Managing and Measuring Two Parallel Programs on a Multiprocessor

Jerry C. Yan

Technical Report CSL-TR-87-333

June 1987

This research was supported by NASA under contracts NAG 2-248 and NCA2-109, using facilities provided under contract NAGW 419.

Table of Contents

1. Introduction

Research is being conducted to determine how distributed computations can be mapped onto multiprocessors so as to minimize execution time. The class of programs being considered here, falls under a subset of the ***Actor paradigm*** [Agha 85]. A computation is expressed as a collection of autonomous executable modules known as *players [Yan* 86a]. They interact with one another via message passing. When a player receives a message, it may perform user-programmed computations; send/wait for specific messages; or create new players. The multiprocessors being considered consist of homogeneous processing elements (or sites) connected via physical communication links. The resource management problem in this context involves mapping a single concurrent computation to the multiprocessor — as opposed to some other efforts that involve minimizing the average turn-around time (or some other parameter) for a collection of unrelated tasks. By

(i) restricting ourselves to programs in which the number and type of players created are independent of input data and system load and

(ii) forcing each player to reside in the site in which it was created for the whole duration of the computation,

the mapping problem reduces to a ***placement problem:*** "In which site should a player be placed after it is created?". The number of all possible solutions is finite, enumerable, but extremely large.

Instead of formulating it as yet another optimization problem based on some abstract program/ machine models, the approach being investigated here (called "post-game analysis") is based on placement heuristics which utilize program execution history [Yan 87b]. Figure l-l below gives a summary of how the "post-game " iteration framework works:

1. The program is either executed or simulated
2. Data gathered during simulation/execution is analyzed by a set of heu- risics which assess relative merits to alternative perturbations to the current mapping.
3. The perturbation given the highest merit is chosen — from which an al- ternative mapping is generated
4. Go to step 1 unless some terminating condition is met.

Figure l-l. **The "Post-game Analysis" Iteration Framework**

Although initial experiments [Yan 87a] have demonstrated that "post-game analysis" indeed discovered mappings that exhibit significantly shorter execution times than the worst cases for the programs tested, three important issues remain to be addressed:

i) *Performance Evaluation* — Just like many other heuristic approaches proposed to solve combinatorially explosive problems, the initial experiments failed to demonstrate how close the proposed mappings actually were to the "optimal". The "optimal mapping", in this context, is defined as the mapping in which the given program executes fastest on the given machine. Furthermore, unless the "distribution" of execution time (over all mappings) is known, it is difficult to assess whether the heuristics actually succeeded or it was "mere luck".

ii) ***Heuristics Development*** — "Post-game" analyzes data gathered during program execution to discover bottlenecks in the system. It then make use of heuristics that "reason" or "make guesses" to generate mappings which exhibit shorter execution time than others. Although the parameters on which these heuristics were based seemed intuitively reasonable, it remains to be shown that they actually describe aspects of the system which directly translate/relate to execution time. A good example involves the amount of "remote messages" — messages that has to be delivered over the communication network connecting different sites of the multiprocessor. It will be shown later that in spite of popular belief, minimizing this quantity does not necessary results lead to small execution times even when the communication links are slow!

iii) ***Control Strategy Development*** — The "control strategy" being used in the "basic" post-game analysis [Yan 87a] involves selecting the best and Smallest possible perturbation. After each iteration, only one of the players is moved to a new location. Although the number of iterations needed to achieve convergence is still small in comparison with the total number of possible assignments, efforts are being made to reduce the number of iterations by relaxing the control strategy to allow more than one player to be assigned to an alternative site. In order to make this selection "intelligently", one must understand very well how and why and to what limit do the heuristics reported earlier works.

In order to determine (i) which measurable parameters are actually related to execution time; (ii) the inter-relationship between these parameters; and (iii) the distribution of execution time over all possible placements and a range of communication costs, parallel program execution was simulated using "Axe" [Yan 86b]. "Axe" provides an integrated environment for computation model description, processor architecture specification,

discrete-time simulation, automated data collection as well as the application of “post-game” heuristics. Communication cost as well as program parameters were varied.

Section 2 gives an overall description of how the experiments were designed and the parameters measured. Each experiment involves the simulation of all possible placements for a program with nine players executed on a multiprocessor with four sites completely connected (over 11,000 possible placement configurations). Five groups of parameters are measured representing different aspects in the concurrent execution environment: (i) overall measurements, (ii) communication parameters, (iii) **CPU** utilization, (iv) **CPU** contention and (v) dependencies between players. Two programs were simulated — representing two major classes of parallel computations. The players in the first benchmark can be visualized as “service centers” in a “pipe-line”. Messages carrying different requests “flow” through each stage of the pipeline — making different processing demands at the site in which the player reside. Section 3 reports the preliminary findings for this benchmark. It was found that the parameters that correlated best with execution time describes the **CPU** contention at the busiest site. The results from the second benchmark is described in section 4. This program is representative of the structure of many "divide-and-conquer” algorithms. Players are organized as “nodes” at different levels of a “tree”. Requests generated at the root propagate down to the “branches” of the tree. Players at higher levels have to “block” and wait for replies from those at the lower levels. It was discovered that the “dependency parameters” correlates much better than all the others. The findings for both programs are summarized in section 5. The paper concludes by evaluating the performance of the application of “post-game” analysis to these programs. It is shown that “post-game” analysis achieved close to 96% optimal speed-up for both programs in most cases.

2. Designing the Experiments

2.1 Benchmark Selection / Program Characterization

It would be desirable to have a set of programs with "different characteristics" to serve as "benchmarks" for experiments of this sort. Unlike sequential programming however, benchmarks for parallel/distributed computing are relatively difficult to come by:

i) **There are not many parallel applications.** Parallel/distributed processors have been in existence for quite a long time — most of which lived as "research secrets" in laboratories across the nation. Only until recently are some commercially available. It is not easy to find many programs that run on these machines — not to mention any that could be termed "representative benchmark" for parallel program behavior.

ii) **It is difficult to classify/describe parallel programs.** Sequential computing on von-Neuman architectures can be described in terms of two basic operations: "data/ instruction access" and "processing". Classification of sequential programs are, therefore, based on the characteristics of such operations (e.g. "access locality", **"CPU/IO** bound" and "branch probabilities"). Although player programs can also be described in terms of the characteristics of three basic observable actions — "player creation", "message sending/ waiting", and "processing", there are many dimensions along which programs can be classified (e.g. degree of parallelism, amount and pattern of communication and computation, dependency/relationships between players and parallelism "grain size" etc.). Different programming paradigms seems to demand different dimensions for classification, The large variety of paradigms in which parallelism can be expressed (or exploited) makes finding "representative" programs difficult.

For the purpose of this experiment, programs are classified into two classes depending on how concurrency is expressed/exploited:

Class A: ***(e.g. Data-flow [Davis 82] and Systolic [Kung 82]***) — With these paradigms, the computation is organized as a collection of computing agents (players) among which requests (messages) of different types are communicated. No player has to be blocked to wait for a specific (reply) message from another player before resuming computation. In other words, a player can always choose to process messages (in its message buffer) in any order.

Consider the simple problem of "concurrent tree search": A data-base is organized as a tree of players — the lowest level (1) of which holds the actual data.

Search can be carried out simultaneously along different branches of the tree. A "class-A" implementation is described in Figure 2-l using BDL — a behavioral description language for player programs [Yan 86a]. Requests are generated at the "root" of the tree and propagates down each branch of the tree. There is no need to block and wait for replies before sending out the next request because all messages (both enquiries and replies) are tagged. A player at stage (k) can accept requests arrived from stage (k+l) and replies from stage (k- 1) and process them in any order.

```
(defPlayer node
 (SEARCH RESULT)                          messages it understand
 (parent left-child right-child)          its acquaintances
 (search-key data)                        internal states
 (SEARCH                                  "SEARCH" message received
   (record search-key)                    extract "search-key"
   (post left-child SEARCH search-key)    propagate request to left child
   (post right-child SEARCH search-key))  ... and right child
 (RESULT                                  "RESULT (from 1 previous
  search) received
   (record search-key data)               extract "search-key" and
  "result"
   (post parent RESULT search-key data)))  forward it to the parent player
```

Figure 2-l. "Data-flow" Implementation of Concurrent Tree Search

```
(defPlayer node
 (REQUEST RESULT)                         messages it understand
 (parent left-child right-child)          acquaintances
 (search-key data)                        internal states
 (REQUEST                                 "REQUEST message received
   (record search-key)
   (post left-child REQUEST search-key)   initiate search...
   (post right-child REQUEST search-key))
 (RESULT (record data)                    result arrives
   (post client RESULT data) ))           forward it to client
```

Figure 2-2. "Parallel-Function-Call" Implementation of Concurrent Tree Search

Class B: ***(e.g. Remote procedures [Nelson 81], "streams" [Weng 75], "fork/join" [Conway 63, Dennis 66], "parBegin/parEnd" [Dijkstra 65], "ForAll/ DoAll", Concurrent Pascal [Hansen 75] and CSP [Hoare 78] etc.)*** — Analogous to function

calls, parallelism is explicitly initiated via message passing between "sender" (c.f. caller) and receiver (c.f. callee) players. After the message is sent (c.f. function call), the sender blocks to wait for its reply (c.f. function return).

Figure 2-2 illustrates the "class-B" implementation of the tree-search example. Recursive waves of message is sent from players at stage "k+1" to those in stage "k". Only one request is allowed to flow down the branches of the tree. At any one time, only players at the same level executes in parallel. All players at higher stages are blocked. The next request will have to wait at the root of the tree until the previous request finishes processing.

2.2 The Multiprocessor Model

The multiprocessors being considered consist of "ensembles of identical, concurrently operating, and regularly interconnected processing elements" [Seitz 82] (e.g. the Cosmic Cube [Su 85]). Each processing element (or "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.

In these experiments, the sites of the multiprocessors are completely connected (Figure 2-3). This topology was chosen in order to eliminate the effects of routing policy — thus provides us a simpler environment for data interpretation.

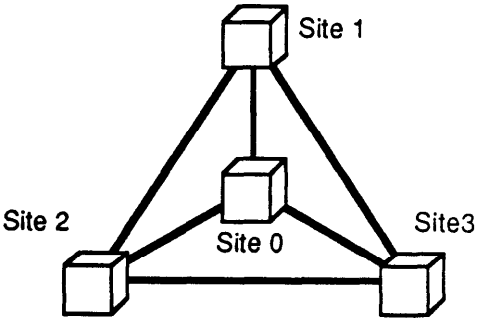


Figure 2-3. Four Completely-Connected Sites

For the purpose of the study, the basic operations of a site is modelled as follows:

i) Local scheduling policy:

a. executable players "time-share" the (only) processor unit in a **ROUND-ROBIN** fashion;

b. players are preempted for "system operations" which include:

OPERATION	TIME TAKEN
• memory management (e.g. player creation/termination)	T_{mem}
• message delivery to local players	$T_{deliver}$
• send a one-packet-message to players in a neighbor site	T_{hop}
• context-switch overhead	$T_{interrupt}$
• routing decisions, scheduling overhead	T_{os}

Throughout these experiments: T_{mem}, $T_{interrupt}$, $T_{deliver}$ and T_{os} are kept small (≈ 0) .

ii) Message sending mechanism — When player **P** at site $<s_p>$ wishes to send a message to player **Q** at site $<s_q>$, the following procedure is carried out:

Operation	Time charged to **CPU** at...
a. message delivered to infinitely large buffer at $<s_q>$;	T_{hop} $<s_p>^{\dagger}$
b. processor at $<s_q>$ alerted;	$T_{interrupt}$ $<s_q>$
c. later*, the operating system of $<s_q>$ delivers the message to the receiver player **(Q);**	$T_{deliver}$ $<s_q>$

† In this case, the time spent in remote communication is charged to the **CPU** at the site in which the sender player reside. An alternative model involves charging T_{hop} to the site of the receiver player. This involves a slightly modification in protocol suggested.

* Although the operating system at site $<s_q>$ can preempt any executing player(s) to handle messages, many messages may arrive at a site simultaneously. Therefore, there is still an indeterminate delay between the time when a message is delivered to the destination site and the time when it is actually delivered to the receiver player. Messages will eventually be received because i) the ports connected to neighboring sites are served in a **ROUND-ROBIN** fashion and ii) incoming messages at each port are served on a **FIRST-COME-FIRST-SERVE** basis,.

2.3 An Exhaustive Search?

These experiments simulates the execution of two 9-players-programs a multiprocessor with 4 completely connected sites. In order to guarantee finding the "optimal mapping", an exhaustive search over all possible mappings was carried out.

"How many ways are there to place **N-P** players into **N-S** sites?" The answer to this question depends on:

a. the program — if some of the players have the same behavior (e.g. communication, synchronization and computation requirements), the number of possible solutions is greatly reduced;

b. whether all the sites are identical; and

c. how the sites are connected (i.e. the topology of the communication network).

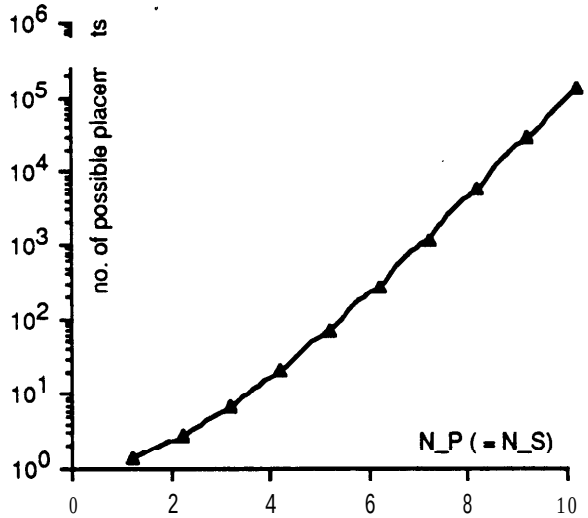


Figure 2-4. Search Space Size for N Players on (at least) N Completely-connected Sites

In the worst case, where all sites/players are different and the topology is non-uniform, there are $N\_P^{N\_S}$ assignments. Even when the sites are completely connected, the total number of possibilities still increases exponentially1 (Figure 2-4). Given the computing resources and time available, the 9-SITE-4-PLAYERS configuration was chosen — it takes 11,05 1 iterations, which translates to 15 hours of simulation time (using one processor on the Sequent Balance-8000) and 1.2 Mbyte of data to be analyzed (Figure 2-5).

N_S	N-P	N_P/N_S	no. of solutions	simulation time	data gathered
9	4	2.25	11,051	15.34 hours	1.2 Mbyte
9	5	2.2	18,002	24.99 hours	2 Mbyte
10	4	2.5	43,947	2.6 days	4.8 Mbyte
10	5	2.0	86,472	5.1 days	9.4 Mbyte

Figure 2-5. Comparing Simulation cost for different N-P and N_S

Beside being economically feasible, the configuration chosen should also post constrains to make the placement problem "interesting". For example, with approximately one site available per player in a completely connected structure (i.e. N-P $\leq$ (or $\approx$) N-S), the solution to the placement problem with low communication cost (obviously) involves placing a player at each site. The higher the average number of players per site is, the more "interesting" the problem becomes. The heuristics have to minimize the contention between players resident in the same site yet at the same time, be aware of the penalty in remote

1 Appendix I contains the full solution to the placement problem with completely-connected sites.

communication when distributing the computation over the machine. If the value of N_P/N_S were too high (e.g. ≈ N-P), the placement problem is also "uninteresting" because total number of possible placement is small (see Figure A-l in the Appendix). The value of N_P/N_S for the 9-SITE-4-PLAYERS configuration was ≈ 2.2 .

