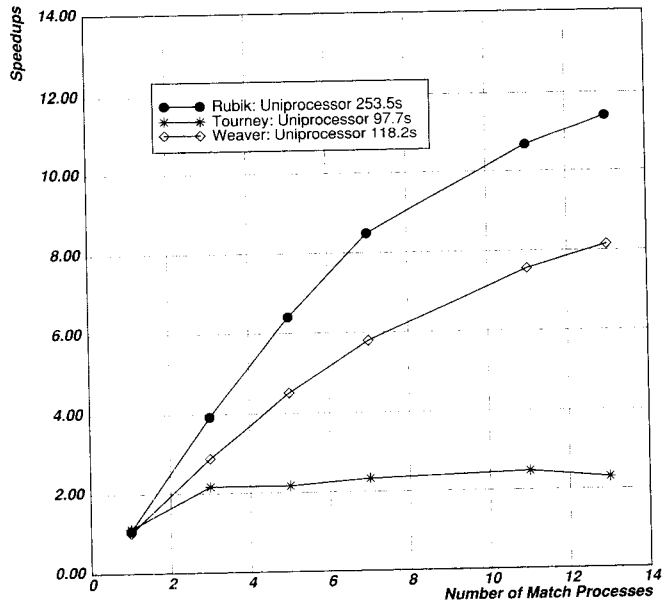

FIG. 3. Speed-ups for OPS5 on the Encore Multimax [10].

3.2. *Task-Level Parallelism*

The limitations of match parallelism described in the previous section prompt the investigation of task-level parallelism. Task-level parallelism has also been referred to as application parallelism [8], concept parallelism [25], and parallel rule firings [13]. Our choice of the term task-level parallelism for this source of parallelism is partly historical and partly dictated by the inadequacy of the other terms to cover the kind of parallelism provided by production systems like Soar [15].

Task-level parallelism refers to parallelism inherent in the given task. Exploiting this task-specific parallelism often requires the system designer to supply knowledge about the particular problem domain to identify the parallelism. Some systems, however, are able to automatically infer the sources of parallelism inherent in a task by a static analysis of the productions that implement the task or a run-time analysis of the execution behavior of the task.

The essential distinction between match parallelism and task-level parallelism is that match parallelism is independent of the task domain. Match parallelism is a feature of the underlying production system architecture that provides the computational engine for executing any production system application. This distinction, which may appear to be somewhat subtle at first glance, is a significant one—especially with respect to the available parallelism. As we saw in Section 3.1, match parallelism is limited by the general nature of production systems. However, task-level parallelism is not constrained by such a fixed bound. It is limited only by the inherent parallelism available in the task itself. In fact, as we shall see, this parallelism can be substantial.

A system exploiting task-level parallelism could be implemented on top of a system exploiting match parallelism. The speed-ups obtained from these two sources are independent and would therefore multiply. Task-level parallelism can be better understood by considering the possible dimensions along which it can be divided. The following is a discussion of three independent dimensions for task-level parallelism.

The first dimension is based on the *implicit* versus *explicit* detection of parallelism. The parallelism is implicit if the system or the compiler has to *extract* parallelism out of the existing OPS5 code. This requires an analysis of the interference [12, 13, 21, 25] caused by firing productions in parallel.

Explicit parallelism refers to providing explicit information to the system for exploiting task-level parallelism. Thus, the system may be supplied with the information that certain parts of a given task can be solved in parallel, or that certain productions can always be fired in parallel. This does require some additional effort by the system designer to analyze and understand the problem domain and select a suitable task decomposition to exploit it.

In implicit parallelism, the system automatically extracts the parallelism in a domain-independent fashion. If it engages in extracting this parallelism at compile-time, then its extraction of parallelism has to be very conservative, as the binding of variables in the productions with working memory elements is not known until run-time. If this parallelism is extracted at run-time, then there are overhead costs paid at run-time to perform the interference analysis. These overheads are sequential, and hence can cause considerable slowdowns. A system for exploiting explicit parallelism is able to avoid these problems.

When the parallelism is implicit, the granularity is usually at the level of productions; it seems difficult to discover a higher level of granularity with implicit parallelism. With explicit parallelism, the user has the freedom to choose the right granularity. The level of granularity is a complex tradeoff of the number of processors available, architectural parameters, variances, data structures, and task management overheads. The granularity issue is discussed in Section 4.3.

The second dimension is based on *synchronous* versus *asynchronous* production firings. Synchronous production-firing systems always require a synchronization in the resolve phase of the recognize–act cycle. All the productions can be matched in parallel. But synchronization must occur in the resolve phase in order to select the production or set of productions to fire next. In the act phase, the selected productions can be fired in parallel.

In asynchronous production-firing systems there is no requirement for a synchronization in the resolve phase across processors. Thus, these systems do not have distinct match, resolve, and act phases across the parallel system.

Synchronous systems are less capable of handling variances in processing times for subtasks [22]. As shown in [22], given a fixed amount of work, in the presence of variance, a synchronous system quickly reaches saturation speed-ups, while an asynchronous system can continue to exploit linear speed-ups. So, in a production system embedded in a computationally intensive environment, if executing the RHS of certain productions takes much longer than others, the performance of the synchronous system will degrade heavily. However, synchronous systems may be preferred in the development and debugging stages.

The third dimension concerns the distribution of productions (rules) and working memory over the processors. There are three choices: *rule distribution, working memory element distribution,* and *no distribution.* These choices of distribution relate to the approach used to partition the processing to best exploit the parallelism.

To better understand this, one might reasonably view the productions as components of an algorithm and working memory elements as the data. The choice of which to partition depends on whether one can readily identify a partitioning and whether its component parts have enough uniformity in their processing times to achieve good parallel performance.

Rule distribution involves a partitioning of the productions in the system among the processors, where each production set has its own conflict set. This distribution could be done automatically [21] or with the help of the user. However, optimal distribution of productions among processors is a difficult problem [24].

A second approach is to allocate all the productions to each processor; the working memory elements are then distributed among the processors. In contrast with rule distribution, which is typically determined at compile time, working memory element distribution is usually performed dynamically as the system executes and thus incurs some extra run-time overheads. However, these overheads may be very small.

The third approach is no distribution and is more or less a default choice in which no attempt is made to partition either productions or working memory across the processors. Some systems do not admit to a disciplined partitioning scheme on these bases. In this case, one often finds more ad hoc approaches to opportunistically exploit parallelism.

Table IV shows the various dimensions and the classification of various parallel rule-firing systems along these dimensions. These dimensions will help to investigate the task-level parallelism in SPAM/PSM. The table uses the names of authors to represent systems that do not have any names. We also indicate the third dimension that classifies the type of distribution used: rule distribution, working memory element distribution, or none.

