Post-Game Analysis-An Initial Experiment for Heuristic-based Resource Management in Concurrent Systems

by

Jerry C. Yan

Technical Report CSL-TR-87-314
February 1987

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 943054055

Abstract

In concurrent systems, a major responsibility of the resource management system is to decide how the application program is to be mapped onto the multi-processor. Instead of using abstract program and machine models, a generate-and-test framework known as "post-game analysis" that based on data gathered during program execution is proposed. Each iteration consists of (i) (a simulation of) an execution of the program; (ii) analysis of the data gathered; and (iii) the proposal of a new mapping that would have a smaller execution time. These heuristics are applied to predict execution time changes in response to small perturbations applied to the current mapping. An initial experiment was carried out using simple strategies on "pipeline-like" applications. The results obtained from four simple strategies demonstrated that for this kind of application, even simple strategies can produce acceptable speed-up with a small number of iterations.

Key Words and Phrases: Resource Management, Parallel Processing, Performance Evaluation, Workload Models, Simulation, Distributed Systems, Operating System

Table of Contents

1. Introduction

Many factors contribute to the effective utilization of a multiprocessor. These factors include problem formulation techniques, programming language/paradigm, implementation methodology, resource management strategies, and hardware characteristics. Needless to say, many schools of thought exist - each with its pros and cons, depending on the kind of application and hardware it is applied to. The research reported here does not involve yet another new programming paradigm nor any new processor structure. The problem of finding *the best way to map a given application to a given multiprocessor* is addressed instead. This problem must be solved properly so that computer architectures exhibiting massive parallelism can be effectively used. It is formulated as a *mapping problem* defined as follows:

GIVEN A PARALLEL APPLICATION AND A MULTIPROCESSOR, MAP[1] THE **APPLICATIONOVERTHEMACHINESUCH** THAT **EXECUTIONTIMEISMINIMIZED.**

Two assumptions are made:

- The application is already expressed in a programming paradigm where "natural" partition boundaries exists. Examples include communicating sequential processes [Brookes 83], Actors [Agha 85], remote procedures [Nelson 81] and data flow paradigms.
- The multiprocessor consists of a set of homogeneous processing elements (or sites) connected in some topology.

Unfortunately, this problem is NP-complete even for *well-structured* programs (see [Mayr 81]). A closer look at real programs reveals further complications. The mapping problem can be divided into 2 categories according to the "nature" of the application:

1. All actors[2] are declared at compile-time.
2. Actors may be dynamically created or destroyed.

Each of these categories may be further divided into two sub-categories

a The total number of actors, their precedence relationships and communication requirements do not depend on the characteristics of the data.

b Program behavior is data-dependent.

This paper reports the results when simple heuristic-based strategies are applied parallel application programs that falls into class 1a — i.e. program behavior is relatively steady over a wide range of input data and all actors are declared at compile-time.

---

[1] i.e. to assign placement locations for various parts of the application

[2] Actors are chosen to be the generic type of computing agents to represent processes, modules etc.

The second section presents an overview of the Actor computation paradigm in which the concurrent application (to be mounted on the multiprocessor) is formulated. The heuristic approach is explained in detail followed by a description of the four simple strategies. The experimentation environment (known as "Axe") is then outlined. The experiments are described with their results and interpretations. The paper concludes with a summary and a projection for future research.

2. The Actor Programming Paradigm

Realistic evaluation of mapping strategies depends on the study of real applications. In the current context however, the development of complete language and compiler tools, together with run-time environments is impractical for short term experimental use in research. Simulation at the "process level" was therefore chosen as a "middle-ground" between lengthy instruction-by-instruction emulation and general stochastic modeling. This simulation is based on "program models" [Yan 86a]. The program is represented by a *behavior description language* (or *BDL*) designed to model a subset of the Actor paradigm [Atkinson 77]:

1. A parallel computation is expressed as a collection of autonomous modules called *actors.*
2. Actors interact with one another only via message passing - to facilitate data transport and the compliance of precedence relationships.
3. When an actor receives a message, it may
   - perform user-programmed computations involving its internal state variables and the values transmitted in the message;
   - send or wait for specific messages to/from other actors; or
   - create other new actors.

This BDL model preserves the message pattern between actors as well as the relative processing and storage requirements of each actor. An example-of a computation expressed in BDL will be given at a later section.