2.4 What to Measure

2.4.1 Preliminaries

Figure 2-6 below gives a very brief summary of some of the approaches taken by other researchers to represent a distributed program, a multiprocessor and their interaction (indicated by "⇄"). A "√" indicates that the particular model was used, "×" for "not used" and "1" indicates the use of "cost models".

	Model	Program	⇄	Machine	References
1.	cost model	√	√	√	[Stone 77/78, Lo 81, Steele 85]
2.	precedence graph	√	1	1	[Sankar 87]
3.	connection graph	√	1	√	[Bokhari 8 1, Gylys 76]
4.	queuing model	√	√	√	[Ni 81, Gao 84, Chow 79]
5.	message count	√	x	x	[Miller 85]
6.	load indices	x	√	√	[Efe 82, Stankovic 8 1]

Figure 2-6. Descriptions of Concurrent Execution Environments

Although some of these approaches produced resource management strategies with performance improvements, these results were, nevertheless, somewhat unsatisfactory:

i) Many of these approaches were concerned with minimizing the average turn-around time for a group of unrelated/independent tasks — the results of which are not directly/readily translatable to the context of managing a tingle computation.

ii) Many researchers have defined "optimal mapping" without giving sufficient justification that such measurements indeed relate to program execution time.

iii) Approaches that based on abstract program models produce results that are not directly applicable to "real" programs.

"Post-game analysis", on the other hand, does not rely on any abstract program or machine model. Instead, it makes use of "measurable quantities" actually gathered during program execution to produce resource management strategies directly incorporable into distributed operating system kernels. In order to attain a better understanding about the resource management problem and why "post-game" works, more than twenty parameters

were measured. These parameters are classified as follows when used in "post-game" analysis:

i. Site descrintors (ω_s) — describe the utilization and contention of the resources at a site;

ii. Plaver descrintors (ω_p) — describe the resource requirements (or characteristics) of individual players;

iii. Inter-Plaver descrintors (ω_{pq}) — describe the interaction between players which includes parameters that describe the amount of communication between them and their inter-dependencies.

iv. Inter-Site descrintors (ω_{st}) — summarizes the interaction between players resident in different sites.

To facilitate their descriptions, here are some more definitions of the terminology used:

- N_S — total number of sites
- N_P — total number of players
- N_PQ — the total number of possible player pairs = $C_2^{N\_P} = \frac{N\_P}{2}$ (N-P - 1)
- $\sum^{S} \omega_s$ — sum of a site descriptor (ω_s) over all sites
- $\sum^{p}$ □□ sum of a player descriptor (ω_p) over all players
- $\sum^{p \rightleftarrows q} \omega_{pq}$ — sum of an inter-player descriptor (ω_{pq}) over all possible players pairs
- $\sum^{p \in s} \omega_p$ — sum of a player descriptor (ω_p) for players that reside in the same site.

For our purposes here however, they are classified into five groups describing different aspects of the execution environment. All the parameters are in time units (with the exception of N_MSG).

2.4.2. " Overall" Measurements

§ Execution time (T_{exec}): Because the "optimal" mapping is defined as the mapping with minimal execution time, this quantity is used to compare the relative "goodness" of among different mappings. Speed-Up (S) is directly deducible from the execution time: $\frac{T_0}{T_{exec}}$ where T_0 = program execution time when only one site is used.

§ The number of remote messages sent (N_MSG): A message is considered "remote" when it is passed between two players that reside in different sites.

2.4.3. CPU Utilization

§ **CPU** time consumed at the busiest site (H **CPUS**) — The "busiest" site is the one whose processor idles the least. It should be noted that the processor is used both for "user computation" ($T_{compute}$) as well as operating system functions such as message passing (T_{msg}). $T_{compute}$ only depends on input data variation for a particular program. In this experiment, T_{mem}, $T_{interrupt}$, $T_{deliver}$ and T_{os} are small and, therefore, ignored. For the processor at each site — $T_{CPU} \approx T_{compute} + T_{msg}$.

§ Average **CPU** time consumed per site (**A_CPUS**) = $\frac{\sum^{S} T_{CPU}}{N\ S}$

§ "Load balance" (USR_{std}) is defined as the standard deviation among the **CPU** utilizations over all sites. When USR_{std} **= 0, the sites are** "loaded evenly".

2.4.4. Communication

§ **CPU time** used for communication at the site from which most remote messages are sent: (**H_MSGS**)

§ Average time used for sending messages per site (**A_MSGS**): $\frac{\sum^{S} T_{msg}}{N\_S}$

2.4.5. Contention

Player ID	Execution Profile			Contention Caused		
	Ready	Start	End	<A>	<B>	<C>
<A>			30	-	15	20
<B>	15	30	47	0	-	17
<C>	10	47	-	0	0	-

Figure 2-7. An Example: Execution Profile and "Contention"

There are many ways in which "contention" can be measured at a site. Instead of choosing the length of the ready-queue as many researchers have, 9 parameters are proposed here as possible candidates. All of them have "time" as the dimension. In Figure 2-7, between **<A>** and **<B>** for example, $T_{contention}$ = 15 — i.e. player **<B>** has to wait 15 time units in the "ready-queue" or player **<A>** to finish

execution before it can get hold of the processor[1]. Only players that reside in the same site can contend with one another.

§ The highest contention (i.e. $T_{contention}$) (H_CONT_{pp}) experienced between a pair of players — e.g. in Figure 2-7, H_CONT_{pp} = 20 (between <A> and <C>)

§ The average contention experienced per player pair ($A\ CONT_{pp}$): $\sum_{p \rightleftarrows q} T_{contention}$, N_PQ)

§ The "total contention received" (or "queue-time") by a player (from others) essentially indicates the total time it spent ($T_{ready\text{-}q}$) waiting for others to finish using the CPU. $H\_CONT\_RT_P$ describes the player which spent the longest time in the "ready queue" (e.g. in Figure 2-7, $H\_CONT\_RT_S$ = 37 received by <C>). For player <P>, $T_{ready\text{-}q} = \sum_{q \in s} T_{contention}$ over all <Q> that resides in the same site.

§ The average "queue-time" ($A\_CONT\_RT_P$) per player indicates the average time a player spent in the "ready queue" before getting hold of the processor.

§ The "total contention caused" by a player (to others) indicates the total time other players spent waiting for it to finish using the CPU. $H\_CONT\_CT_P$ describes the player which causes the most "total contention" (e.g. in Figure 2-7, $H\_CONT\_CT_S$ = 35 caused by <A>).

§ $A\_CONT\_CT_P$ represents the average "total contention caused" per player.

§ The "contention sum" at a particular site ($\sum_{p \in s} T_{ready\text{-}q}$) is defined as the total time resident players spent in the ready-queue. H_CONT_S describes the site which has the "highest "contention sum".

§ A_CONT_S describes the average "sum of contention" per site.

§ Contention balance ($CONT_{std}$) is defined as the standard deviation of the "contention sum" ($\sum_{p \in s} T_{ready\text{-}q}$) over all sites.

2.4.6. Dependencies

During the life-time of a player, it is always in one of three possible states:

i) "ACTIVATED" — when it is ready to make use of (or using) the processor computing or sending/receiving messages;

---

[1] Throughout this section, the term "$T_{contention}$" represents the time a player spent waiting (in the "ready queue") for the CPU.

ii) **"BLOCKED"** — when it is waiting for a specific "reply" (message) from some specific player; or

iii) "IDLE" — any other occasions in which the player is not computing, waiting to be "activated" by the next message.

Players in "class A" (c.f. Section 2.1) applications *are* either *activated* or *idle* but never *blocked*.

§ When a player <A> is activated by a message sent from player <B>, the time <A> spent in this "idle" period is defined as the "idle time" between the two players (T_{idle}). The player pair which exhibits the highest idle time is described by $\mathbf{H\_IDLE_{pp}}$.

§ Average idle time between two players ($\mathbf{A\ IDLE_{pp}}$) is defined as: $\frac{\sum^{p \rightleftharpoons q} T_{idle}}{N\_PQ}$

§ When a player <A> is blocked waiting for a reply from player <B>, the time <A> spent in this "blocked" period is defined as the "block time" between the two players ($T_{blocked}$). The player pair which exhibits the highest "block time" is described by $\mathbf{H\_BLOCK_{pp}}$.

§ Average idle time between two players ($\mathbf{A\_BLOCK_{pp}}$) is defined as: $\frac{\sum^{p \rightleftharpoons q} T_{blocked}}{N\_PQ}$

§ The site which exhibits the largest sum of "idle time" ($\sum^{p \in s} T_{idle}$) for players resident is described by $\mathbf{H\_IDLE_S}$.

§ The average sum of "idle time" per site is defined as $\mathbf{A\_IDLE_S}$

§ The site which exhibits the largest sum of "block time" ($\sum^{p \in s} T_{blocked}$) for players resident is described by $\mathbf{H\_BLOCK_S}$.

§ The average sum of "block time" per site is defined as $\mathbf{A\_BLOCK_S}$

3 Benchmark I: "Pipeline"

3.1 Introduction

The first benchmark being tested is schematically represented in Figure 3-l. Appendix II contains a listing of this computation. Requests (coded as messages of different kinds) are generated by player <1> and propagate "downstream" from left to right. Each player has two acquaintances [1] — indicated by "virtual links" ("▒▒▒" and "▬▬") to the "right-hand-side" of each player. After a player finishes processing a request, the result it generates is then passed to one of its acquaintances for further processing[2].

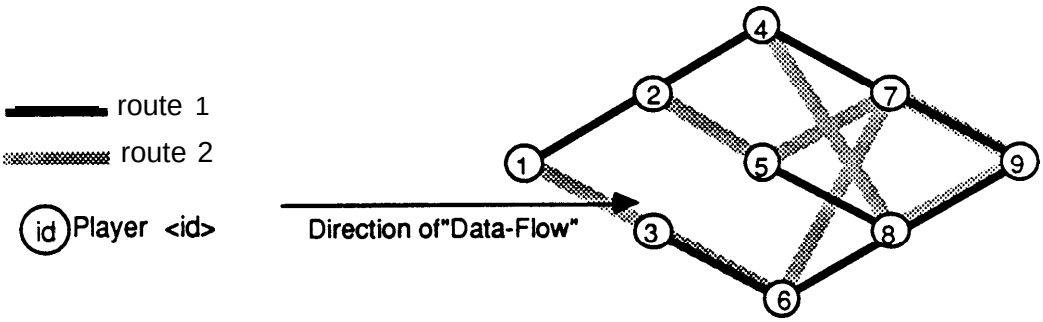


Figure 3-l. The "Pipeline" Benchmark

3.2. Overall Profile

Figure 3-2 gives a summary of the behavior of this program when mapped onto a multi-processor with four completely connected sites. It can be seen that the range of attainable speed-ups decreases as communication cost increases (in fact this is true for both benchmarks).

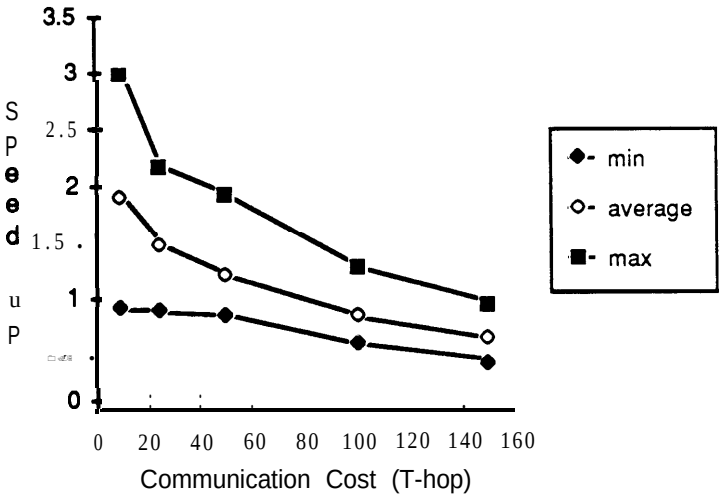


Figure 3-2. Distribution of "Speed-Up"s

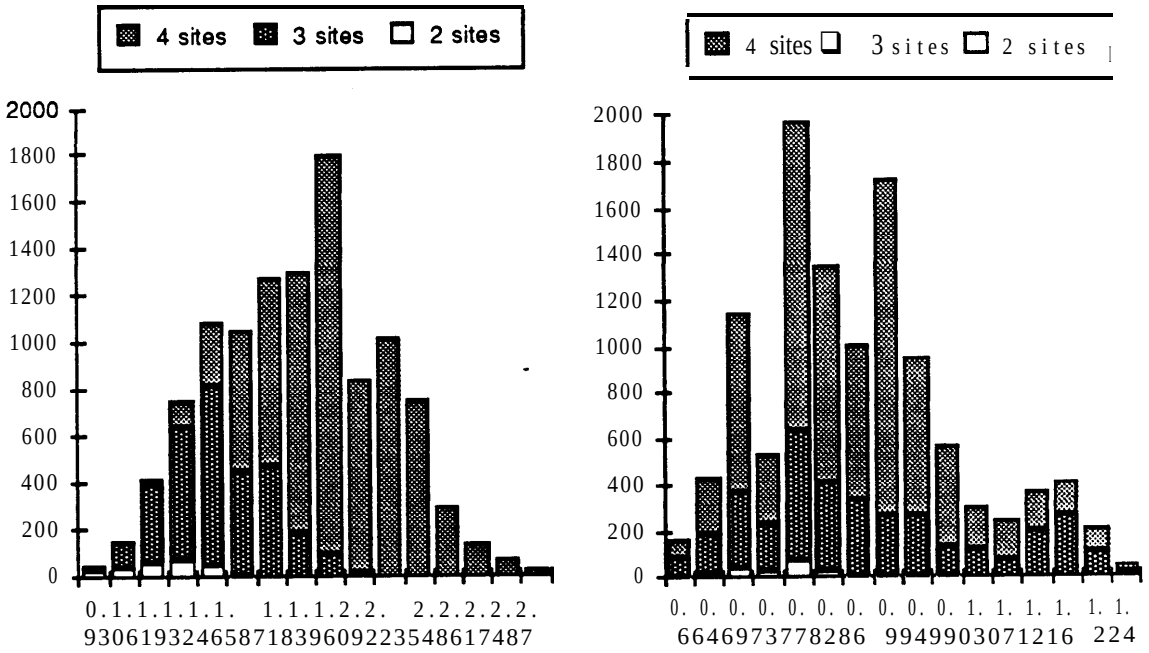
---

[1] The "acquaintances" of a player <A> are those whom <A> "knows the address of" or "holds a pointer to". A player can only send messages to its acquaintances.

[2] When both links are connected to only one player downstream (e.g. from <3> to <6>), all results are forwarded to thatpalticularacquaintance.

It seems that when communication cost is high, spending a lot of effort tackling the assignment problem is not worthwhile since the best solution attainable may not differ much from one generated randomly. When the communication cost is low however, a good assignment makes a lot of difference in execution time.

3.3. Distribution of Execution Time



(a) T_{hop} = 10 **(b) T_{hop} = 100**

Figure 3-3. Speed-Ups "Classified"

Mappings may be conveniently classified into four categories according to the number of sites that actually have players resident. Figure 3-3 illustrates the distribution of "speed-up"s attained by each category at high and low communication costs[1]. Two observations canbemade:

1. The "shape" of the distribution in both cases indicate that mappings randomly generated (most probably) achieve a speed up of about one-half of the optimal (or maximum).