The SPAM/PSM system is the system described in this

TABLE IV

Dimensions of Task-Level Parallelism

Dimensions	Synchronous :: Distribution		Asynchronous :: Distribution	
Implicit	Ishida and Stolfo [13]	Rule	CREL [21]	Rule
	Ishida [12]	Rule		
	Oshisanwo and Dasiewicz [25]	Rule		
	Baker and Miller [3]	Rule		
Explicit	Soar [15]	None	SPAM/PSM	WME

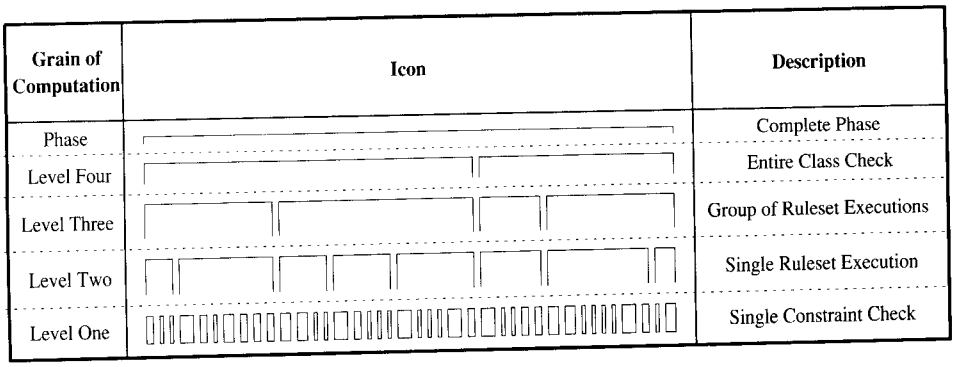

FIG. 4. Levels of processing in SPAM LCC.

paper; the design choices are discussed in detail in Section 4. These dimensions are not intended to be binary; rather, different systems could take different positions along a continuum in these dimensions. However, in the interests of clarity, the table makes a binary division. For instance, the system in [25] is classified as using implicit parallelism—however, it uses some explicit parallelism. It should be noted that except for Soar and SPAM/PSM, all other systems present simulation results on miniproduction systems (with 50 or less productions).[2]

Table IV shows that much of the work in task-level parallelism is based on the implicit and synchronous dimensions of task-level parallelism [13, 12, 25, 3]. In [3], an interesting extension for OPS5 is presented to allow parallel execution of multiple actions. In [21], an attempt is made to introduce asynchronous execution. They allow asynchronous execution of rules in different groups (called *clusters*), but require synchronization within a single cluster. Most of these systems make use of the techniques introduced in [13] for discovering implicit parallelism. The SPAM/PSM system differs from all these systems in that (i) it uses explicit decomposition and (ii) it uses working memory element distribution for achieving parallelism. Our results will illustrate that the SPAM/PSM approach is effective in achieving good speed-ups via task-level parallelism.

4. DESIGN METHODOLOGY

This section develops a methodology for applying task-level parallelism within the context of SPAM. As discussed in Section 2.2, we will concentrate on the RTF and LCC phases of SPAM for parallelization. The LCC phase, because of its computationally intensive nature, was the first one to be parallelized. Therefore, our methodology is described in detail for LCC; the methodology for the RTF phase is similar and only the relevant results from that phase are presented.

[2] While we reported results of match parallelism in Soar from a real implementation in [28], the results of task-level parallelism in Soar are based on simulations [8].

In examining a given problem domain and arriving at a system design, we are not so much faced with making a choice of which approach to use in each of the dimensions of task-level parallelism described in Section 3.2. Rather, we are faced with determining what the system characteristics are and what the opportunities are for effectively exploiting the inherent parallelism. The dimensions of task-level parallelism serve to focus this identification of the computational characteristics of the problem domain and then guide the design of the system to best exploit the parallelism based on these characteristics.

However, as pointed out in Section 3.2, the alternatives in some of the dimensions of task-level parallelism have attendant problems which influence the view one takes in characterizing the problem domain. For example, in the distribution dimension, the difficulty in determining optimal rule partitioning motivates one to avoid it. Thus a bias is made for working memory distribution if the system can be characterized in this way.

In this section, our design methodology begins with a characterization of the SPAM computation. With this characterization, we identify the dimensions along which we will exploit the task-level parallelism in SPAM. Finally, having identified an explicit decomposition of SPAM, we address the selection of the right level of decomposition to use in the parallel system and conclude with a summary of our decomposition methodology.

4.1. *Characterizing the SPAM Computation*

As described in Section 2.2, the SPAM system exhibits a hierarchical processing structure with distinct levels of organization. For a given SPAM phase, the organization of the processing is characterized by the set of objects and their relationships according to the level of detail at which they are being examined. Higher levels tend to aggregate objects and generalize relationships while lower levels identify further refinements of these. We can use this knowledge about the task domain to specify several hierarchical task decompositions of the problem in which parallelism can be exploited.

The LCC phase applies geometric knowledge (constraints) from the selected domain (airport here) to the set of interpretations made from the dataset. This application of geometric knowledge can be logically decomposed into several levels, where the tasks within each level are independent and can be performed in parallel. The decomposition proceeds from higher to lower levels of granularity at which the lower levels expose more details of the computation and subdivide the processing into smaller pieces or tasks. This is illustrated in Fig. 4. These levels of decomposition are described below:

- *LCC phase:* At the highest phase level, the computation is for the entire LCC phase.
- *Level 4:* The phase level computation may be decomposed into tasks at Level 4, where each task applies multiple constraints to a single class of objects. For instance, a task may apply multiple constraints to all objects of class terminal building.
- *Level 3:* A single task at Level 4 may be decomposed into multiple tasks at Level 3. A task at Level 3 applies multiple constraints to a single object within the class of objects selected at Level 4. For example, a Level 3 task may apply multiple constraints to a single terminal building object.
- *Level 2:* A single task at Level 2 involves applying a single constraint to a single object. Thus, a task at Level 2 may apply a constraint such as access roads lead to terminal buildings to a single terminal building chosen for a task at Level 3.
- *Level 1:* A single task at Level 2 may have several components to check in applying a constraint to an object. Thus a constraint such as access roads lead to terminal buildings requires several roads be checked against the terminal building. A task at Level 1 would perform one of these constraint components.

4.2. *Identifying the Dimensions of Task-Level Parallelism*

The characterization of SPAM in Section 4.1 yielded an explicit decomposition of the system into several hierarchical levels. Thus, the characteristics of SPAM fit the requirements for exploiting task-level parallelism along the explicit dimension described in Section 3.2. Within a decomposition level, each task involves the firing of from 3 to approximately 100 productions. As mentioned in Section 3.2, an implicit approach to extracting parallelism would make it difficult to obtain parallelism at a higher level of decomposition than individual production firings. Therefore, for this application, an explicit approach to parallelism is appropriate.

The second dimension of task-level parallelism addresses the issue of synchronous versus asynchronous execution. In our explicit decomposition, the tasks at each level are independent and there is no synchronization requirement. This allows us to exploit asynchronous production firings across the parallel system. As stated in Section 3.2, asynchronous models help in reducing the impact of variance.

The final dimension of task-level parallelism is concerned with production versus working memory partitioning. As discussed earlier in this section, working memory partitioning is to be preferred. The processing on the data objects in the task decompositions provides a natural mapping to working memory element distribution. We also see in the next subsection that the variance in processing time among the tasks within several of the task decompositions is small. This uniformity in processing time among the tasks is useful in achieving good parallel performance and further supports the use of the more easily specified working memory partitioning.