3. Applying Heuristics to Resource Management

An "optimal" solution to the mapping problem exhibits the shortest execution time. The optimal solution cannot be guaranteed because:

1. The number of possible solutions is infinite. Different programs have different resource utilization requirements. The behavior of a single program may be a complex function of the input data sets and the stage of the computation.

2. Even if the search space were finite (say, for one particular program) exhaustive search is often the only means to verify the optimality of the solution because the dependencies between actors is non-trivial. Exhaustive search is also not feasible since the number of possible solutions increases exponentially.
3. In fact, attaining the optimal solution does not only require complete knowledge of the system at all time, program behavior in the future has to be correctly predicted as well!

Since an exact solution to the mapping problem cannot be guaranteed in general for any program on any machine, a heuristic approach is suggested. The "mapping strategy" — expressed as a set of heuristics - attempts to find an approximate solution to the mapping problem for a given program on a given machine. A complete mapping strategy consists of two parts - *placement* and *migration* heuristics. Placement heuristics suggest a site for placing an actor when it is created whereas migration heuristics suggest when and where to move an actor (to another processing site) after it has been assigned to a site for a certain time. The proposed strategy must be *adaptive* - incorporating run-time observations (past and present) when deciding where to place or move actors. This ability to respond to program behavior variations which in turn, are dependent on changes in input data characteristics is critical to the success of the strategy.

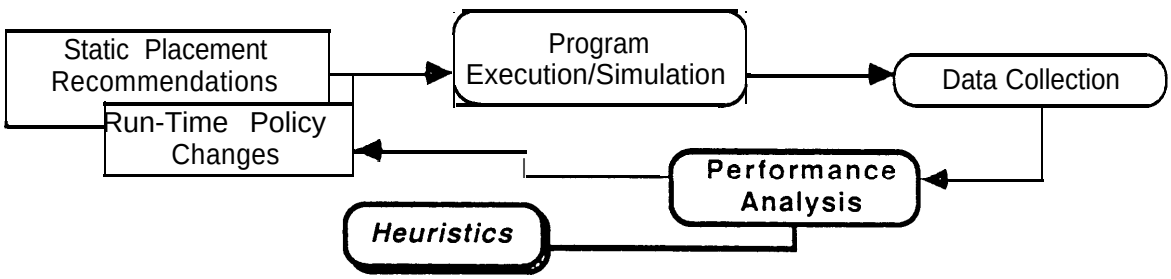

Figure 1. Heuristic-based Post-Game Analysis —
The application of heuristicsto incrementally improve the mapping of actors to sites

Heuristics may be put to use in three phases: *pre-game, mid-game* and *post-game:*

1. *Pre-game application* - Before the program is loaded and executed, the program text can be analyzed (with or without directives from the programmer). Heuristics can help decide where to place actors that have already been declared.
2. *Mid-game adjustments* - After execution commences, the decisions made at the first phase may have to be revised (i.e. migration may have to take place) in response to unexpected changes in input data characteristics. Placement decisions also have to be made for actors created at run-time.

3. *Post-game evaluation* - After execution terminates, the data collected during the run may be analyzed to:

- revise the decisions made in phase 1 and
- improve the decisions made in phase 2.

when the program is executed again!

The initial experiment reported here applied heuristics in the "post-game" fashion. In order to evaluate the feasibility of this approach and compare relative merits among individual heuristics, each of the four strategies proposed in the next section consist of a single heuristic.

4. Four Simple Mapping Strategies

Four simple mapping strategies are proposed to be candidates for "post-game analysis". The relative merits of these strategies were evaluated by simulations. They suggest modifications for a given mapping of actors to site that may (hopefully) result in a reduction of execution time. Data gathered during simulation is used for this analysis. All strategies contains only one heuristic in the form of a "rule":

"IF <SOME-CONDITION> **THEN** <MOVE AN ACTOR TO SOME PLACE>".

When the predicate is satisfied, the consequence proposes is carried out. The four strategies tried are listed as follows:

1. IF **(and** (*most_crowded* <SITE-A>) (*most_unhappy* <ACTOR_X> <SITE-A>)
   (most-vacant <SITE-B>))
   THEN (move-obj <ACTOR_X> <SITE_B>)[3]
2. IF **(and** (*most_crowded* <SITE_A>) (*most_happy* <ACTOR-X> <SITE-A>)
   (most-vacant <SITE-B>))
   THEN (*move_obj* <ACTOR-X> <SITE_B>)[4]
3. IF (**and** (*most_crowded* <SITE-A>) (*most_unhappy* <ACTOR-X> <SITE-A>)
   (*highest_reward_if_moved* <ACTOR-X> <SITE_B>)
   THEN (move-obj <ACTOR-X> <SITE_B>)[5]

---

[3] It reads "If <SITE-A> is most crowded and <OBJECT-X> is most unhappy in <SITE-A>, then move <OBJECT X> to <SITE B> which is most vacant."