[1] In response to each message received, a player expand an average of 100 time units. A value of 10 for T_{hop} is considered to be "low" because the time spent in communication is only 10% of that spent on computing. A value of 100 for T_{hop} is considered high as the time spent communicating is approximately equal to that spent in computing. For values of greater than 100 time units, the assignment problem becomes very uninteresting because the best speed-up attainable is 1. In other words, distributing the computation over more than one site does not help at all!

2. When communication cost is low, mappings that utilizes all four sites generally achieve higher speed-ups than those which do not — e.g. in Figure 3-3b, there are more mappings with three sites loaded than those with four for which speed up is > 1!

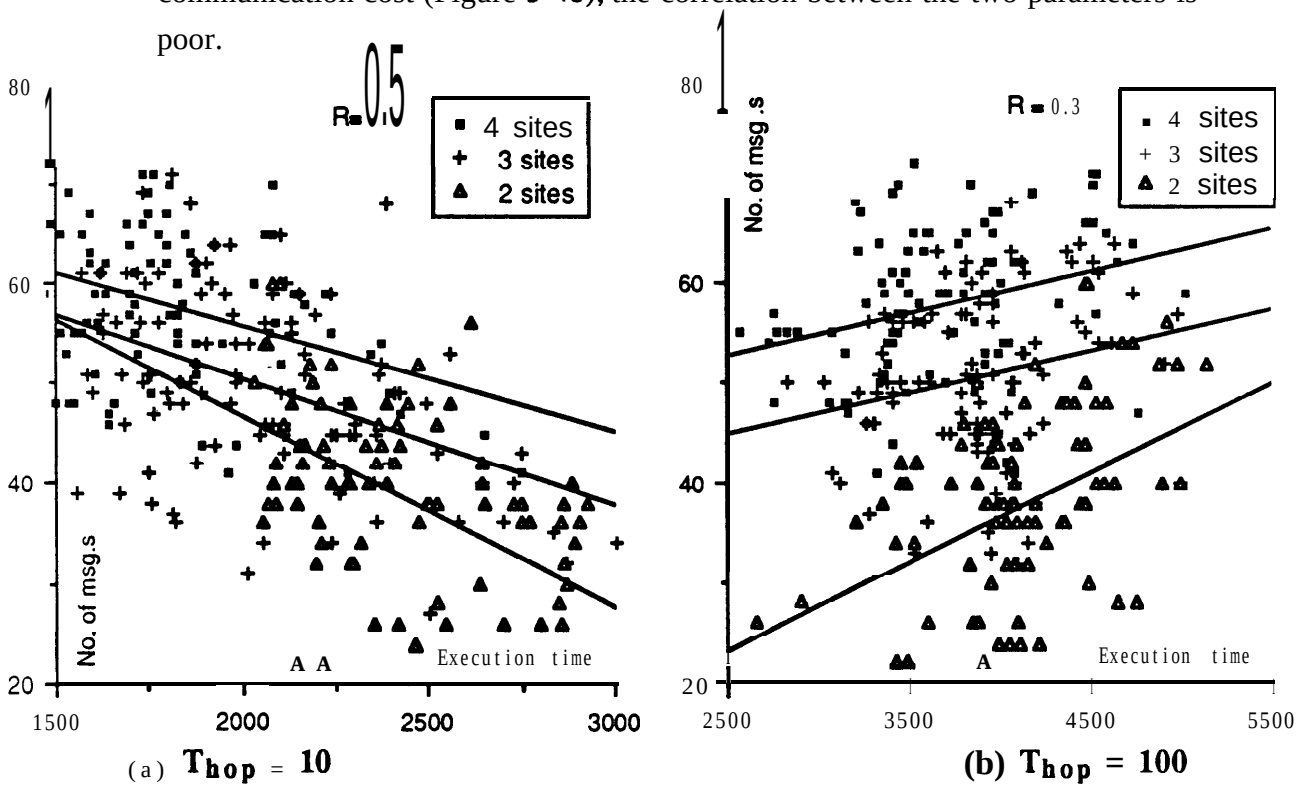
3.4 "Remote" Message Count

Many people believed and preached that minimizing communication cost is an important goal (if not the objective function) when considering alternative mappings. "Remote" message count or the total time spent in sending/receiving messages has been commonly used for such measuring communication cost. The experimental data here (shown in Figure 3-4) demonstrates that:

i) The correlation between the number of "remote" messages and the execution time is poor and decreases with increasing communication cost.

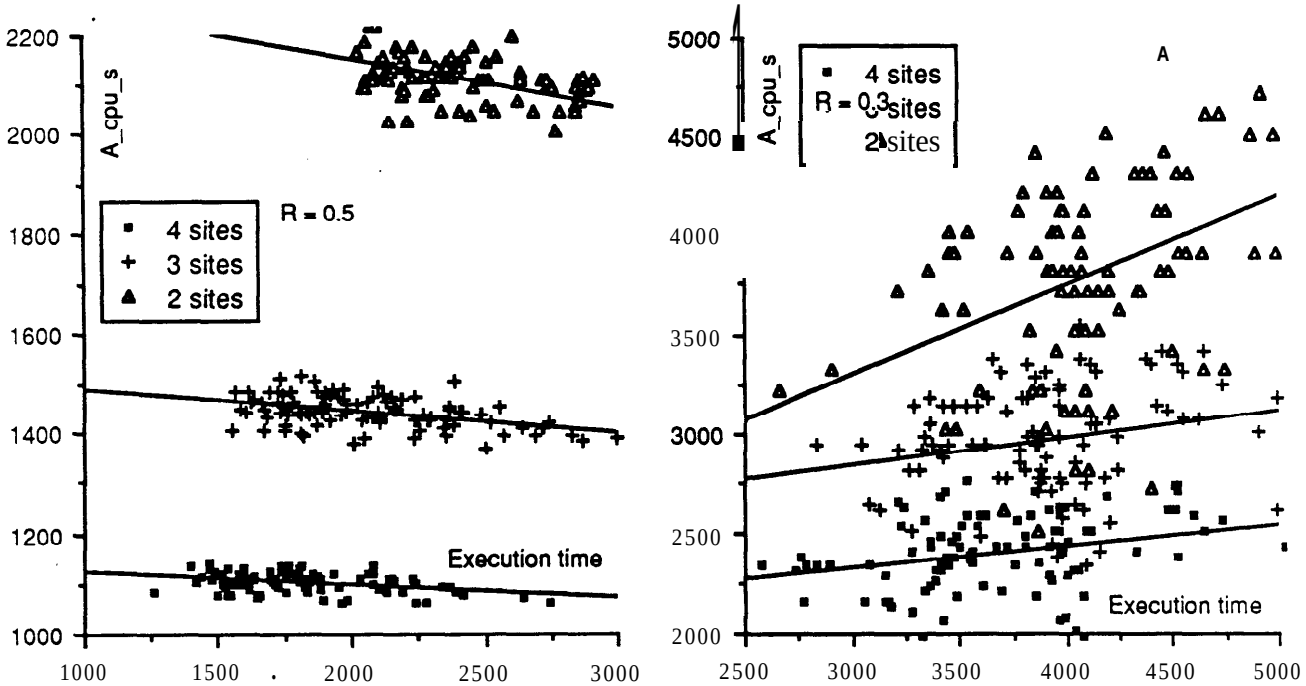
ii) In Figure 3-4a, execution time decreases with increased traffic. When communication cost is low compared with the amount of computation required, the more "distributed" the players are, the faster the program would execute.

iii) Although execution time does increase with increased traffic at high communication cost (Figure 3-4b), the correlation between the two parameters is poor.



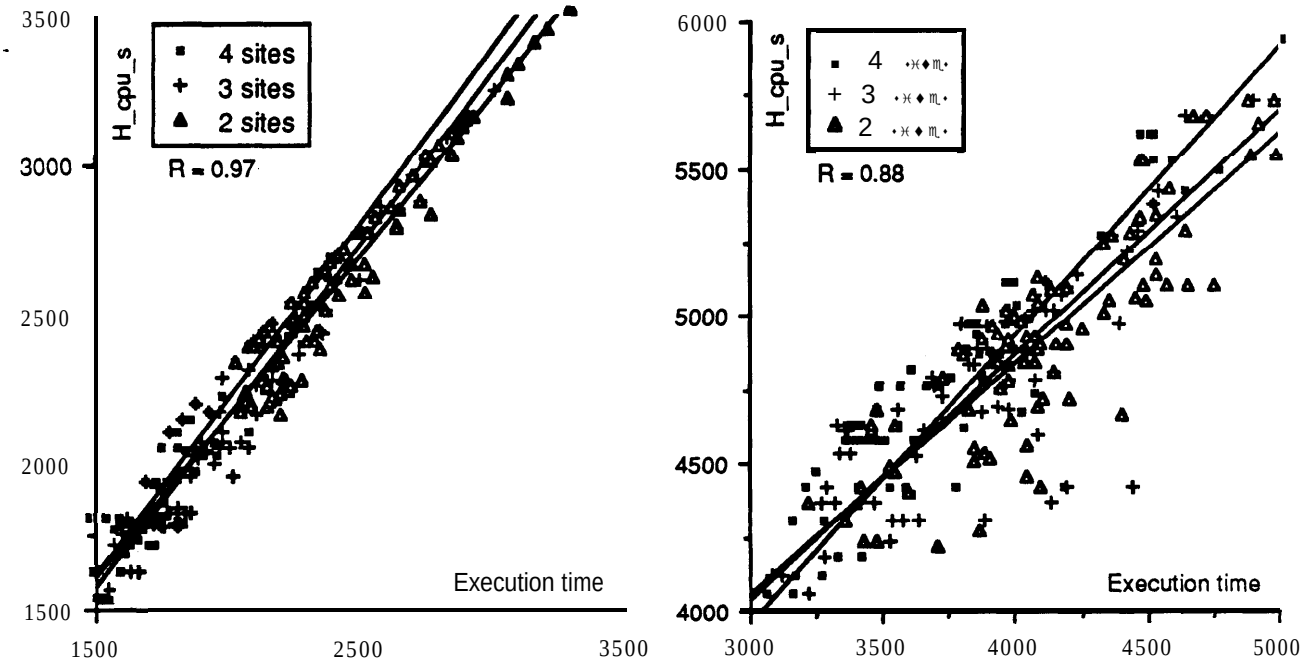
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-4. Relating "Remote" Messages with Execution Time



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-5. Average CPU Time Used per Site



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-6. Highest CPU Time Used at a Site

When interpreting the figures, it should be noted that:

i) There are three sets of points on each graph representing the mappings in which 4 ("▪"), 3 ("+") or 2 ("▲") sites are used.

ii) A line that best-fit each set of points is plotted on the graphs as well. The best-fit minimizes the root-mean-square of the errors.

iii) the average correlation coefficient for the three lines ("R") is also indicated.

3.5. CPU Utilization

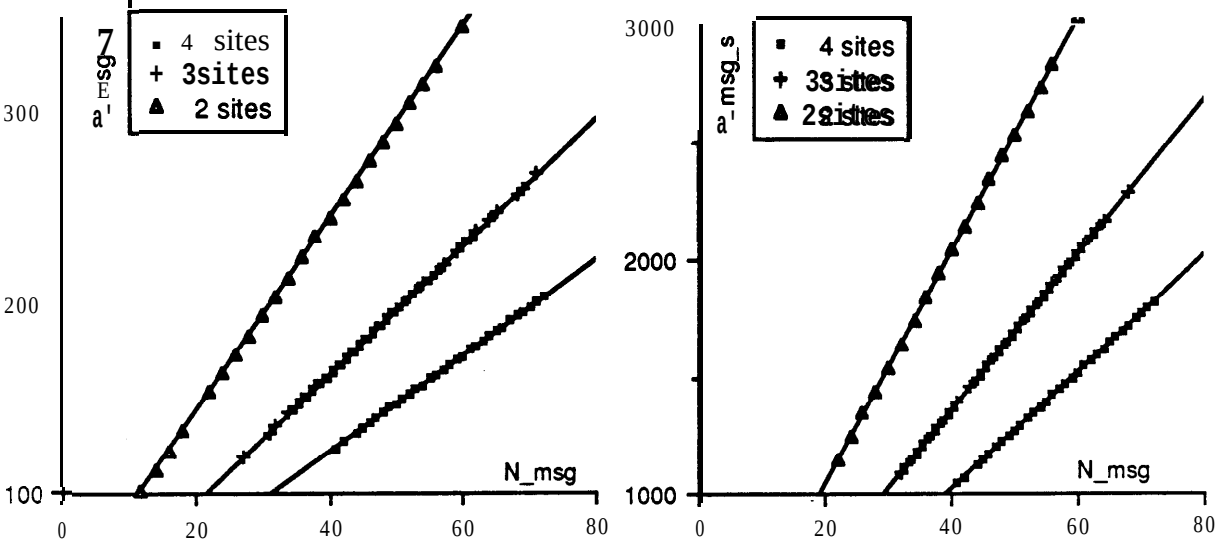
The average **CPU** utilization per site used (A_CPU_S) and highest **CPU** utilization at the "busiest" site (H_CPU_S) are plotted against execution time in Figures 3-5 and 3-6 respectively. It is observed that:

i) The correlation between A_CPU_S and execution time is poor and deteriorates with increasing communication cost.

ii) Correlation of H_CPU_S **with** execution time are good in both cases. (Figure 3-6)

iii) The high correlation between H_CPU_S and execution time suggests that the total execution time depends on the operation of the "busiest site" of the multiprocessor — which is also the bottle-neck of the system.

3.6. Communication

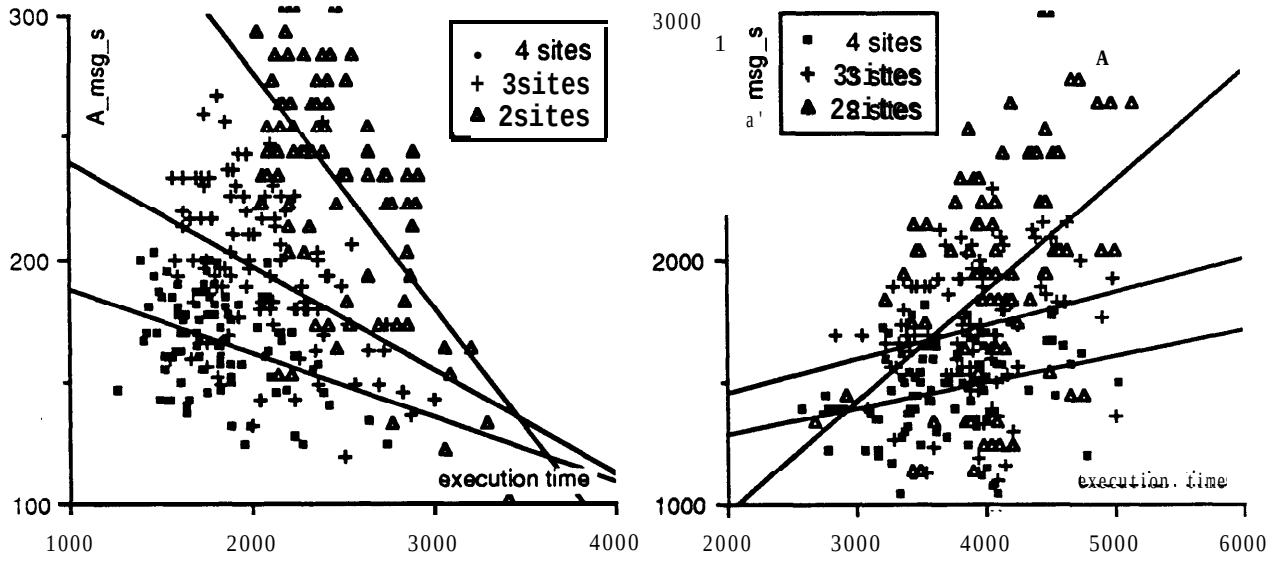


(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-7. Number of Remote Messages vs. the Time Spent Sending Them.