4.3. *Choosing the Right Level of Decomposition*

With an explicit approach to parallelism, the choice of the right level of decomposition, or the right granularity, for parallelization must be made. This choice is determined by several factors:

1. *Task granularity:* As the average time per task gets smaller, task management overheads will have a greater impact and communication overheads and system resource contention will become more of a bottleneck.
2. *Ratio of tasks to processors:* The achievable parallelism is bounded by the number of available processors. At lower task to processor ratios, a large variance in task processing time will have a negative impact on processor utilization and the speed-ups obtained from parallelism. With higher ratios, the impact is less pronounced.
3. *Coefficient of variance:* Defined as (standard deviation/average), this provides a means of normalizing, for different levels of decomposition, the effect of variance in task granularity on processor utilization. A high coefficient of variance will reduce processor utilization, resulting in lower speed-ups. This effect is more severe in synchronous systems.
4. *Decomposition effort:* This is a somewhat subjective measure. Proceeding down the hierarchy of levels, each task at the current level must be decomposed into several tasks at the next level of granularity. Usually, more work is required to specify the decomposition and design an implementation at the lower levels. The benefits of the additional parallelism that can be achieved at a lower level relative to the effort required must be assessed.

In order to choose the right level of decomposition at which to parallelize the SPAM LCC phase, we instrumented the SPAM system to obtain measurements at each level for the number of tasks and their run-time average, standard deviation, and coefficient of variance. The results of these measurements for each of the three airport datasets are presented in Tables V, VI, and VII.[3]

[3] Since the analysis is performed using the original, expensive Lisp-based SPAM system, we have extracted a representative subset of the three airport datasets to drive the analysis.

TABLE V

Average, Standard Deviation, and Coefficient of Variance for SF

Level	Average time/task (s)	Standard deviation (s)	Coefficient of variance	Number of tasks
4	875.27	525.92	0.601	9
3	65.65	29.51	0.449	120
2	20.90	8.48	0.406	377
1	0.489	0.0782	0.159	16,104

TABLE VII

Average, Standard Deviation, and Coefficient of Variance for MOFF

Level	Average time/task (s)	Standard deviation (s)	Coefficient of variance	Number of tasks
4	165.60	121.20	0.732	9
3	20.07	8.02	0.399	74
2	5.57	2.43	0.436	268
1	0.349	0.0455	0.130	4274

Using information from Tables V, VI, and VII, the appropriate level of granularity can now be chosen for our target architecture, an Encore Multimax with 16 processing elements. For Level 4, the task-to-processor ratio is smaller than one, so we immediately rejected pursuing parallelism at this level. Levels 3 and 2 are very similar to each other in that they have enough tasks, their variances are not large, and the task granularities are much larger than the expected task management and communication overheads. Both levels, therefore, seemed to us to be worthwhile candidates. Level 3 seemed somewhat more desirable as less effort appeared to be required of us to achieve amounts of parallelism similar to that available in Level 2.

Level 1 was rejected for several reasons. First and most importantly, the additional effort involved in decomposing the system at the granularity of Level 1 would not allow us to achieve any more parallelism than at Level 2 or 3 because of the limitation on the number of processors. Second, the task granularity is much smaller and thus closer to the overheads for task management and communication than any of the other levels. Finally, the task-to-processor ratio is on the order of 1000. This can have a detrimental effect due to the initialization overhead. Our conclusion, then, was to exploit parallelism at the granularity of Level 2 or 3.

Our decomposition methodology can be summarized as follows:

- Analyze the baseline system and determine where the time is going.
- Determine if the explicit dimension of task-level parallelism (Section 3.2) is appropriate.
- Characterize the computation in terms of independent task decompositions at different granularities.
- Obtain measurements of the system characteristics for each level of decomposition.
- Analyze the measurements to select a level of decomposition for parallelization.

A similar analysis of the RTF phase was performed. This resulted in a decomposition level providing approximately 60–100 tasks in the three datasets at roughly the same granularity as Level 2 of the LCC phase, and a low coefficient of variance of approximately 0.3.

TABLE VI

Average, Standard Deviation, and Coefficient of Variance for DC

Level	Average time/task (s)	Standard deviation (s)	Coefficient of variance	Number of tasks
4	1308.66	641.72	0.490	9
3	78.51	30.48	0.388	150
2	24.04	9.51	0.396	490
1	0.430	0.0677	0.157	27,399

5. SPAM/PSM IMPLEMENTATION

This section describes the SPAM/PSM system for exploiting task-level parallelism in SPAM. The system is built on top of the ParaOPS5 system described in Section 3.1. The SPAM/PSM system is implemented on an 16-processor Encore Multimax, a shared-memory multiprocessor based on the National Semiconductor NS32332 processor (rated at approximately 1.5 MIPS), running the MACH operating system.

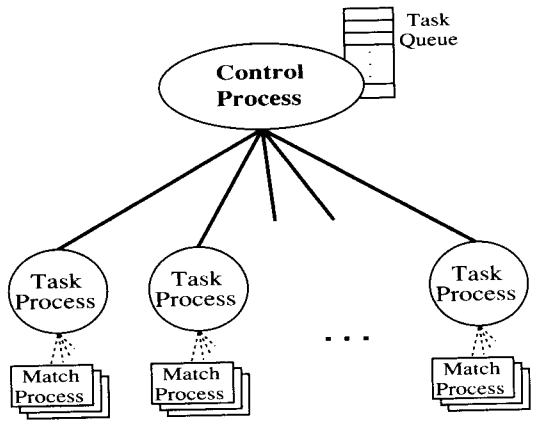

FIG. 5. Organization of the SPAM/PSM system.

5.1. *SPAM/PSM Architecture*

Figure 5 gives a process hierarchy view of the SPAM/PSM system. Viewed from the top level, the execution model consists of a *control process,* a set of *task* processes, and a *queue of tasks* to be executed. The size and number of tasks in the queue reflect the level of decomposition chosen for the particular SPAM phase. As part of *initialization,* the control process builds the queue of tasks. It then forks the task processes and, once they have completed all the tasks, collects from them the results that will be passed on to the next SPAM processing phase.

Each of the task processes is a complete and independent ParaOPS5 system. Thus, each task process has its own working memory, conflict set, etc. Each task process has a production memory, which represents all the productions in the system, and effectively has a copy of the initial working memory supplied by the control process. At system initialization time, each task process can also fork a set of match processes (see Fig. 5) which will perform the match in parallel.

Execution of a particular SPAM phase involves a task process removing a task from the queue and executing its ParaOPS5 system on that task. The task itself is just a working memory element, which initializes the production system of the process. Thus, each task can be characterized as the execution of an independent OPS5 program.

In the absence of the match processes, a task process performs the usual ParaOPS5 role of match, conflict-resolution, and production firing, to carry out the OPS5 recognize–act cycle. If dedicated match processes are present, they perform the match instead, providing a second and independent axis of parallelism in the SPAM/PSM system. When there are no productions left to fire, the task is complete, and the task process goes to the queue for another task.

Thus, the SPAM/PSM system realizes our specifications:

1. *Explicit parallelism:* The decomposition of the problem is explicitly specified. The task queue is initialized with independent tasks, depending on the level of decomposition, in the beginning of the run.
2. *Asynchronous production firing:* All the task processes are independent ParaOPS5 systems. Therefore, these processes can fire productions without synchronizing with each other.
3. *Working memory element distribution:* Each task process has a copy of the entire set of productions. The working memory is distributed among the various task processes.