[4] The rationale for moving an actor away from the site in which it is already "happy" with will be explained later.

[5] It reads "If <SITE A> is most crowded and <OBJECT X> is most unhappy in < SITE-A>, moving <OBJECT-X> to <SITE-A> will ***probably*** reduce the execution time a lot.".

4. IF (and (*most_crowded* <SITE-A>) (most-happy <ACTOR-X> <SITE_A>)
(*highest_reward_if-moved* <ACTOR-X> <SITE_B>)
THEN (*move_obj* <ACTOR_X> <SITE_B>)

The predicates that describes sites and actors are detailed as follows:

- The "happiness" of an actor describes whether the actor's request for resource utilization is being met without contention with other actors that reside in the same site. It is measured by a weighted sum of two terms:
  i) the response ratio: t_w/t_u ; and
  t_w = time spent waiting for cpu
  t_u = actual cpu time used
  ii) the remote-local ratio (t_{rc}/t_{lc}).
  t_{lc} = time spent in communicating with actors reside in the same site
  t_{rc} = time spent communicating with actors reside in different sites
- The "crowdedness" of a site describes how the resources are being utilized at a particular site. The "most vacant" site is the "least crowded one". It is measured by a weighted sum of these terms:
  i) the response time of the CPU;
  ii) the average length of its ready queue (for execution);
  iii) the total no. of actors resident; and
  iv) % idle time of the CPU.
- The "reward" (e.g. of moving <ACTOR_X> from <SITE_A> to <SITE_B>) measures the "potential reduction in cpu usage" when a certain action takes place:
  reward = $\Delta cpu\_load_{site\ a} - \Delta cpu\_load_{site\ b}$
  $\Delta cpu\_load = At_{,,} + \Delta t_{cpu} + \Delta t_{lc}$
  Δt_{rc} = change in time spent in remote communication
  Δt_{ic} = change in time spent in local communication
  Δt_{cpu} = change in time spent in using the cpu for computation

5. "Axe" - The Experimentation Environment

The "Axe" simulation environment [Yan 86b] was specifically designed to facilitate such investigations using discrete-time simulation. The current version of "Axe" consists of a set of software tools that allows 'the following tasks to be performed in an integrated environment:

1. *Computation model specification using BDL (behavior description language)* - A compiler/translator converts the application program models from BDL into forms understood by other modules of the simulation environment;

2. *Execution environment specification* - This includes various parameters that describe the multiprocessor and operating system algorithms;
3. *Simulation* - a simulator projects the execution time of the program model on multiprocessor structures with various machine parameters and mapping strategies;
4. *Instrumentation* - a built-in monitor system gathers run-time statistics during simulation to enable evaluation of mapping strategies, as well as software and hardware architecture; and
5. *Experimentation* - Although specifically built for investigations in mapping strategies, a flexible user interface enables the researcher to study issues in parallel processing such as: problem formulation, hardware architectural issues, matching machines and programs, and operating system level algorithms. He/she may inspect (and over-ride) decisions recommended by mapping heuristics sets during simulation.

Currently, "Axe" models a class of multiprocessors known as *ensemble architectures* [Lutz 84, Seitz 82]. This class consists of a collection of homogeneous processing elements - each of which is connected to its nearest neighbors in a regular fashion (e.g. the Cosmic Cube [Su 85]). Each site is *autonomous* - it contains its own storage, processor and a distributed operating system kernel governing local activities such as message forwarding, task scheduling, and memory management. A collection of pre-defined abstract machines (or sites) is provided in "Axe". They may be tailored further parametrically by the user:

1. Values of hardware parameters which can be specified include:
   - message sending/receiving overhead;
   - relative speeds of communication links to the processors; and
   - number of processors and memory size at each site
2. A number of *built-in* topologies and routing algorithms can be selected by modifying a single variable or by writing one simple function in C[6];
3. A number of *built-in* mapping and scheduling algorithms[7] are also offered.