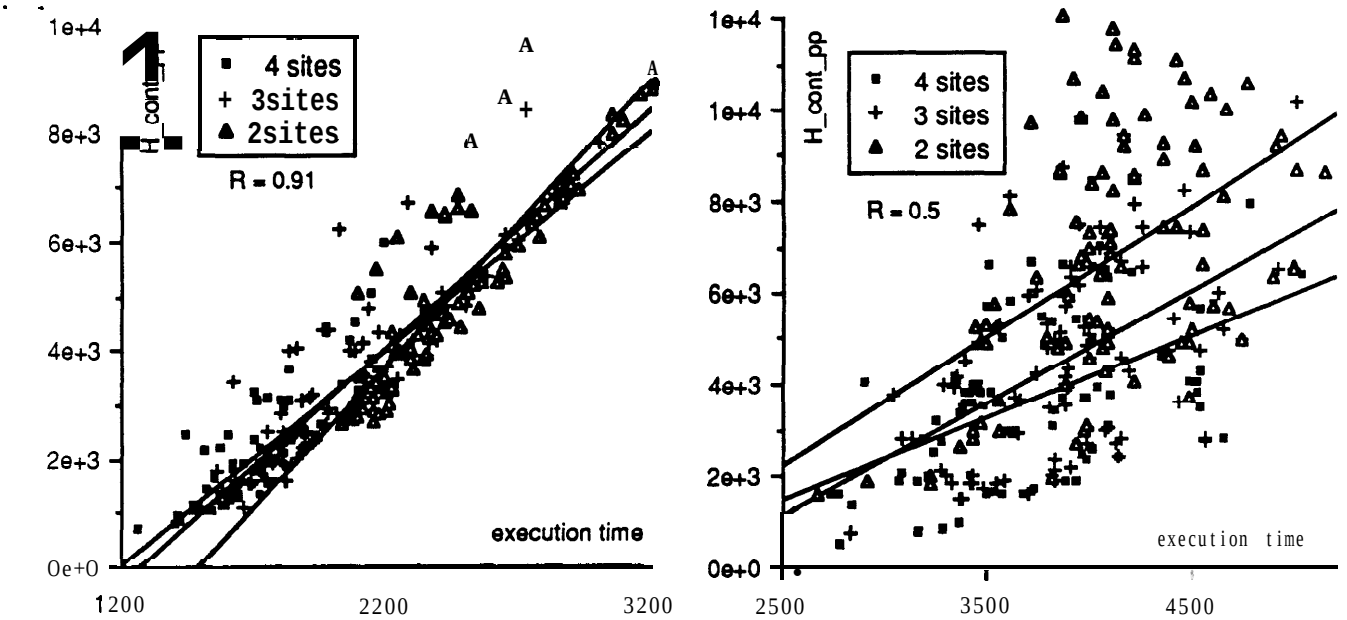
Figure 3-7 suggests that the number of remote messages is directly proportional to the average time spent to route them. Figure 3-8 confirm the results reported in section 3.4 — namely that time spent sending remote messages correlates badly with the total execution

time. It turned out that H_MSGs also did not correlate well with execution time. They follow the general distribution of A_MSGs and are not plotted here.



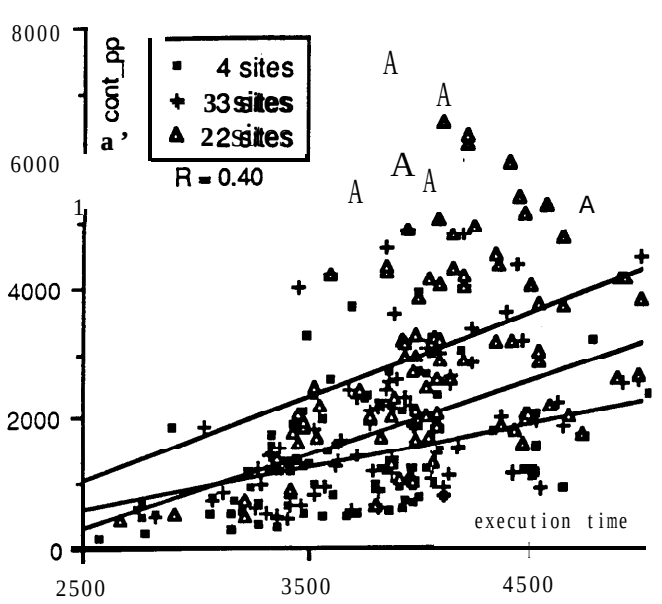
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-8. Correlating the Average T_{msg} with Execution Time

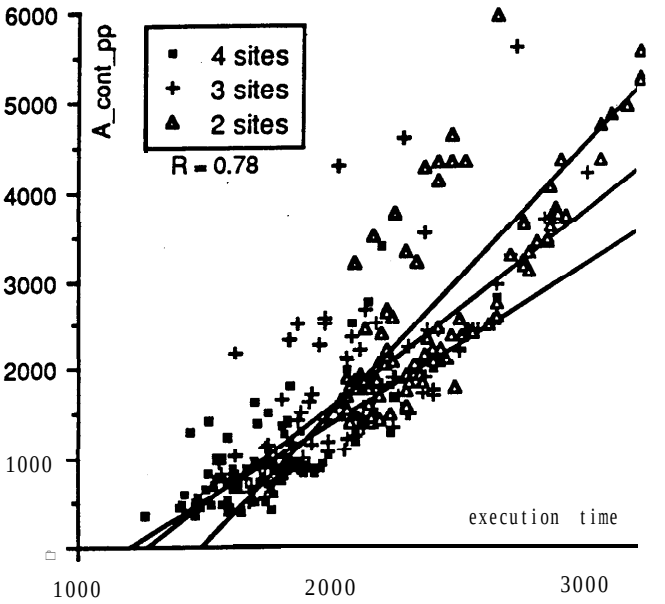


(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-9. Highest "Contention" Experienced between Two Players

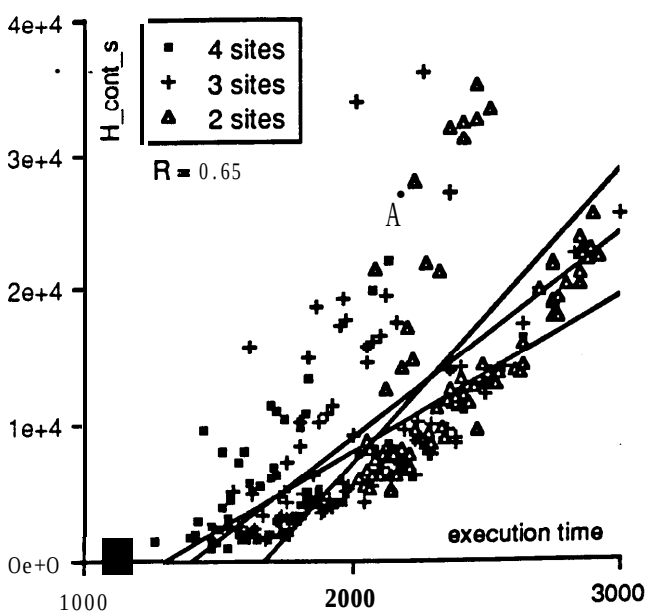


(a) T_{hop} = 10

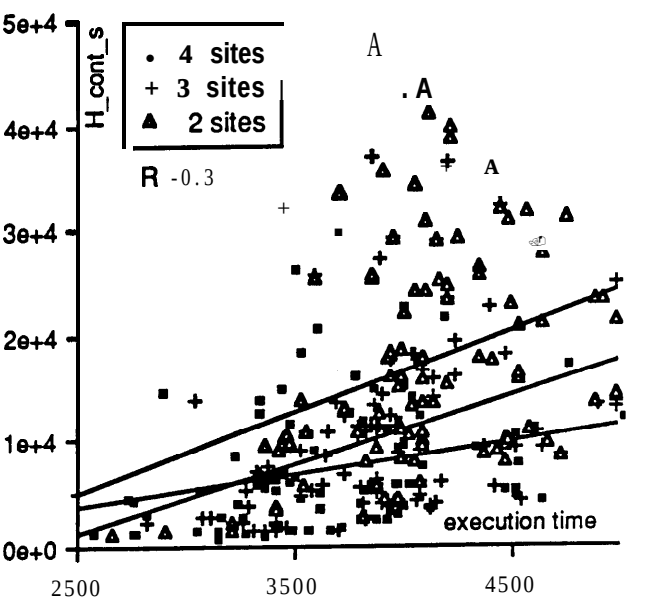


(b) T_{hop} = 100

Figure 3-10. Average "Contention" Experienced between Two Players.



(a) T_{hop} = 10



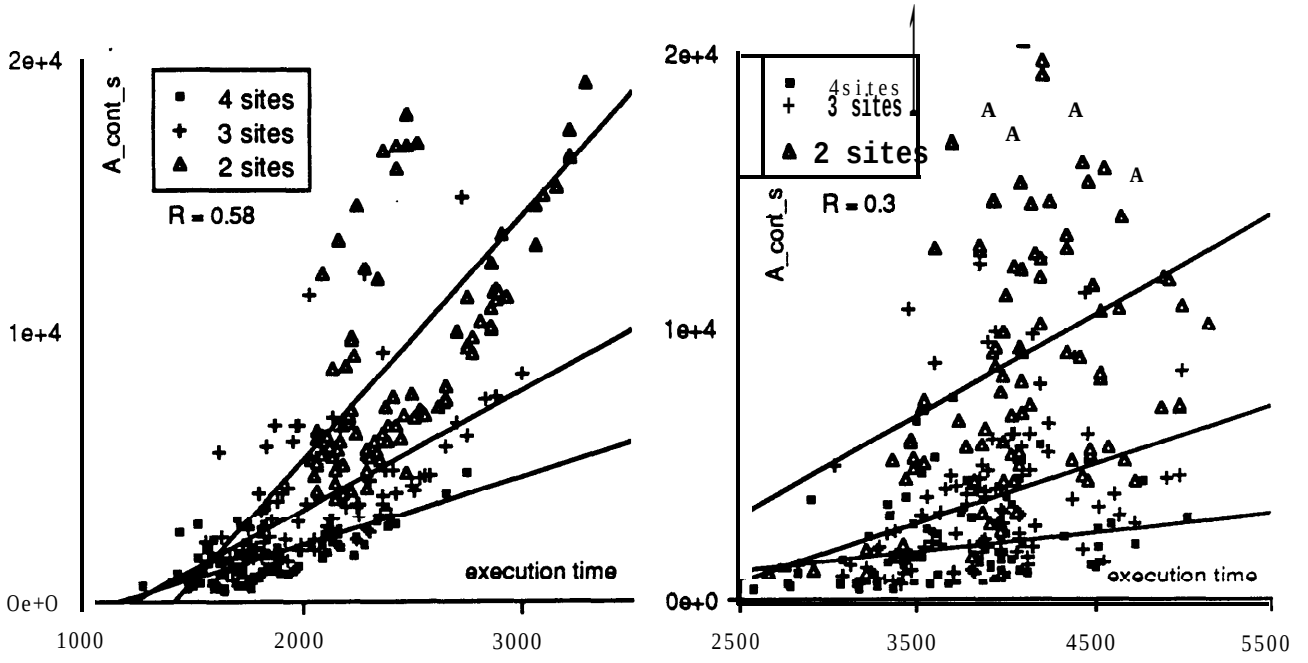
(b) T_{hop} = 100

Figure 3-11. Highest Total "Contention" Experienced by Players at a Site.

3.7. "Contention"

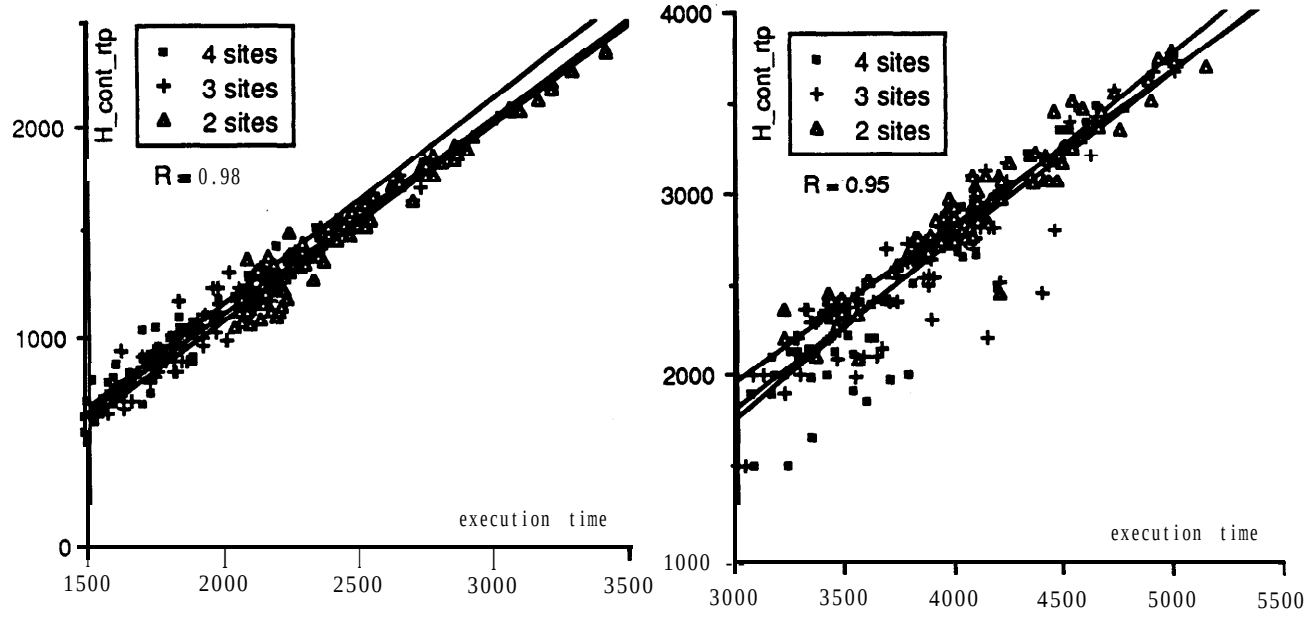
The contention measurements are illustrated in Figures 3-9 to 3-14. Four points should be noted:

i) In general, choosing a mapping that exhibit lower contention (measured by any of the six parameters proposed) results in a shorter execution time".



(a) T_{hop} = 10 **(b) T_{hop} = 100**

Figure 3-12. Average Total "Contention" Experienced at a Site.