5.2. *Measurement Techniques*

The control process previously described is used to monitor and time the processing. Measurement begins at the point after which the control process has built the task queue and forked the task processes, and all the task processes have performed their initializations. Speed-ups are computed by comparing the measured execution time against the execution time of the BASELINE version, which consists of the control process, one task process, and no dedicated match processes.

Because of the 16-processor limit, we measure the effects of task-level parallelism and match parallelism in isolation. We allocate one processor for the control process, which is used only to time and not to perform tasks, and we allow one processor to the operating system. This permits us to vary the number of task processes from 1 to 14 in the isolated measurement of task-level parallelism. Next we measure the effect of match parallelism in isolation by using a single task process and varying the number of dedicated match processes from 0 to 13.

We are then able to use these two separate measures of task-level parallelism and match parallelism to predict the combined effect of the two. However, with 14 available processors, we are able to test only a subset of the possible combinations. For example, 4 task processes, each having 2 dedicated match processes, use 12 processors (4 + (4*2)). Thus, dedicating 3 match processes requires 16 processors (4 + (4*3)) and, therefore, cannot be accommodated.

6. RESULTS AND ANALYSIS

This section presents the results of task-level parallelism in the LCC and RTF phases on three different airport datasets: SF, DC, and MOFF. As described above, the speed-ups are obtained for applying task-level parallelism and match parallelism in isolation and then for a combination of the two. Again, we provide detailed results for LCC and then provide the summary of results for RTF.

It is important to note that all the speed-ups are computed against a baseline system which represents an optimized uniprocessor implementation. The original SPAM system is implemented in Lisp, using an unoptimized Lisp-based OPS5, and has an interface to C in order to perform geometric computations and other specialized vision processing through external calls made in the RHS. We ported this entire system to C and ParaOPS5 to obtain our baseline system. This baseline system itself provides approximately a 10- to 20-fold speed-up over the original Lisp-based implementation on the three datasets used here.

6.1. *The Baseline System for the LCC Phase*

The baseline version of the LCC phase of the system uses a single task process to execute all the tasks in the system. The results from this version are given in Table VIII and provide a picture of the magnitude of the LCC phase. The column marked Dataset gives the name of the airport and the decomposition level used. The column marked Total time shows the total time to execute all the tasks from the queue for the given number of tasks executed. The average

TABLE VIII

Measurements for Baseline System on the Datasets[a]

Dataset	Total time (s)	Number of tasks	Average time/task (s)	Prods. fired	Effective prods./ second	RHS actions
SF						
level 3	1433	283	5.07	33,475	23.36	42,383
level 2	1423	941	1.51	32,251	22.66	41,159
DC						
level 3	988	151	6.55	20,059	20.30	31,205
level 2	956	490	1.95	19,418	20.31	30,564
MOFF						
level 3	991	209	4.74	22,203	22.40	23,637
level 2	973	700	1.39	21,294	21.88	22,728

Note. Represents the optimized, ParaOPS5-based, uniprocessor version.
[a] These datasets are larger than those shown in Tables V, VI, and VII.

time per task is then shown in the next column. Finally, we further characterize the LCC phase with the total number of productions fired (Prods. fired), rate of firing (Effective prods./second), and RHS actions performed (RHS actions).

The execution times in Table VIII provide the basis for computing all of the speed-ups. For a given airport dataset, there is a small difference in the total execution time between the two levels of decomposition. These differences arise due to the differences in the initial set of productions fired for generating the *tasks* for the two levels.

6.2. *Speed-Ups Due to Task-Level Parallelism in LCC*

The results of applying task-level parallelism to LCC are shown in Fig. 6. The speed-up curves show near linear speed-ups for both levels of decomposition. The speed-ups within a level are almost the same among the three airport datasets. The maximum speed-up achieved using 14 processors is 11.90-fold in Level 3 and is 12.58-fold in Level 2.

Across the two levels, we see that the curves are consistently better in Level 2, although by only a small factor (less than 10%). While the difference is small, Level 3, with its higher granularity, was expected to have the edge in speed-up, since its task management overheads would be lower. However, the task management overheads in both levels are very low: less than 0.25 s, or less than 0.1% of the processing time for all the tasks. Moreover, the coefficient of variance for tasks at both levels was seen to be the same in Section 4.3.

Further investigation of the individual processing times of the tasks in the queue showed that there are a few tasks in each level that have execution times that are an order of magnitude larger than the average task in that level. Some of these tasks occur at the end of the task queue and create a tail-end effect in which processor utilization is low at the end of the phase. The relative disparity of these large tasks is greater within Level 3 and thus accounts for the slightly better speed-ups in Level 2.

One way to both negate this disparity and reduce the tail-end effect would be to use a separate task queue for the larger tasks and process them at the beginning of the phase. This would result in better processor utilization and thus better speed-up curves in both levels. SPAM can provide the necessary information to identify the sizes of the tasks. This and other related issues of scheduling tasks are subjects for future work.

6.3. *Speed-Ups Due to Match Parallelism in LCC*

Figure 7 shows the results for applying match parallelism to each of the tasks in a parallel decomposition for Level 2. Match parallelism is obtained by dedicating processes to perform the match within the OPS5 recognize–act cycle. Since the baseline version of the system has only a task process and no dedicated match processes, it is represented in the graph at position 0 on the horizontal axis. From the graph, we see that applying match parallelism to the LCC phase yields very different speed-up results from those achieved using task-level parallelism. As stated in Section 3.1, the theoretical maximum speed-up that can be obtained is limited according to the percentage of total execution time spent in match, which is less than 50% in LCC.

The dotted lines on the graph show the theoretical speed-up limits. For Level 2, these limits are 1.99, 1.36, and 1.55 for SF, DC, and MOFF, respectively. We were able to obtain respective speed-ups of 1.71, 1.29, and 1.43 which represent 86, 95, and 92% of the corresponding asymptotic limits. In all three cases, the speed-ups peaked using six or less match processes. The results for Level 3 are similar (see [11]).

6.4. *Multiplicative Speed-Ups in LCC*

To validate the multiplicative effect of the two independent axes of parallelism [26], the system was run using task-level and match parallelism in consort. While the scope of the experiments was limited by the small number of processors, the speed-ups obtained in these combined runs were consistent with the speed-ups predicted by the multiplication of speed-ups from the two separate sources. Table IX shows the results of some of these combined runs on SF for Level 2. The top row of the table varies the number of dedicated match processes from 0 to 4. The left column of the table varies the number of task processes from 1 to 7. The first row of numbers in the table gives the speed-ups from match parallelism in isolation. The first column of numbers in the table gives the speed-ups from task-level parallelism in isolation.