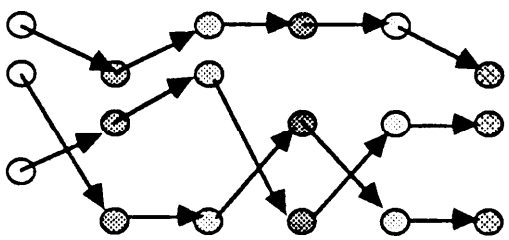

Figure 2. A "Pipe-line" Computation

[6] "Axe" is built on top of CSIM - a simulation utility written in C [Schwetman 86].

[7] A partition strategy is concerned with placing actors into sites where as a scheduling strategy determines how the cpu at a site is to be shared among the actors residing in the same site.

6. Pipelined Applications

The idea of obtaining speed-up through pipelining has been exploited in hardware for a long time. "Software pipelining" was recently proposed again as a method of managing concurrent software. A pipeline application consists of a number of actors organized into stages. The number of actors at each stage need not be identical. Figure 2 illustrates the general scheme of the pipeline operation. Depending on how saturated the pipeline is, the maximum speed-up obtained in such an arrangement is (w x s) where s = number of stages and w = number of actors in each stage. It should be noted that such an optimal speed-up could only be obtained when:

i) the pipeline is always full;

ii) each stage has equal service time;

iii) actors do not have to compete for cpu at any time

The BDL specification of a typical actor in this stage is illustrated below in Figure 3. Basically, when a message is received by an actor, certain amount of processing is initiated. After some time, a result is obtained and is passed to an actor in the next stage. Different data sets were used to create different message patterns and cpu demands for each actor.

```
(DefActor Pipe-stage
   (init compute)                          ; messages it understands
   (nxt_1 nxt_2 nxt_3 nxt_4 tmp)           ; pointers to other actors
   (value)                                 ; internal states
   (init                                   ; behavior 1 – "initialization"
      (record nxt_1 nxt_2 nxt_3 nxt_4      ; record pointers to next stage &
        basic-duration))                   ; . ..default residence time
   (compute                                ; behavior 2 – "computation"
      (record value)                       ,"value" from previous stage
      (run (some-time value))              ; residence time is data dependent
      (setq tmp (target-select value))     ; target selection is data dependent
        (post tmp compute value)))         ; pass data to next stage
```

Figure 3. BDL Description of a pipe-stage

7. The Experiment - Results and Interpretations

The simulations were carried out with the following parameters:

Machine model:

- number of sites: 4, 9, 16, 25
- connection topology: two-dimensional square-grid with "no wrap around"

- t_{hop}[8]: 4, 16, 28, 40, 52, 64, 88 msec.
- number of cpu/site: 1

Operating system:

- scheduling policy:
  - operating system preempts any "user-processes"[9] when delivering/routing messages
  - "user-processes" - are served until completion on a first-come-first-serve basis
- overhead:
  - context-switch: 10 msec.
  - message receiving: I context-switch
  - message delivery: remote = t_{hop}; local = 0
  - actor creation: 1 context-switch
  - "user-process" creation: 1 context-switch

Application:

- number of actors: 48 actors arranged in 6 stages (each stage is 8 wide)
- number of requests through the pipeline: ranges from 20 to 200 on various simulations
- message pattern: four patterns were simulated by re-programming the functions *'Select target"* & *"next-value"* and modifying the input-data:
  - *Random uniform:* whenever a message is received, there is an equal probability for each actor in the following stage to be sent the next value.
  - *Deterministic uniform:* the message pattern is pre-determined so that each actor will receive an equal number of messages.
  - *Random biased:* certain actors have higher probabilities of receiving messages
  - *Deterministic biased:* "preferred paths" are programmed so that certain actors receive more messages than others.
- cpu-demand: the average time required by each actor to serve a message is 100 ms. Various ranges were used in the simulations

Absolute maximum speed up: 14.5[10]

Initial Placement: All actors were placed on one site at the beginning.

Although many simulations have been carried out, the results shown here only comes from one particular data-set. All simulations produced similar results.

---

8 time spent routing each packet across a communication link connecting two sites

9 Loosely speaking, every time an actor computes, it becomes an active "user process".

10 The "absolute maximum speed up" is obtained by simulating the application on an "ideal machine" with an infinite number of sites and *"instantaneous"* communication between any two sites.