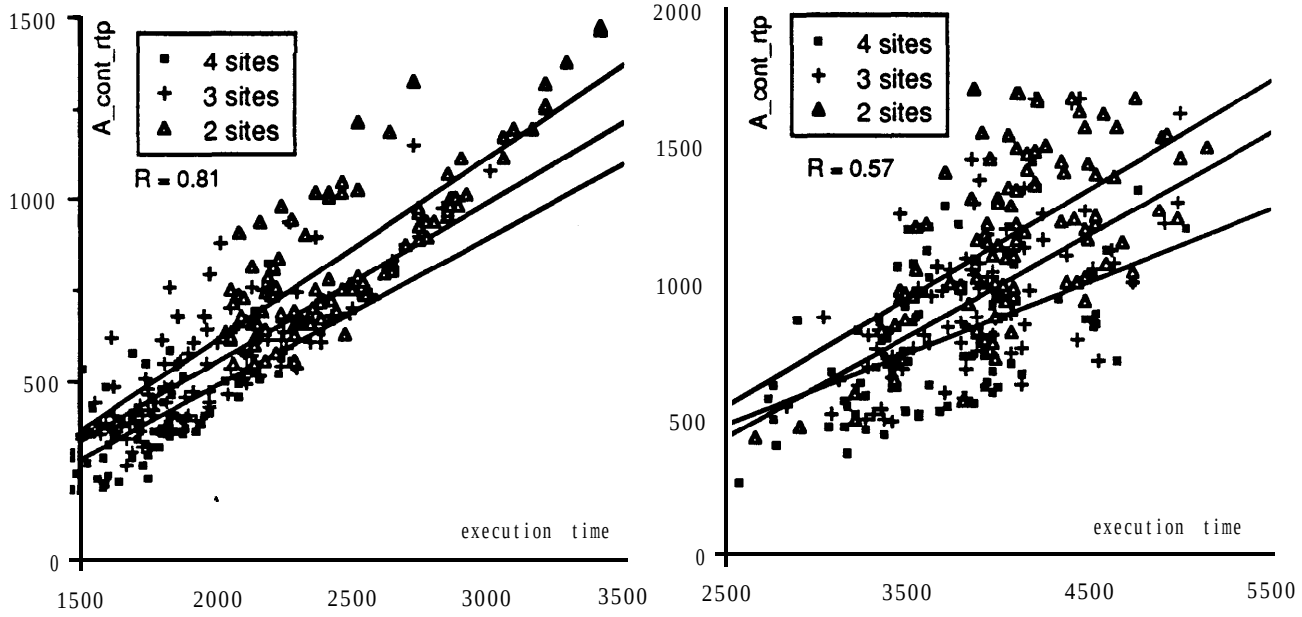
(a) T_{hop} = 10 **(b) T_{hop} = 100**

Figure 3-13. Longest CPU Wait Time.for a Player

ii) The correlation between contention and execution time also decreases with increased communication costs.

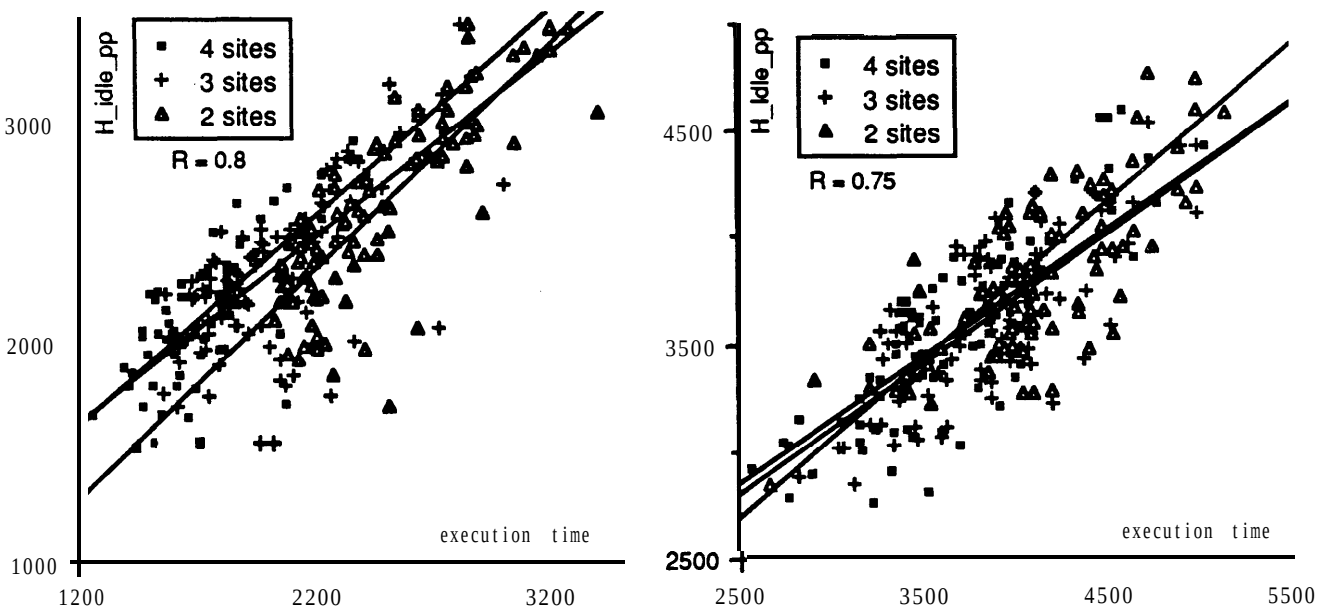
iii) The best correlation is observed in Figure 3-13 — between the "longest total time a player spent waiting to use the **CPU**" and the execution time.

iv) Although as explained in section 2.4, contention "caused" is different from contention "received", their correlation with execution is very similar. Therefore, the corresponding graphs for $H\_CONT\_CT_p$, $A\_CONT\_CT_p$ are not plotted.



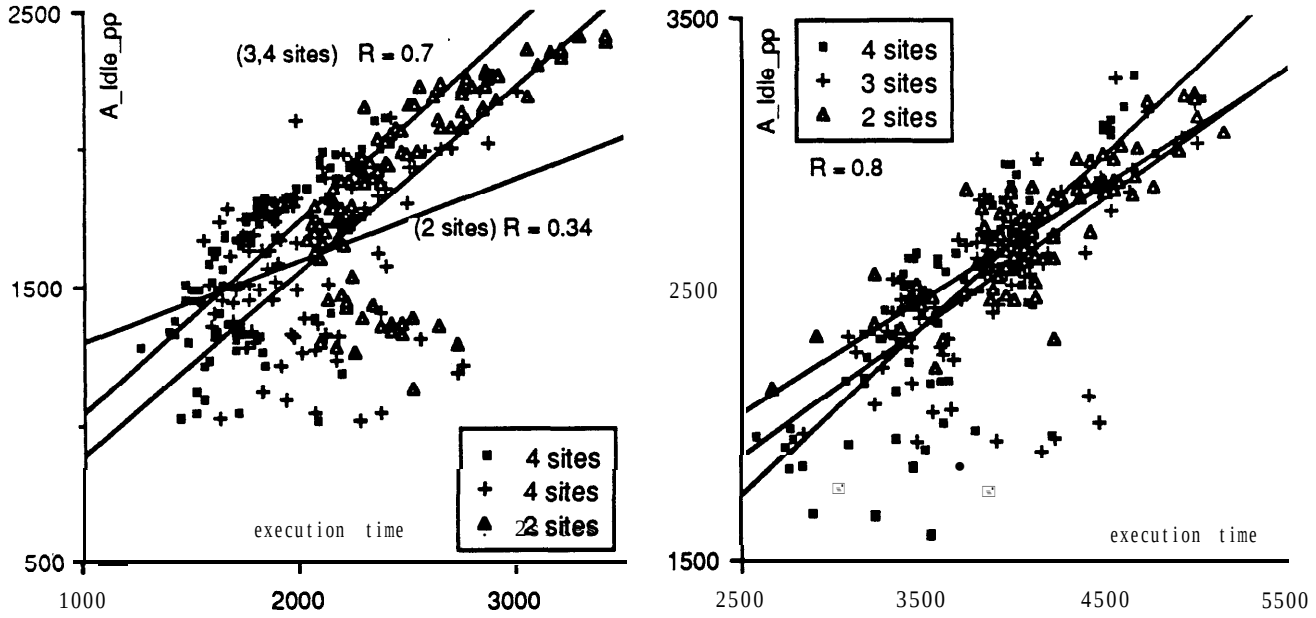
(a) $T_{hop} = 10$ (b) $T_{hop} = 100$

Figure 3-14. Average CPU Wait Time.per Player.



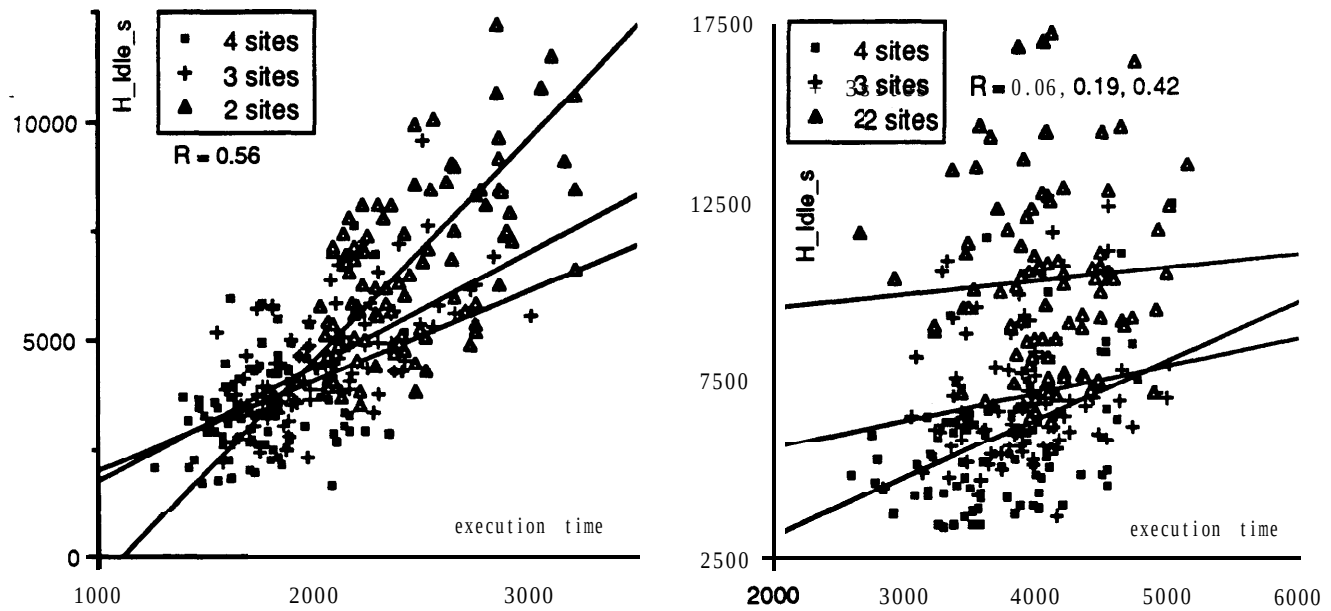
(a) $T_{hop} = 10$ (b) $T_{hop} = 100$

Figure 3-15. Longest Idle Time between a Pair of Players



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-16. Average Idle Time between a Pair of Players

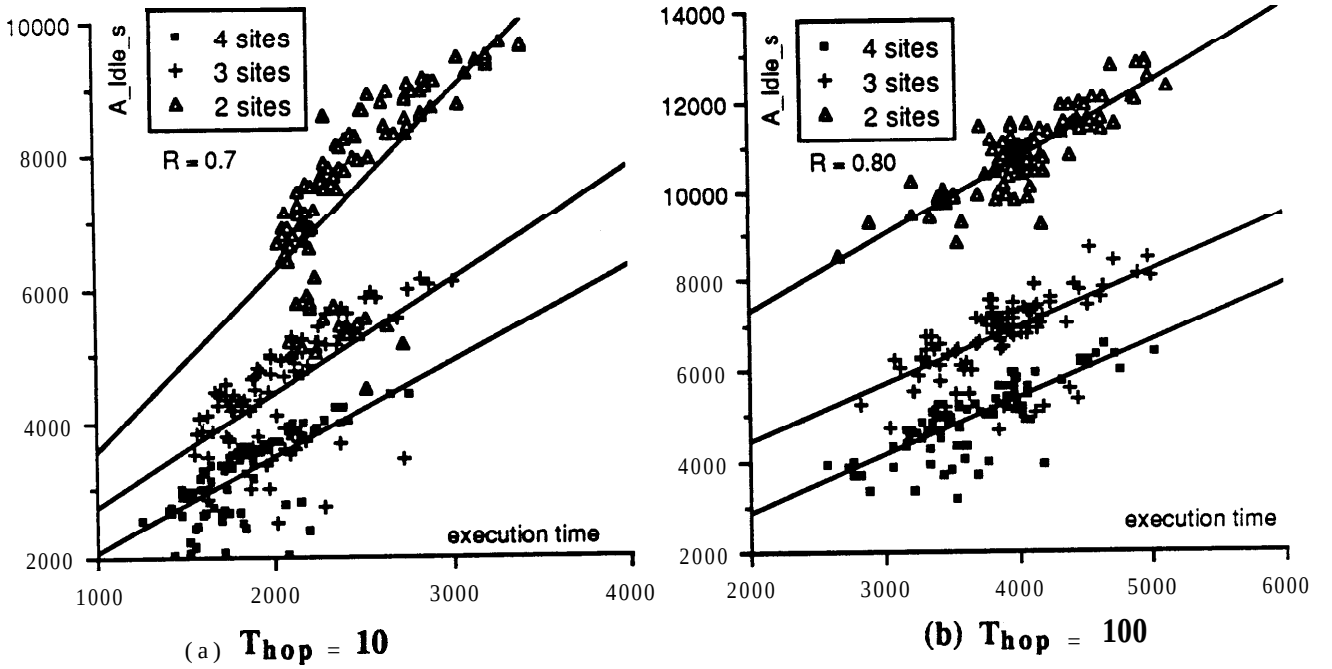


(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-17. Largest Sum of Idle Time between Players in Two Sites

The highest correlation among the six parameters proposed is found in Figure 3- 13 for $H\_CONT\_RT_p$. In other words, the bottle-neck of the system lies in the CPU of the "busiest" site. It should be noted that the correlation here is even better than that for Figure

3-9 — suggesting that the time players spent in the ready-queue is a better indicator for "busy" sites than the **CPU** utilization at the site.



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 3-18. Average Sum of Idle Time between Players in Two Sites

3.8 Dependencies

With this benchmark, no players are ever "blocked" waiting for a specific "reply". Therefore, the parameters H_BLOCK_{pp}, A_BLOCK_{pp}, H_BLOCK_{s}, and **A-BLOC&are** not defined.The **"IDLE"** parameters are plotted in Figures 3- 15 - 3-l 8. Although the general trend of the results are as expected, none of the measured quantities exhibit a very high correlation with the execution time.. The situation for the next benchmark is not the same (see Section 4.7).

4 Benchmark II: Divide and Conquer

4.1 Introduction

The second benchmark being tested is schematically represented in Figure 4-l (BDL listing in Appendix III). A stream of requests are initially presented to player <1>. It partitions the first request into two (not necessarily equal) sub-tasks and delivers them to players <2> and <3>. Player <1> then "blocks" and wait for players <2> and <3> to "reply" before servicing the other requests one by one. Player <2> (<3>) invokes players <4>, <5>, <6> (<7>, <8>, <9>) in parallel, blocks to wait for all their replies before replying to player <1>. Each player also expand an average of 100 time units in response to a message received — part of which is spent creating the sub-tasks and the rest in processing the results received.