The table entry at ($Task_1$, $Match_0$) represents the baseline version of the system. Each of the other table entries shows the achieved multiplicative speed-up from the combined sources with the predicted speed-up in parentheses directly below. For example, the entry ($Task_4$, $Match_2$) represents the use of 4 task processes with each having 2 dedicated

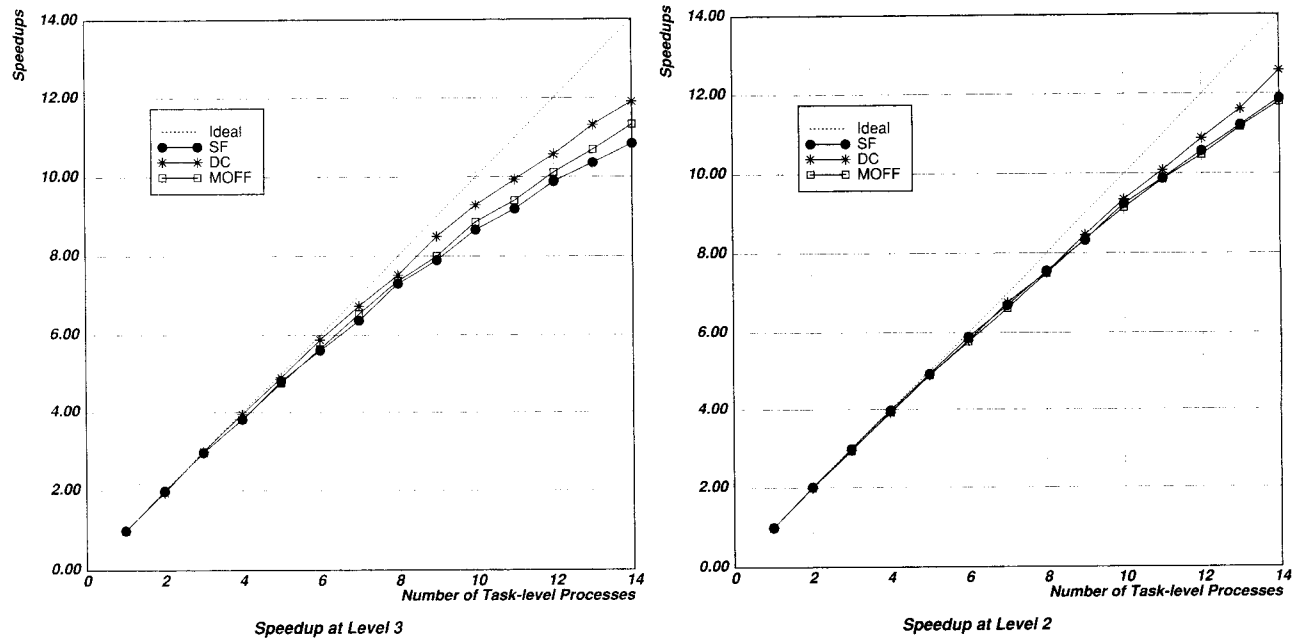

FIG. 6. Speed-ups varying the number of task-level processes.

match processes. The achieved speed-up for this configuration is 5.82-fold and the predicted speed-up is 5.96 (3.98 * 1.50). Table entries marked with an asterisk could not be measured due to a lack of processors on the machine (see Section 5.2). For example, ($Task_4$, $Match_3$) requires 17 processors: 1 control process, 4 task-level processes, and 12 (=4 * 3) dedicated match processes. The table shows the achieved speed-ups to be very close to the predicted speed-ups. Similar results were obtained for DC and MOFF.

6.5. *Results from the RTF Phase*

From a static analysis of the productions, the RTF phase was expected to be match intensive and hence provide speed-ups from match parallelism in step with the results of traditional OPS5-based systems. Recall that the RTF phase of SPAM is a traditional heuristic classification task, mapping regions in an image to interpretations called fragments. However, measurements revealed that production system match constituted 60% of the execution time. This is closer to more traditional OPS5 systems; although traditional OPS5 systems spend an even higher percentage of their time in match. Our results showed that, as expected, the speed-ups from match parallelism are limited to a factor of 2.5 as shown in Fig. 8. However, task-level parallelism still provides good speed-ups.

Although task-level parallelism provides good speed-ups in the RTF phase, these speed-ups appear to be a little lower than the speed-ups provided in the LCC phase. This occurs partly because RTF tasks are fewer and finer-grained than the LCC tasks. Part of the reason is also that these results come from a more optimized version of SPAM/PSM. This phenomenon of measuring speed-ups on somewhat different versions of the two phases is actually a part of a more general phenomenon—we have been working with the SPAM/PSM system for almost 3 years now. When working with a large system like SPAM/PSM over an extended period of time, speed-ups available in the system vary. Such variations have

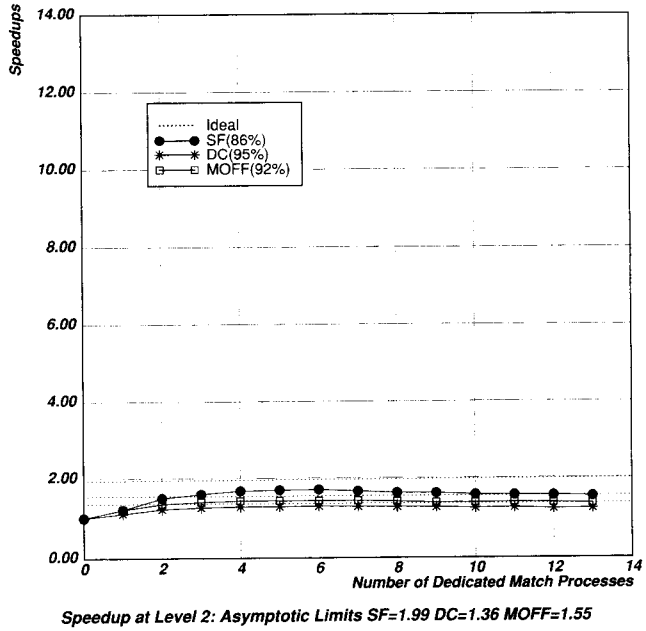

FIG. 7. Speed-ups varying the number of match processes.

TABLE IX

Multiplicative Speed-Ups in SPAM/PSM for SF Level 2

	$Match_0$	$Match_1$	$Match_2$	$Match_3$	$Match_4$
$Task_1$	1	1.21	1.50	1.60	1.68
$Task_2$	2.01	2.42 (2.43)	2.97 (3.01)	3.16 (3.22)	3.30 (3.37)
$Task_3$	2.98	3.57 (3.60)	4.42 (4.46)	4.73 (4.78)	* (5.01)
$Task_4$	3.98	4.73 (4.81)	5.82 (5.96)	* (6.37)	* (6.69)
$Task_5$	4.93	5.82 (5.95)	* (7.39)	* (7.89)	* (8.28)
$Task_6$	5.89	6.98 (7.12)	* (8.83)	* (9.42)	* (9.90)
$Task_7$	6.70	8.04 (8.09)	* (10.05)	* (10.72)	* (11.26)

Note. Parenthesized numbers are the predicted speed-ups.

occurred due to optimizations and changes in the underlying SPAM system, the SPAM/PSM environment, as well as changes in the operating system. These changes typically reduce the granularity of the computation, reducing the speed-up by a small amount. Therefore, we expect that if the speed-ups in RTF had been computed with the earlier version of SPAM/PSM, they would have been somewhat higher.

6.6. *Performance Summary and Predictions*

The speed-up curves for task-level and match parallelism graphically indicate that the benefits from task-level parallelism are much more significant than from match parallelism. We believe that the potential for additional speed-ups in SPAM from task-level parallelism is quite high; an expectation of 50- to 100-fold does not seem unreasonable, since:

1. The tasks in all decompositions are independent of one another.
2. Several hundred tasks are available in all decompositions.
3. The task queue management overheads measured are very low, especially with respect to the task granularity, and thus are not a factor. A centralized task queue may potentially become a bottleneck for an increasing number of processes; currently, this is not the case.