7.1 Progress Charts — General Features

The performance of various heuristics is represented in the form of a *"progress chart"*. Figure 4. illustrates the improvement made to the solution of the mapping problem as the heuristics are applied one at a time:

- Y-axis: "speed up" is used as a principle criteria to evaluate the mapping suggested by mapping heuristics. It is defined as the ratio: (t_1/t_{actual}) where
  t_1 = execution time when only one site is used
  t_{actual} = actual time used when all sites are used.
- X-axis: Each "iteration" involves the application of a heuristic - resulting in the migration of one actor to an alternate site.

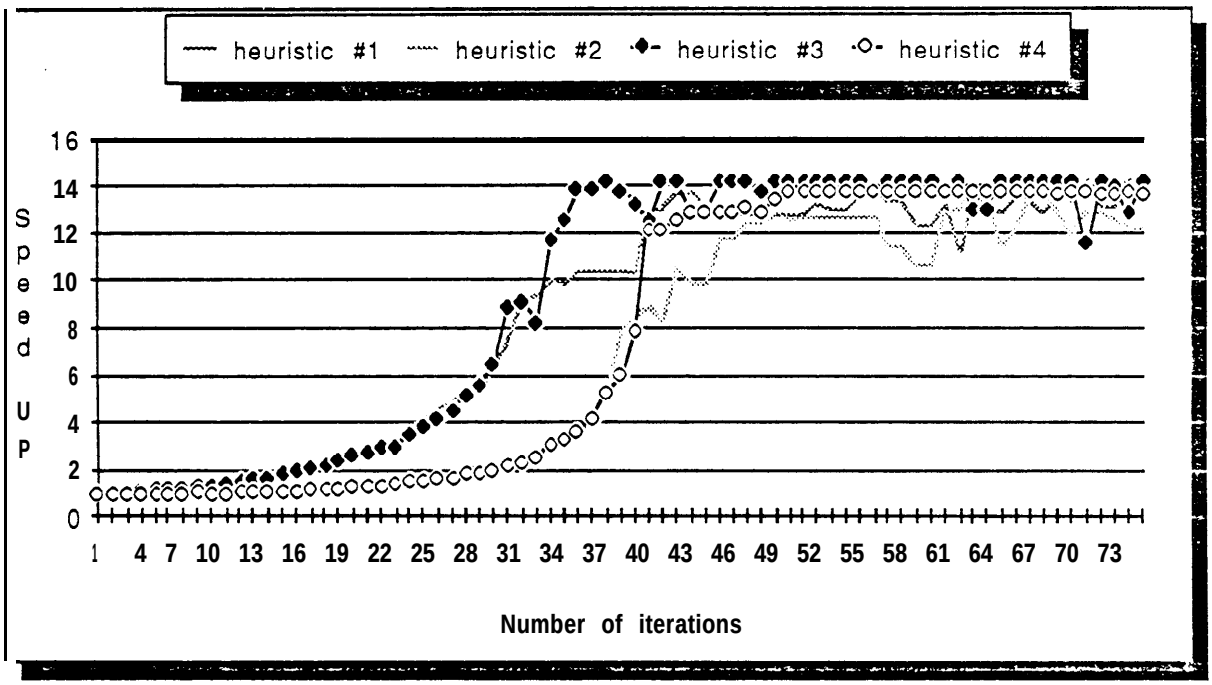

Figure 4. "Progress Chart" for 25 Sites and low communication costs

Each progress chart may be divided into three phases:

- firstly, no significant performance improvement is usually observed
- *ascending phase:* then, a dramatic improvement is observed which usually attains the "maximum achievable speed-up" for the heuristic.
- *final phase:* In stead of further improvement, the progress charts merely "oscillate" around the "maximum achievable speed-up"

It should also be noted that the curve obtained is non-monotonic.

7.2 Progress Charts — Interpretation

1. *The length of the initial phase* - No substantial improvements are made initially since more than half of the actors are still crowded on one site!
2. *A rapid ascending phase:* Since over half of the actors are (more or less equally) distributed over (n_{site} - 1) sites, each additional evacuation of actors from the "start up" site alleviates the one remaining bottle-neck of the system *in relatively larger steps* - thus resulting in a dramatic improvement
3. *After a (more or less) optimal point is found, not much improvement is observed* - All the heuristics are designed to alleviate a relatively crowded site. At the end of the ascending phase, all the sites are more or less equally loaded, thus rendering these heuristics useless.
4. Heuristics #2 and #4 have a longer initial phase than #1 and #3 respectively - These two heuristic sets works very differently. Heuristics #2 & #4 first distributes the actors with less demands around and then alleviate the "start up" site until it does not stand out as a "very" crowded site. Heuristics #1 & #3 greedily alleviates the "start up" site right from the beginning - thus resulting in a shorter initial phase.
5. *The curves are not monotonic* - This does not necessary mean that the heuristic had made an error. The alleviation of one bottle-neck may result in the creation of a bigger one elsewhere in the system. This decrease in performance is always compensated for in the next move. In the initial stage, the compensation usually results in a solution with an even shorter execution time!