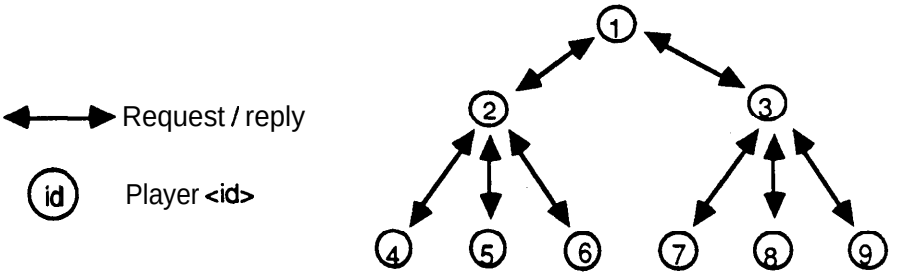
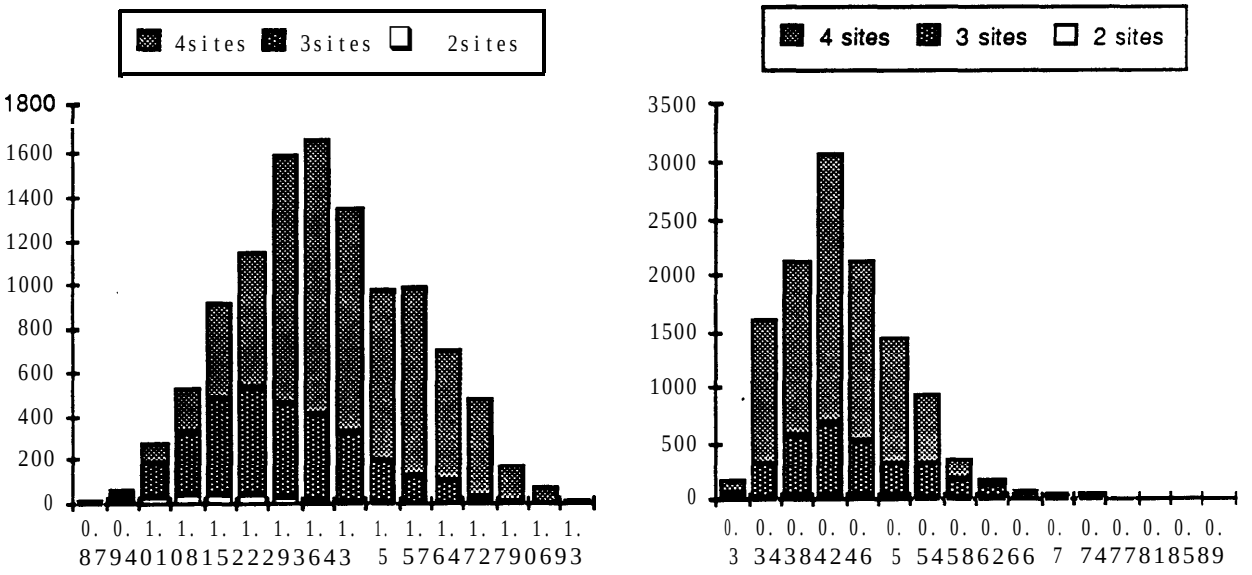


Figure 4-l. The "Divide-and-Conquer" Benchmark



(a) T_{hop} = 10 **(b) T_{hop} = 100**

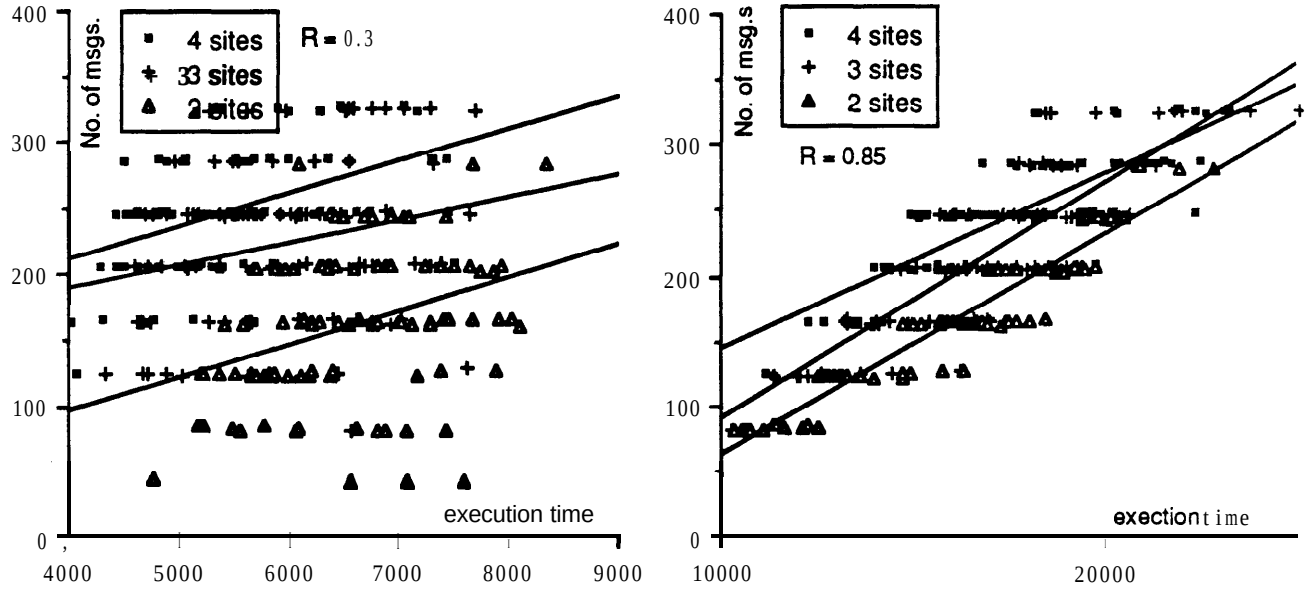
Figure 4-2. Speed-Ups "Classified"

4.2 Summary of Findings

Instead of going through the graphs one by one, a summary is given here at the beginning — comparing the major findings here with the previous benchmark. The interested reader is encouraged to inspect the graphs listed in the rest of this chapter (section **4.3** to **4.7).**

1. The distribution of execution time (Figure 4-2) indicates that the success of the resource management system is also critical for exploiting available parallelism on the multiprocessor for this benchmark — especially when communication cost is low.
2. The correlation between the number of messages routed over the net is worse than that of the previous benchmark (Figure 4-3);
3. so are the parameters describing **CPU** utilization (Figures 4-4, 4-5a, 4-6a).
4. With high communication cost however (Figure 4-5b, 4-6b), the correlation factors for **CPU** utilization increases (as opposed to a decrease for the previous benchmark!).
5. None of the contention figures measured at high communication cost correlates well with execution time (Figures 4-9b, 4-10b, 4-11b, 4-12b, 4-13b, 4-14b)
6. With low communication cost however, the best contention parameter is $H\_CONT\_RT_p$ — which'describes the total time a player spent waiting for the **CPU** in the ready-queue (Figure 4-13a and 4-14a).
7. All "dependency parameters" (Figures 4-15, 4-16, 4-17, 4-18) correlates (with execution time) better than the previous benchmark. In fact, execution time is almost directly proportional to H_IDLE_{pp} (as well as H_BLOCK_{pp}) which describes the longest time a player spent waiting for the next request arriving from its "parent" upstream.

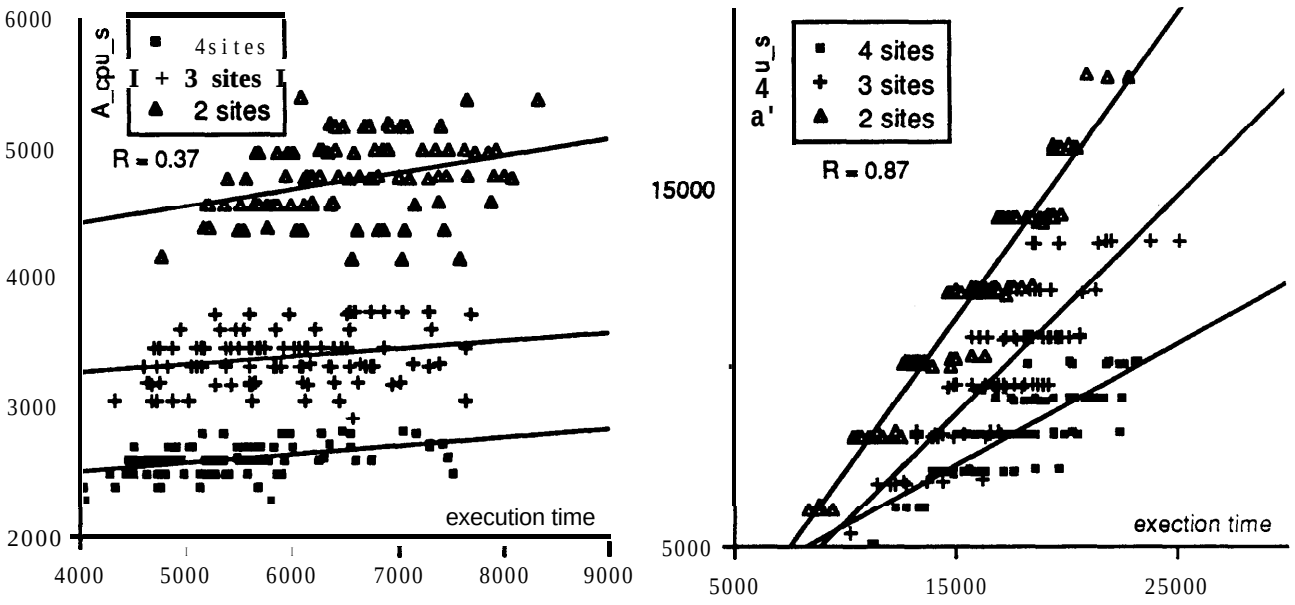
4.3 Remote Message Count



(a) T_{hop} = 10 (b) T_{hop} = 100

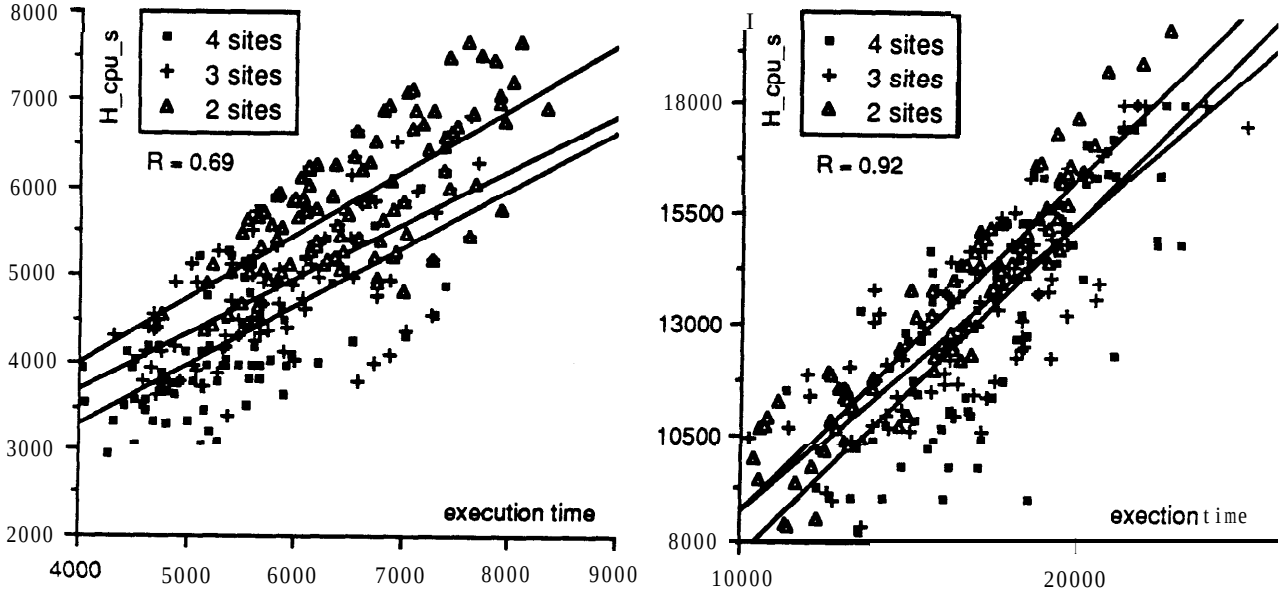
Figure 4-3. Relating "Remote" Messages with Execution Time

4.4 CPU Utilization



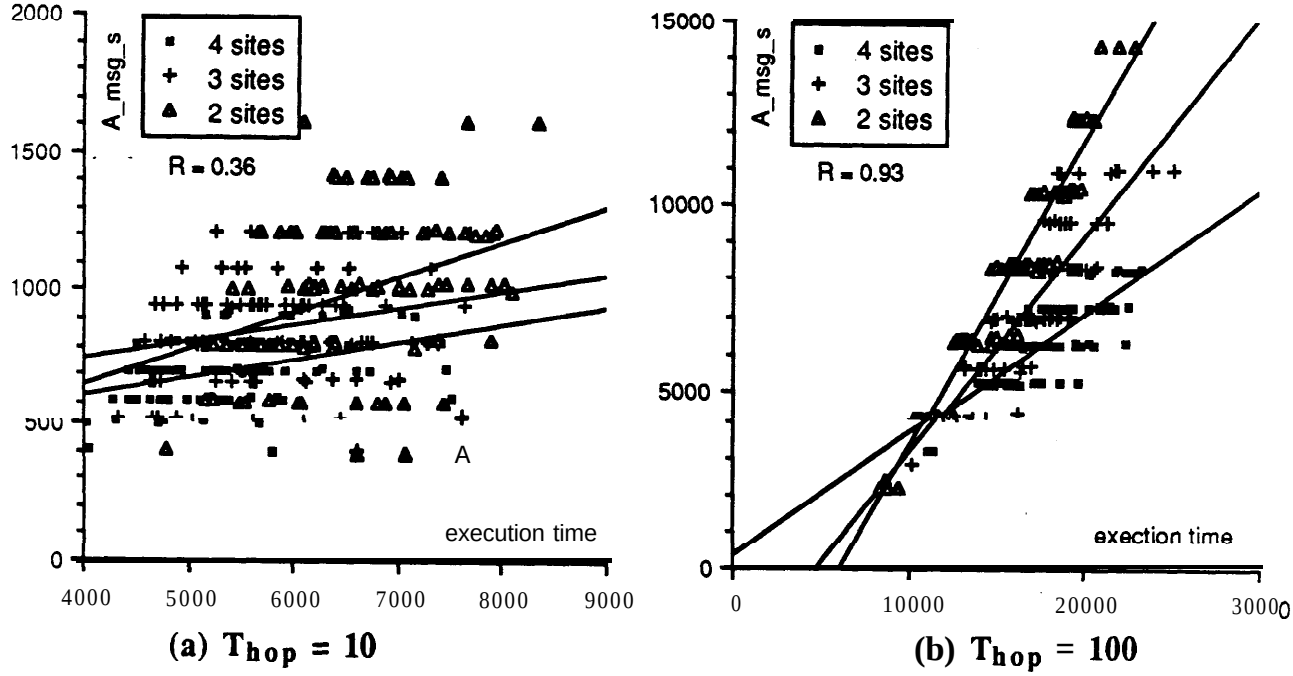
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-4. Average CPU Time Used per Site



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-5. Highest CPU Time Used at a Site

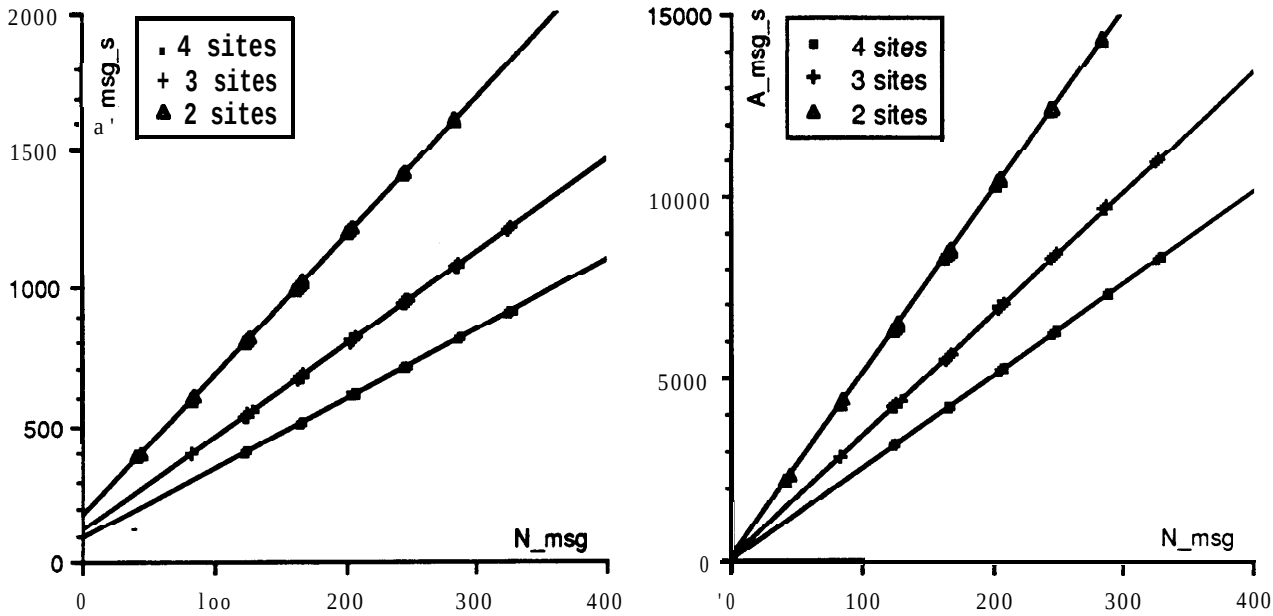


(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-6. Correlating T_{msg} with Execution Time.