Despite this optimism, one can see that the speed-up curves in Fig. 6 do depart somewhat from the ideal. We were interested in understanding just what factors in our system or in the operating system might be responsible for the performance difference. We identified several issues (some touched upon earlier) for investigation:

- Tail-end effect (see Section 6.2)
- Task queue contention
- Memory management overheads
- System call overheads
- Measurement error
- Page faulting in the virtual memory system
- Context switching and processor scheduling

Our explorations with an instrumented version of our system along with a synthetic workload indicate that none of these issues pose a significant impact for the SPAM system. One can speculate that the performance difference is just a feature of reaching the performance limits of the machine when approaching the maximum number of processors. This is still a subject under investigation.

7. EXPERIMENTS WITH THE VIRTUAL SHARED-MEMORY SYSTEM

In Section 6.2, the speed-up curves clearly indicate more parallelism to exploit if only a sufficiently large multiprocessor were available. The limitations on the number of processors on the Encore Multimax led us into investigating the *virtual shared-memory* system. A virtual shared-memory system can provide a virtual address space among all processors in a loosely coupled multiprocessor [7, 16]. Our local computing environment has two separate Encore Multimax machines, each with 16 processors. Recently, the *shared-memory server* became available on these machines [7], providing a 32-processor (16 from each Encore) virtual shared-memory machine. However, the MACH kernel and the virtual shared-memory system tend to occupy 2 processors on each of the Encores; thus, in reality this is a 28-processor virtual shared-memory machine. The latency across the two Encores with the virtual shared-memory is reported to be 50 ms [7], much lower than the granularity of the SPAM/PSM tasks. Furthermore, the SPAM/PSM tasks are independent, with the processes requiring minimal communication through the task queue. Therefore, the SPAM/PSM system appeared to be an ideal candidate for experimenting with the virtual shared-memory system.

Virtual shared-memory systems have only recently started receiving attention in the literature [7, 16]. In particular, there appear to be no reports in the literature on experiences with such a system for large-scale applications like SPAM. Therefore, we think our experiences with virtual shared memory will be of interest.

Introducing the virtual shared memory in the SPAM/PSM system turned out to be more complex than our initial expectation. In the virtual shared-memory system, the programmer has to be sensitive to the allocation of data-structures to pages to avoid contention. This contention problem was made acute due to *false contention,* i.e., two or more processes across the Encores contending for objects located

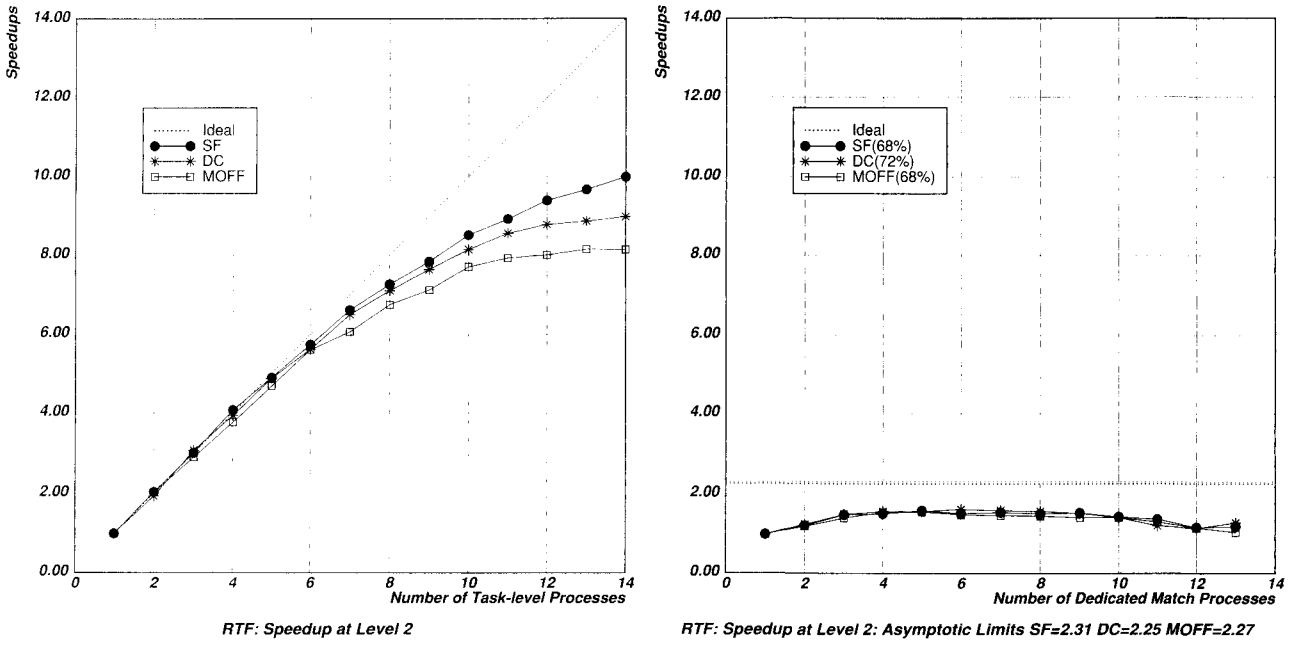

FIG. 8. Speed-ups varying the number of task-level and match processes for the RTF phase.

on the same page though not shared between them. At first, no attention was paid to this problem—however, the overhead incurred from constantly page faulting across the network due to false contention brought our system to a halt just during the initialization. Two separate approaches were employed to solve this problem. We organized our data-structures in the address space in order to eliminate the contention across Encores. The designers of the virtual shared-memory system proceeded to provide some optimizations and heuristics in the shared-memory server to minimize the amount of data sent over the network to service a page fault. For example, instead of shipping a full 8K page, the server ships only small, 64-byte segments of the page that has been modified.

After the elimination of the false contention problem, and the introduction of other optimizations, real speed-ups were possible. The speed-up results are shown in Fig. 9. The following observations can be made about these results:

1. Two separate speed-up curves are shown in Fig. 9. The first one was obtained from the virtual shared-memory system. The second was obtained from the pure task-level parallelism system, i.e., the system without the virtual shared memory (the speed-up curve from this system is indicated in Fig. 9 as Pure TLP). The comparison of these two curves shows that real speed-ups are indeed possible with the virtual shared-memory system—underscoring the usefulness of the virtual shared-memory system for large applications. However, these speed-ups came only after many rounds of optimizations in our system and in the virtual shared-memory server. A point here is that the speed-up for the pure task-level parallelism system is seen to be lower than the speed-ups shown in Fig. 6. This change in speed-up occurred because these experiments were performed with a more recent (and optimized) version of SPAM/PSM. The speed-ups for the virtual shared-memory system are also from this recent

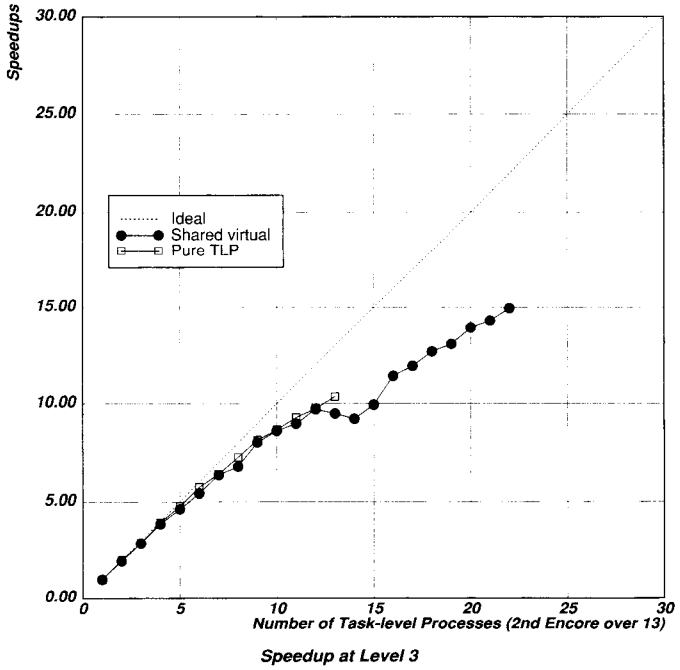

FIG. 9. Speed-ups varying the number of processes using the virtual shared-memory server.

optimized version (see Section 6.5 for a discussion of this effect).