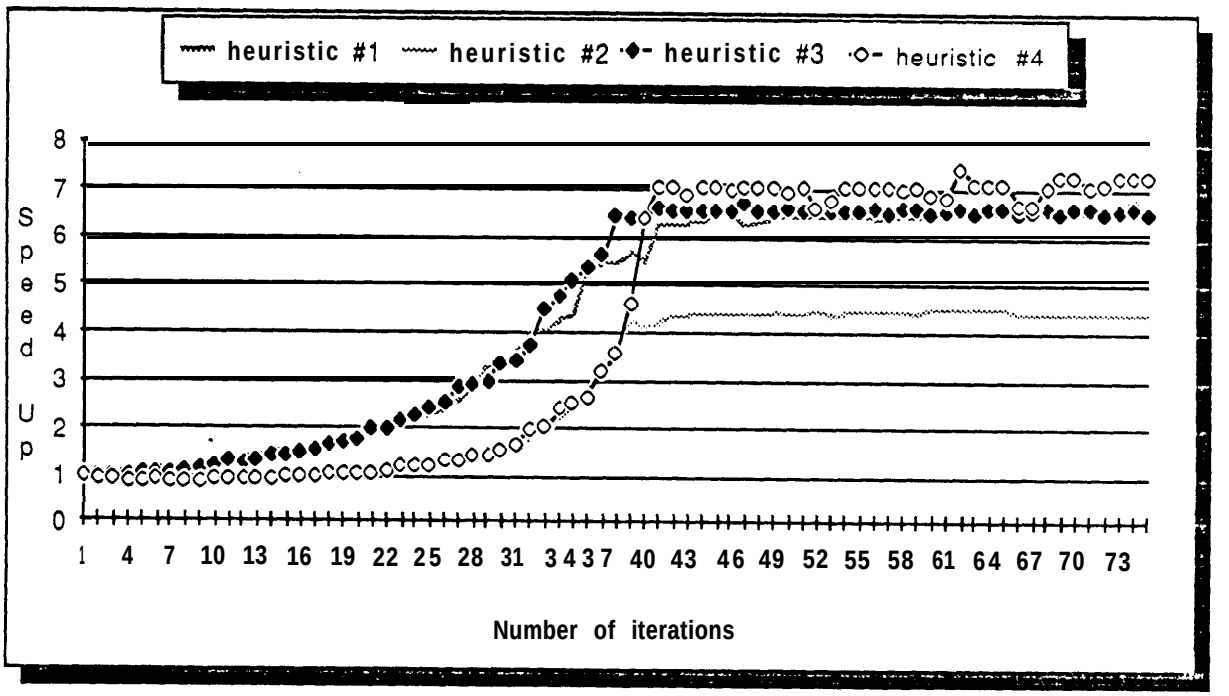

Figure 5. "Progress Chart" for 25 Sites and high communication costs

7.3 Varying Communication Cost and Number of Sites

Figure 4 illustrates a typical situation with lots of sites and low communication cost (4 msec)— all heuristics reached within 95% of the absolute achievable maximum speed-up. Heuristic #3 attained the highest speed-up with the fewest number of iterations, closely followed by heuristic #2, #1 and #4. Figure 5 shows the same application mounted on an identical multiprocessor with very high communication cost (88 msec) - all heuristics reached within 95% of one another with the exception of heuristic #2. Other experiments also show similar results.