Unlike the first benchmark, A_CPU_S and H_CPU_S correlates better with executing time with increased high communication cost. Figure 4-4b bears a close resemblance to Figure 4-3b because the CPU at each site in fact spend more time routing messages than computing. The pattern in Figure 4-4b actually reflect the number of messages routed.

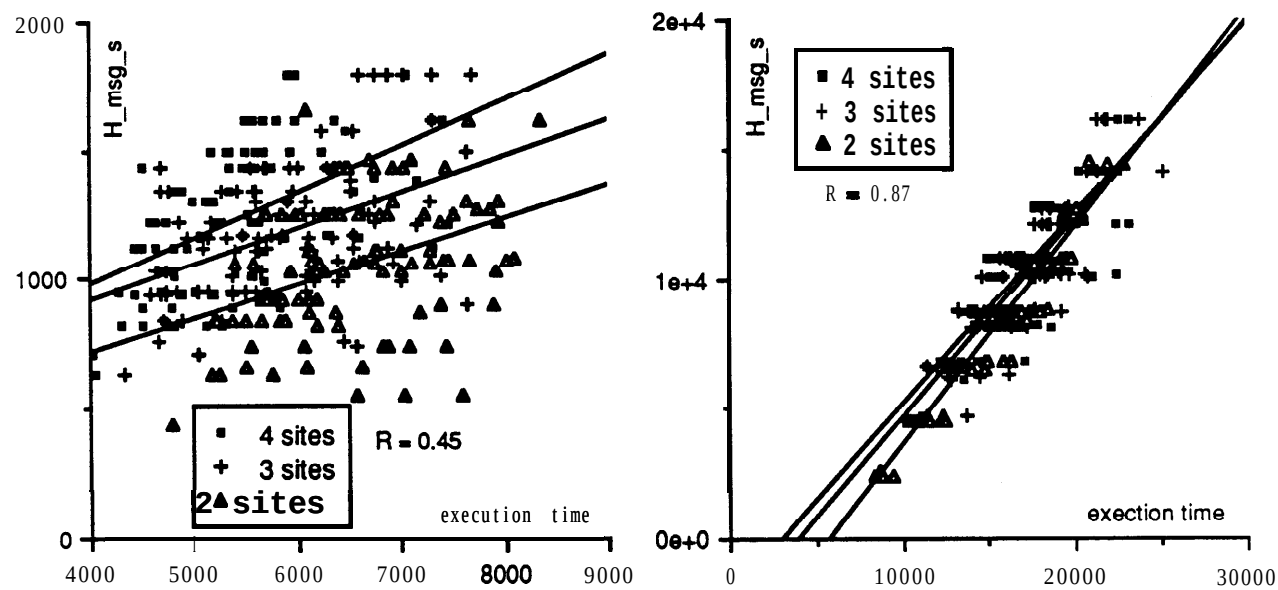
4.5 Communication



(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-7. Number of Remote Messages vs. the Time Spent Sending Them

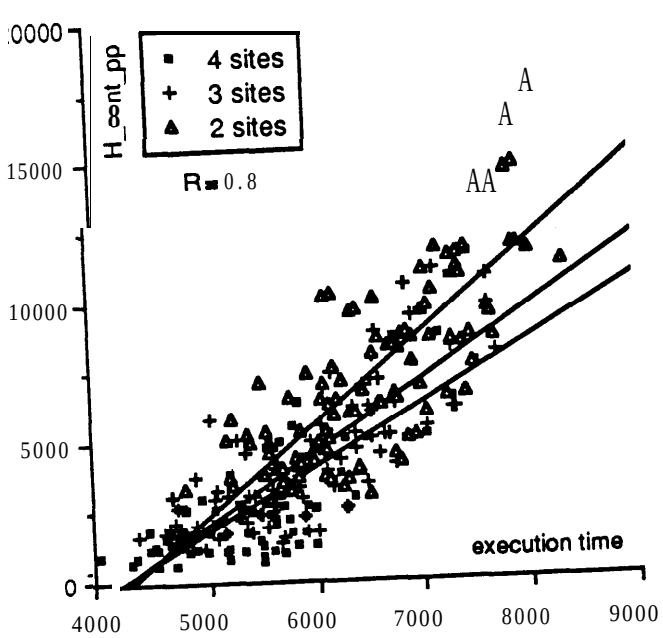
Unlike the previous benchmark, the correlation factors for A_MSG_S (as well as H_MSG_S) increase with the communication cost.



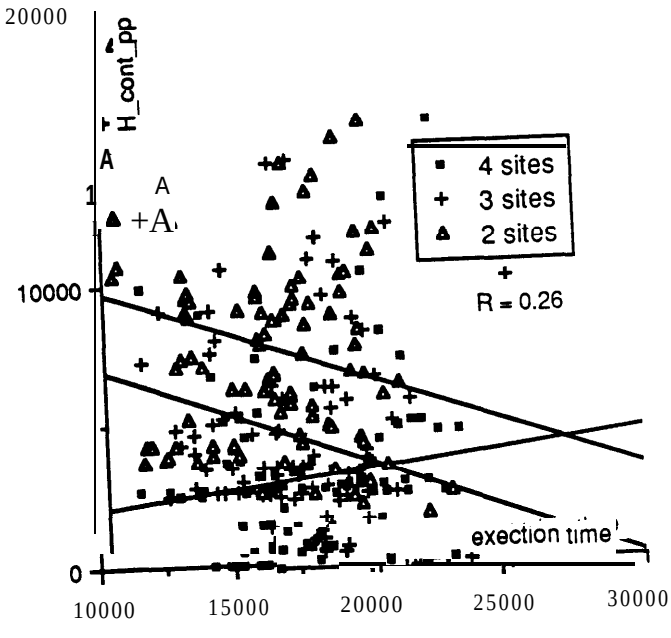
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-8. Correlating the Site which Spent the Most Time Routing Messages with Execution Time

4.6 "Contention"

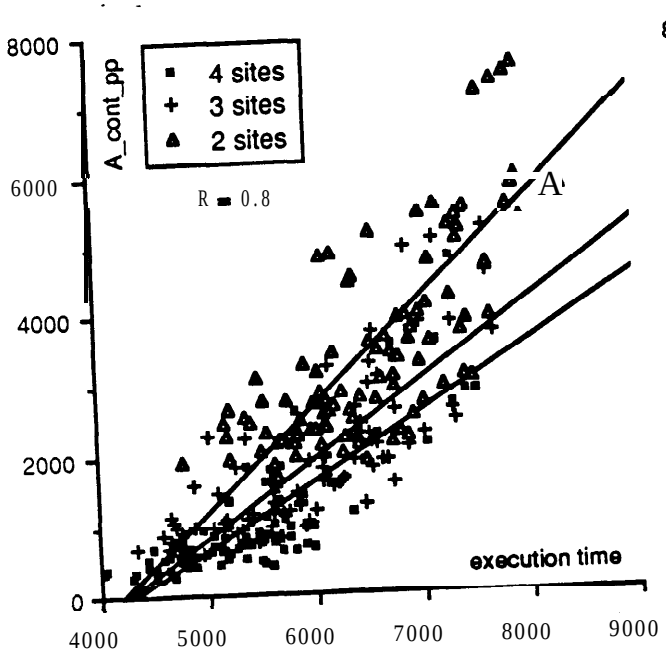


(a) T_{hop} = 10

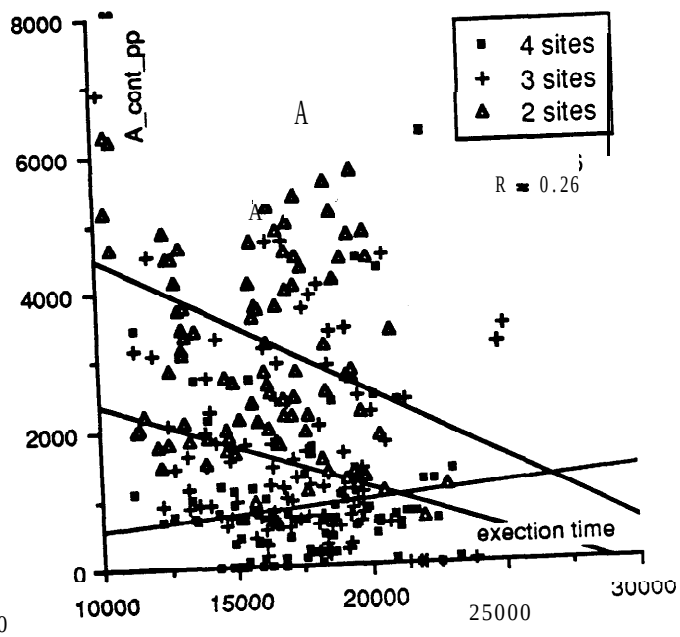


(b) T_{hop} = 100

Figure 4-9. Highest "Contention" Experienced between Two Players

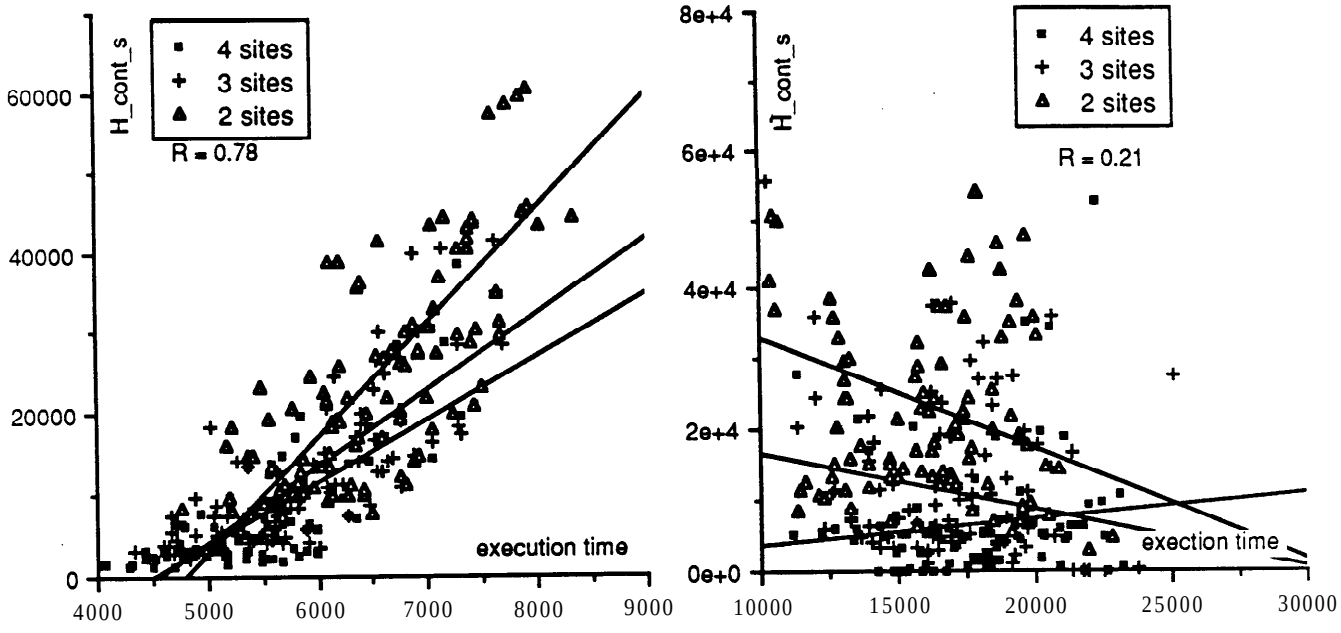


(a) T_{hop} = 10



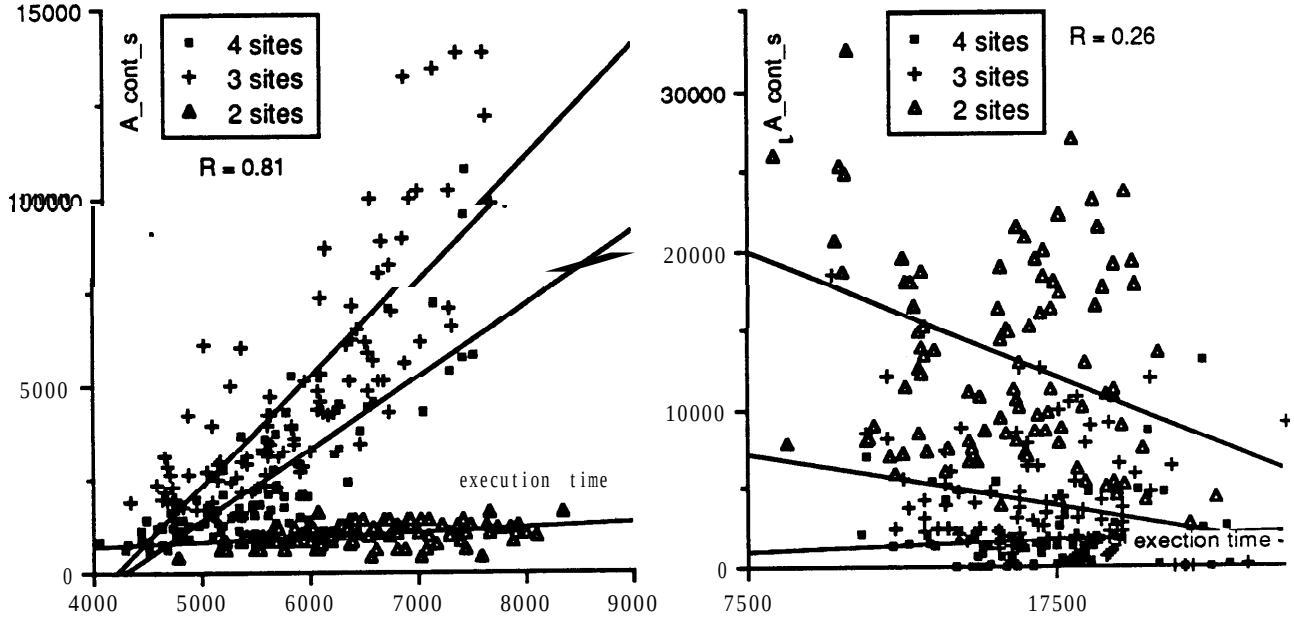
(b) T_{hop} = 100

Figure 4-10. Average "Contention" Experienced between Two Players.