2. As we can see from Fig. 9, the speed-ups for the virtual shared-memory system are close to the pure task-level parallelism system, as long as all the processes are running on a single Encore. As soon as we add a process on the remote Encore, we see an abrupt change in the curve—which produces a translational effect on the curve. This translational effect is equivalent to the loss of about 1.5 processors. This performance hit comes from the overheads of communication across the network.

3. We were able to provide results for only 22 processors—13 on the first Encore and 9 on the second. Our application placed severe demands on the MACH kernel, preventing us from using all the processors. These robustness issues are being addressed by the designers of the virtual shared-memory system.

4. In our final optimized system, only a single task queue is present. The contention for this task queue is minimal. Separate experiments were performed on these task queues, which indicated that introducing separate task queues (one for each Encore) would not change the results.

5. Resolving the performance issues raised in Section 6.6 could mean additional improvement for the virtual shared-memory system as well.

8. SUMMARY AND CONCLUSIONS

Most investigations of speeding up production systems via parallelism have been focused on match parallelism. While these have provided good speed-ups, the total speed-up available from this source is limited and is insufficient to alleviate the problems of inefficiency in large-scale production systems. In fact, in SPAM, speed-up from match parallelism is limited to about a factor of 2. In this paper we have focused on task-level parallelism. The speed-ups obtained from task-level parallelism will multiply with those obtained from match parallelism. We characterized task-level parallelism in production systems along three dimensions and, from that, selected an explicit, data-driven, asynchronous approach for exploiting it. The system we presented, SPAM/PSM, is a real, computationally demanding, high-level vision system that relies on knowledge-based reasoning. With the SPAM/PSM system, we showed that an explicit approach to task-level parallelism can yield significant speed-ups.

We presented a methodology for obtaining a parallel task decomposition in SPAM/PSM. We parallelized two of SPAM's phases and obtained near linear speed-ups with a maximum of over 12-fold using 14 processors. Further, with the use of the virtual shared-memory system, even higher speed-up was obtained. The results obtained indicate that speed-ups on the order of 50- to 100-fold from task level parallelism might be realized on a machine with a comparably large number of processors. Note that in parallelizing both these phases, it was possible to exploit the framework of a control process queueing tasks for several task processes as shown in Fig. 5. However, other phases may require a different framework, e.g., one that exploits pipelining tasks from one task process to another. We believe that the success achieved with the SPAM/PSM system gives hope to designers of other rule-based systems to realize systems with much lower execution times by applying task-level parallelism. Also the potential for very large speed-ups indicated here should serve as encouragement to the designers of large-scale multiprocessor systems.

A relevant consideration is the difficulty in specifying an explicit decomposition in our methodology for task-level parallelism. Our experience has been that this decomposition is fairly straightforward and can be arrived at relatively quickly. We demonstrated the applicability of this explicit decomposition in two diverse phases of SPAM. One of these phases performs a traditional AI classification task (RTF), while the second performs a constraint-satisfaction task (LCC). The RTF phase fits more closely into the framework of a traditional OPS5 system. In both these phases, the explicit decomposition is made based on the data upon which the system must operate—giving rise to levels of decomposition. We saw that the choice of the correct level at which to exploit parallelism is based upon a number of factors; among these are the task granularity, task management and communication overheads, the variance in task processing times, and the ratio of total tasks to processors. Recently, we have finished the decomposition of the third phase of SPAM, functional area. FA performs an evaluation/grouping task, and again fits more closely into the framework of a traditional OPS5 system. We are currently investigating the speed-ups available in FA.

There is a tradeoff involved in explicit parallelism. It avoids possible run-time overheads or synchronization issues. SPAM/PSM allows asynchronous production firings, escaping the synchronization requirement of the OPS5 recognize–act cycle. However, there is a cost associated with it in the form of additional analysis required of the system designer to generate a decomposition.

On the whole, the framework for exploiting task-level parallelism presented in this paper seems most suitable for parallelizing knowledge-intensive systems that exhibit weak interaction between the individual subtasks of the task. This framework is especially useful for systems with a large computational demand separate from the demand imposed by match.

9. FUTURE WORK

Although all of SPAM is currently implemented in ParaOPS5, the benefits from task-level parallelism are not fully available to the SPAM researchers. Providing ParaOPS5

with task-level parallelism as a useful tool requires moving beyond the current experimental setup for task-level parallelism. For instance, currently the initialization subphase within the local-consistency phase consumes a large amount of processing time, and needs to be optimized and/or parallelized.

Results from Section 6 show that large amounts of parallelism can be exploited in SPAM, and thus significantly larger numbers of processors could be employed in exploiting the parallelism. Shared-bus multiprocessors like the Encore Multimax cannot support such a large number of processors. Investigations of virtual shared-memory systems were motivated due to this concern and they have provided promising results. However, we need to obtain speed-ups with even larger numbers of processors. We also need to analyze the current speed-ups from the virtual shared-memory system in detail, accounting for the various overheads.

Our long term plan is the investigation of task-level parallelism in systems besides SPAM. We hope such investigations will help us define a general methodology for exploiting task-level parallelism in production systems.

ACKNOWLEDGMENTS

We thank Anurag Acharya, Charles Forgy, Anoop Gupta, Brian Milnes, and members of the Digital Mapping Lab for helpful discussions. Special thanks go to Joseph Barrera and David Black of the MACH group for their help with the virtual shared memory. Matt Diamond provided invaluable programming assistance, and Kathy Swedlow helped make our prose a bit more understandable. This research was partially supported by the Air Force Office of Scientific Research, under Grant AFOSR-89-0199, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-87-C-1499. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Air Force Office of Scientific Research, or of the Defense Advanced Research Projects Agency, or of the United States Government.