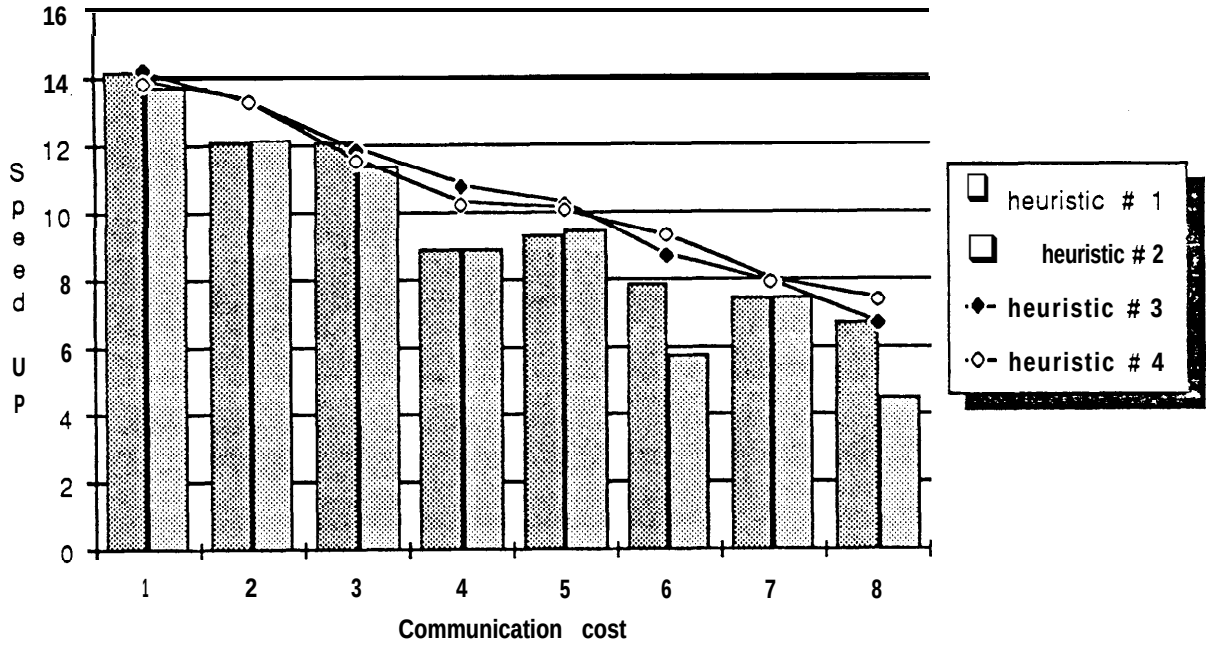

Figure 6. "Maximum Achievable Speed-Up" for 25 sites over a range of communication costs [11]

Figure 6 and 7 further illustrates that:

1. Heuristics #3 and #4 performs consistently better than #1 and #2 over a range of number of sites and communication costs. This is probably due to the fact that "gain projection" is predicts fairly correctly what happens when an actor to move to a new site. Heuristic #1 and #2 does not consider increased cpu demand at the new site at all.
2. Heuristic #3 performs consistently better than #4 over a range of sites.

These experiments demonstrated two important findings:

[11] The X-axis labels corresponds to a linear increase in communication cost:

1 = 4 msec, 2 = 16 msec . . . n = ((n-1)*12 + 4) msec etc.

1. An heuristic based on the concept of "alleviation of the worst bottle-neck in the system" works.
2. "Gain prediction" proved to be successful - but only marginally for a simple computation like the one illustrated here.