REFERENCES

1. Acharya, A., and Tambe, M. Production systems on message passing computers: Simulation results and analysis. *Proc. International Conference on Parallel Processing.* Aug. 1989.
2. Bachant, J., and McDermott, J. R1 Revisited: Four years in the trenches. *AI Magazine* **5,** 3 (1984), 21–32.
3. Baker, J. W., and Miller, A. R. A Parallel production system extending OPS5. *Proc. Frontiers of Massively Parallel Computing.* 1990, pp. 110–118.
4. Brownston, L., Farrell, R., Kant, E., and Martin, N. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA, 1985.
5. Butler, P. L., Allen, J. D., and Bouldin, D. W. Parallel architecture for OPS5. *Proc. Fifteenth International Symposium on Computer Architecture.* 1988, pp. 452–457.
6. Forgy, C. L. OPS5 user's manual. Tech. Rep. CMU-CS-81-135, Computer Science Department, Carnegie Mellon University, July 1981.
7. Forin, A., Barrera, J., and Sanzi, R. The Shared Memory Server. *Proc. USENIX—Winter 89.* 1989, pp. 229–243.
8. Gupta, A. Parallelism in production systems. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, Mar. 1986.
9. Gupta, A., Forgy, C. L., Kalp, D., Newell, A., and Tambe, M. Parallel OPS5 on the Encore Multimax. *Proc. International Conference on Parallel Processing.* Aug. 1988, pp. 271–280.
10. Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A. Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis. *Internat. J. Parallel Programming* **17,** 2 (1989).
11. Harvey, W., Kalp, D., Tambe, M., McKeown, D., and Newell, A. Measuring the effectiveness of task-level parallelism for high-level vision. Tech. Rep. CMU-CS-89-125, School of Computer Science, Carnegie Mellon University, Mar. 1989.
12. Ishida, T. Methods and effectiveness of parallel rule firings. *Proc. Sixth IEEE Conference on Artificial Intelligence Applications.* 1990.
13. Ishida, T., and Stolfo, S. Towards the parallel execution of rules in production system programs. *Proc. International Conference on Parallel Programming.* Aug. 1985, pp. 568–574.
14. Kalp, D., Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K. Parallel OPS5 user's manual. Tech. Rep. CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, Nov. 1988.
15. Laird, J. E., Newell, A., and Rosenbloom, P. S. Soar: An architecture for general intelligence. *Artificial Intelligence* **33,** 1 (1987), 1–64.
16. Li, K. IVY: A shared virtual memory system for parallel computing. *Proc. International Conference on Parallel Processing.* 1988, pp. 94–101.
17. McKeown, D. M., Harvey, W. A., and Wixson, L. Automating knowledge acquisition for aerial image interpretation. *Comput. Vision Graphics Image Process.* **46,** 1 (Apr. 1989), 37–81.
18. McKeown, D. M., Harvey, W. A., and McDermott, J. Rule based interpretation of aerial imagery. *IEEE Trans. Pattern Anal. Mach. Intelligence* **7,** 5 (1985), 570–585.
19. McKeown, D., McVay, W., and Lucas, B. Stereo verification in aerial image analysis. *Opt. Engrg.* **25,** 3 (1986), 333–346.
20. Miranker, D. P. Treat: A new and efficient match algorithm for AI production systems. Ph.D. thesis, Computer Science Department, Columbia University, 1987.
21. Miranker, D. P., Kuo, C. M., and Browne, J. C. Parallelizing compilation of rule-based programs. *Proc. International Conference on Parallel Processing.* 1990, pp. II-247–II-251.
22. Mohan, J. Performance of parallel programs: Model and analyses. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, July 1984.
23. Newell, A. *Unified Theories of Cognition.* Harvard Univ. Press, Cambridge, MA, 1990.
24. Oflazer, K. Partitioning in parallel processing of production systems. Ph.D. thesis, Computer Science Department, Carnegie Mellon University, Mar. 1987.
25. Oshisanwo, A. O., and Dasiewicz, P. P. A parallel model and architecture for production systems. *Proc. Internation Conference On Parallel Processing.* 1987, pp. 147–153.
26. Reddy, R., and Newell, A. Multiplicative speedup of systems. In Jones, A. (Ed.). *Perspectives on Computer Science.* Academic Press, New York, 1977, pp. 183–198.
27. Schreiner, F., and Zimmerman, G. Pesa-1: A parallel architecture for production systems. *Proc. International Conference on Parallel Processing.* Aug. 1987, pp. 166–169.
28. Tambe, M., Kalp, D., Gupta, A., Forgy, C. L., Milnes, B. G., and Newell,

A. Soar/PSM-E: Investigating match parallelism in a learning production system. *Proc. ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems.* July 1988, pp. 146–160.

WILSON HARVEY is a senior research programmer in the School of Computer Science at Carnegie Mellon University. He received a B.S. in physics and mathematics from Carnegie Mellon University in 1986. Mr. Harvey has been with the School of Computer Science for more than six years, working primarily on the development of knowledge-based computer vision systems and the representation of spatial knowledge for scene analysis. His research interests include knowledge representation and image understanding.

DIRK KALP is a senior research programmer in the School of Computer Science at Carnegie Mellon University. He graduated from Carnegie Mellon University in 1973 with a B.S. degree in mathematics and from the University of Pittsburgh in 1981 with an M.S. degree in computer science. On the research staff at Carnegie Mellon since 1985, he has worked primarily in the area of parallel production systems. Prior to that he worked in the Operating Systems Group at Perq Systems Corporation in Pittsburgh on the development of one of the first high-performance graphics workstations. In addition to parallel architectures for production systems, his research interests include multiprocessor operating systems, virtual memory systems, and the development of tools for fitting symbolic parameter cognitive models.

MILIND TAMBE is a research associate in the School of Computer Science at Carnegie Mellon University. He graduated from Birla Institute of Technology and Science, Pilani, India, in 1986 with an M.Sc. (Tech.) in computer science. He received his Ph.D. in May, 1991, from the School of Computer Science at Carnegie Mellon University. His interests are in the areas of integrated AI architectures and efficiency of AI programs. He is a member of AAAI.

Received February 1, 1991; revised May 15, 1991; accepted June 25, 1991

DAVID MCKEOWN is a senior research computer scientist in the School of Computer Science at Carnegie Mellon University and has been a member of the research faculty since 1986. He received a B.S. degree in physics and an M.S. degree in computer science from Union College, Schenectady, New York. Prior to joining the faculty he was a researcher at Carnegie Mellon from 1975, a research associate at George Washington University and a member of the technical staff at NASA Goddard Space Flight Center, Greenbelt, Maryland (1974–1975), and an instructor in computer science and electrical engineering at Union College (1972–1974). His research interests in computer science are in the areas of image understanding for remote sensing and cartography, digital mapping and image/map database systems, computer graphics, and artificial intelligence. He is the author of over 35 papers and technical reports and is an active consultant for government and industry in these areas. He is a member of ACM, IEEE, AAAI, American Society for Photogrammetry and Remote Sensing, and Sigma Xi.

ALLEN NEWELL is the U. A. and Helen Whitaker University Professor of computer science at Carnegie Mellon University, where he has been since 1961. He was educated in physics (B.S., Stanford University), mathematics (Princeton University), and industrial administration (Ph.D., 1957, Carnegie Institute of Technology). He contributed to the emergence of cognitive psychology and artificial intelligence in the mid-1950s, and has continued to make contributions to them ever since, many with Herbert Simon. His current research is on Soar, a general cognitive architecture that embodies a unified theory of human cognition (with John Laird and Paul Rosenbloom). He has also made contributions to list processing, expert systems, computer structures, and human–computer interaction. He is a member of the National Academy of Science and the National Academy of Engineering, and the recipient of the Association of Computing Machinery's A. M. Turing Award (with Herbert Simon), and the American Psychological Association's Distinguished Scientific Contribution Award. He is the author or coauthor of over 250 papers and 10 books.