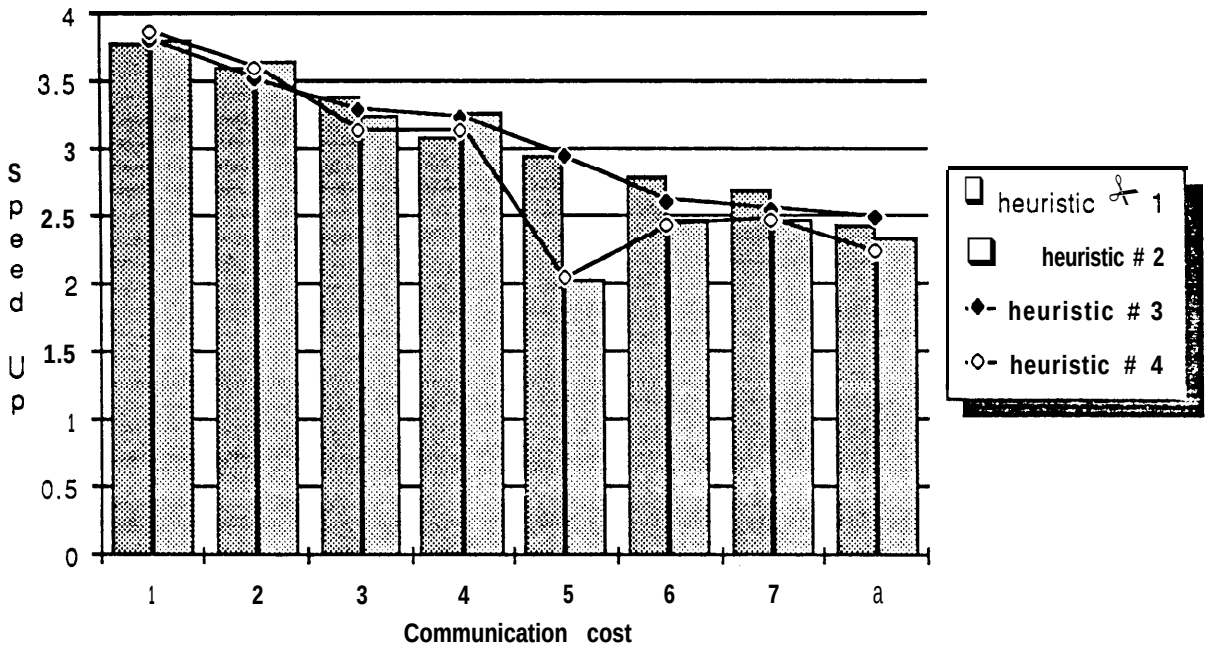

Figure 7. "Maximum Achievable Speed-Up" for 4 Sites over a range of communication costs

8. Summary and Further Research

The use of heuristics is proposed as a means for resource management with multiprocessors. Initial results when comparing four simple heuristics have been reported. Models of applications formulated as a pipeline were simulated in the "Axe" simulation environment. All four strategies attempts to alleviate the relative crowded site(s). Two of the heuristics (#3 & #4) incorporate "gain prediction" when making the mapping decision (by guessing the increased cpu load this move will cause in the *target* site). These experiments have demonstrated that:

1. By gathering run-time statistics, program behavior can be exploited to aid resource management.
2. Simple heuristics based on "alleviation of bottle-necks" works.
3. Gain prediction provides a marginal improvement.
4. The "Axe" experimentation environment was easy to use. Automatic data collection and analysis enabled easy implementation of heurisitics. All experiments were carried out with minimal re-compilation of the system. The turn-around time is small because:

. simulation at the process-level is relatively fast

- application specification in BDL is simple and automatically integrated into the rest of the environment
- machine parameters are easily modified between simulations

Further research involves:

1. experimenting with real applications
2. modeling different processor structures
3. incorporate processor structure into the heuristics
4. combining these heuristics to form a complete heuristic set -
   - arrive at the maximum achievable speed-up faster,
   - with less oscillation, and
   - hopefully result in an even shorter execution time.
5. These heuristics will be use as the basis for developing the heuristics to be used in the "pre-game" and "mid-game" activities.

References

[Agha 85] Gul A. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 844.

[Anderson 77] R Atkinson and C Hewitt, "Specification and Proof Techniques for Serializers", Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 438.

[Brookes 83] S D Brookes, C A R Hoare, A W Roscoe, "A Theory of Communicating Processes", Carnegie-Mellon University Department of Computer Science, CMU-CS-83-153.

[Lutz 84] Christopher Lutz, "Design of the Mosaic Processor", California Institute of Technology, Technical Report 5203:TR:85.

[Mayr 81] Ernst Mayr, "Well Structured Parallel Programs are not Easier to Schedule", Stanford University, Computer System Lab Technical Report STAN-CS-8 l-880.

[Nelson 81] B J Nelson, "Remote Procedure Call", Xerox Corp. Palo Alto Research Center, CSL-81-153.

[Schwetman 86] Herb Schwetman, "CSIM: A C-Based, Process-Oriented Simulation Language", Microelectronics and Computer Technology Corporation, Proceedings for Winter Simulation Conference '86, Washing ton DC.

[Seitz 82] Charles L Seitz, "Ensemble Architectures for VLSI - A Survey and Taxonomy", 1982 Conference on Advanced Research in VLSI, Massachusetts Institute of Technology.

[Su 85] W-K Su *et al*, "C Programmers' Guide to the Cosmic Cube", California Institute of Technology, M.S. Thesis (CITCS) 5129:TR:84.

[Yan 86a] J.C. Yan, "Parallel Program Behavior Specification and Abstraction using BDL", Computer System Laboratory, Stanford University, CSL-TR-86-298

[Yan 86b] J.C. Yan and SF. Lundstrom, "AXE: A simulation environment for actor-like computations on ensemble architectures", Proc. 1986 Winter Simulation Conference, Washington DC.