(a) T_{hop} = 10 **(b) T_{hop} = 100**

Figure 4-11. Highest "Contention Sum" Experienced by Players at a Site.

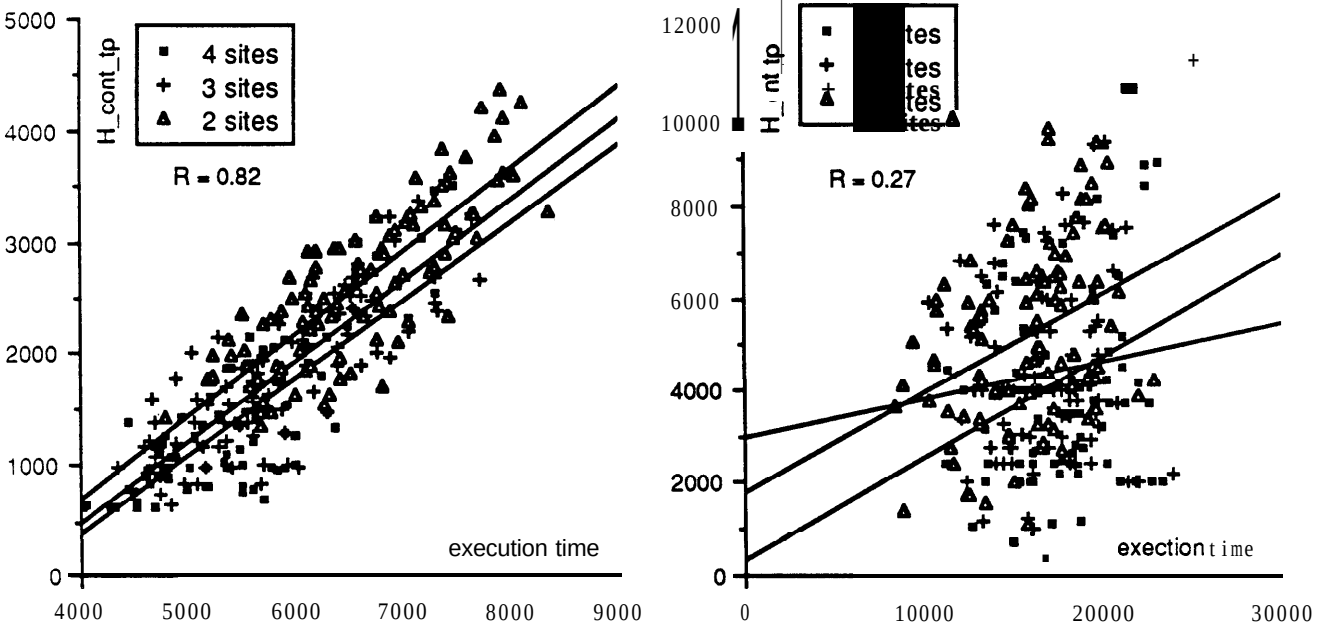


(a) T_{hop} = 10 **(b) T_{hop} = 100**

Figure 4-12. Average "Contention Sum" Experienced at a Site.

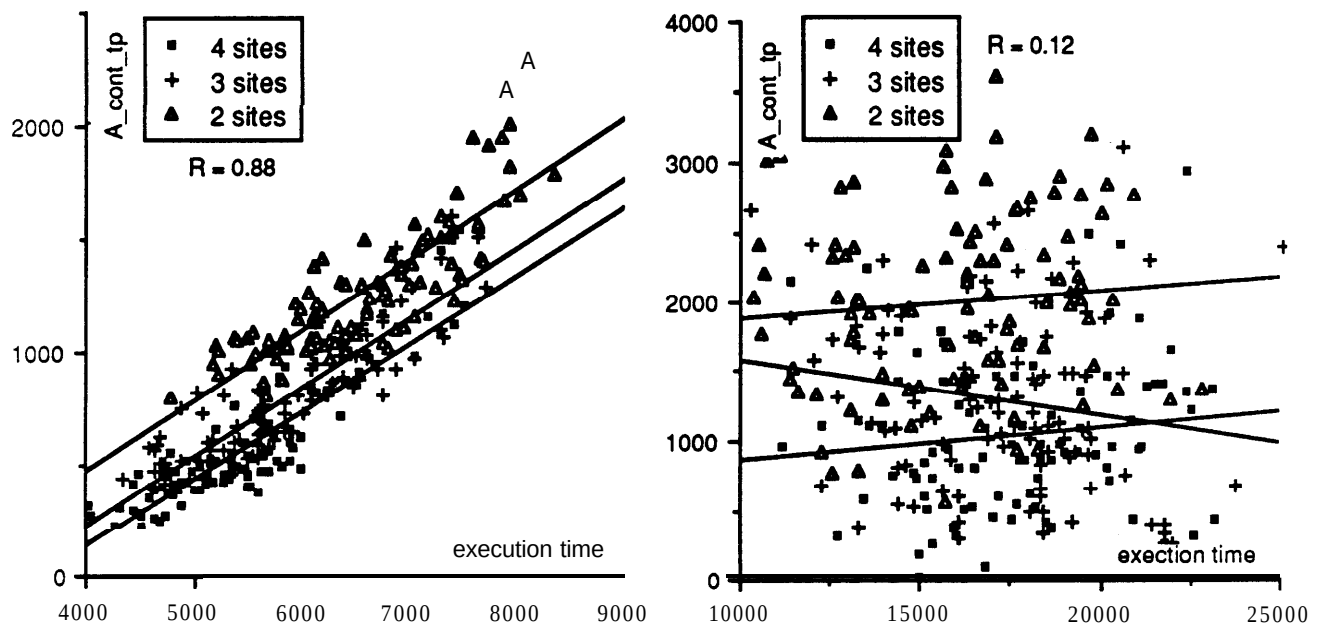
Using "sequential terminology", the poor correlation for all these (CPU) contention parameters seems to indicate that the system is "I/O bound". In other words, although

players within a site still competes for the processor, proper allocation of computing resources to facilitate message sending is more critical to obtaining speed-up!



(a) T_{hop} = 10 **(b) T_{hop} = 100**

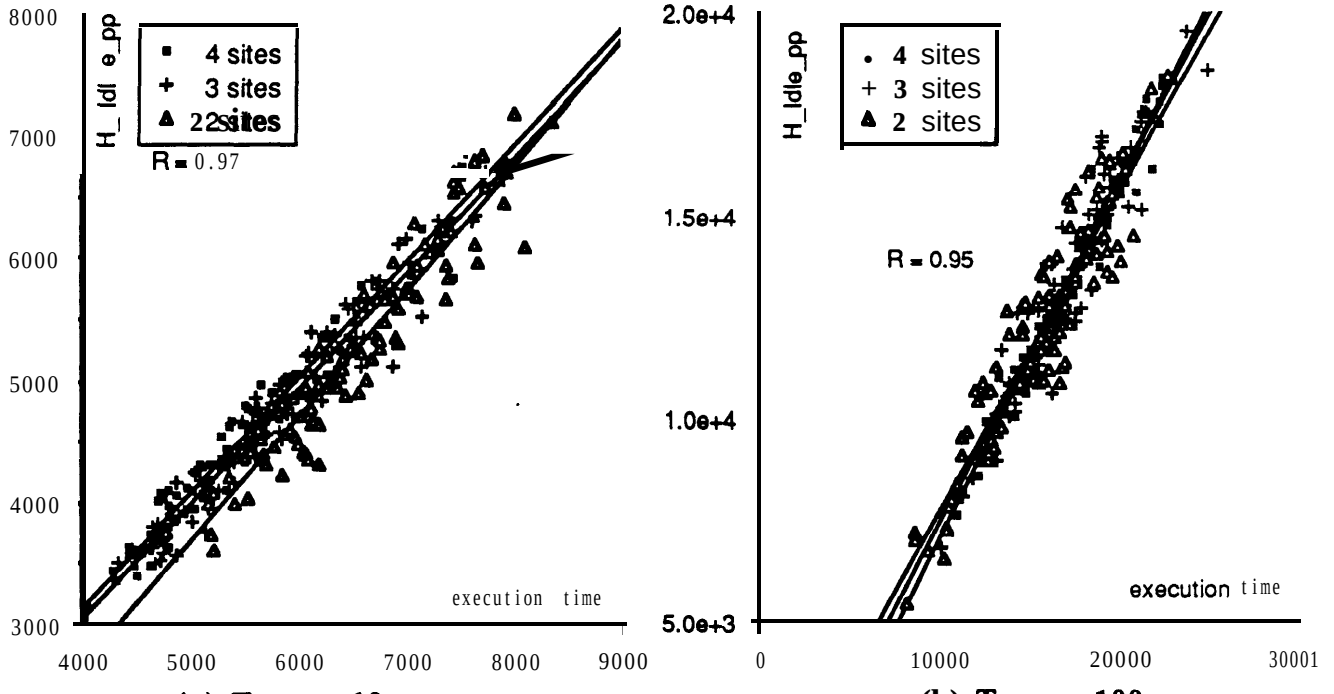
Figure 4-13. Longest CPU Wait Time.for a Player



(a) T_{hop} = 10 **(b) T_{hop} = 100**

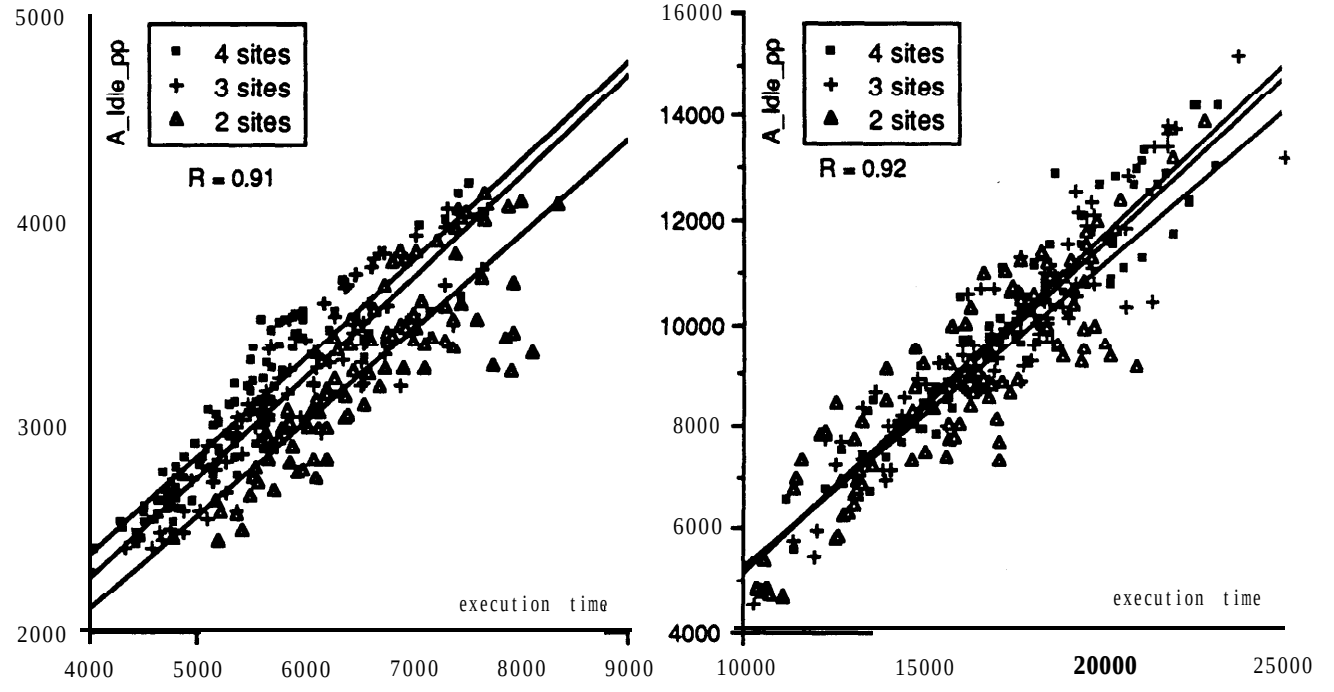
Figure 4-14. Average CPU Wait Time.per Player.

4.7 Dependencies



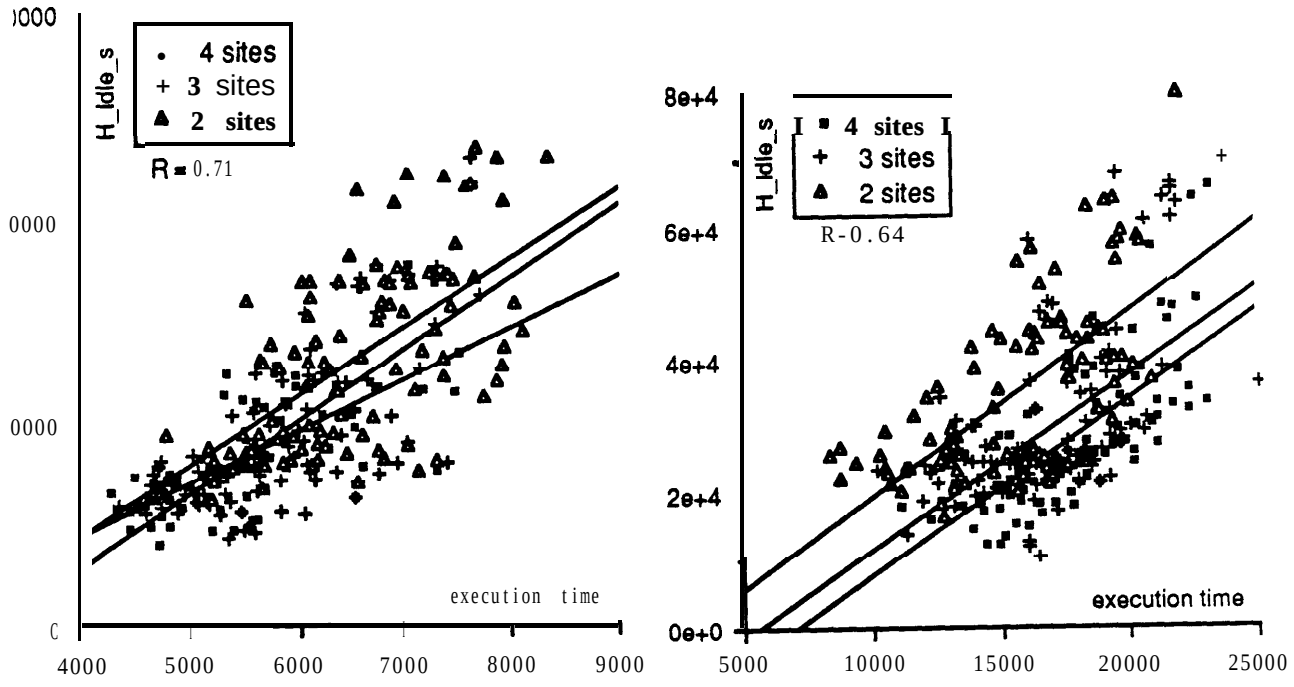
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-15. Longest Idle Time between a Pair of Players



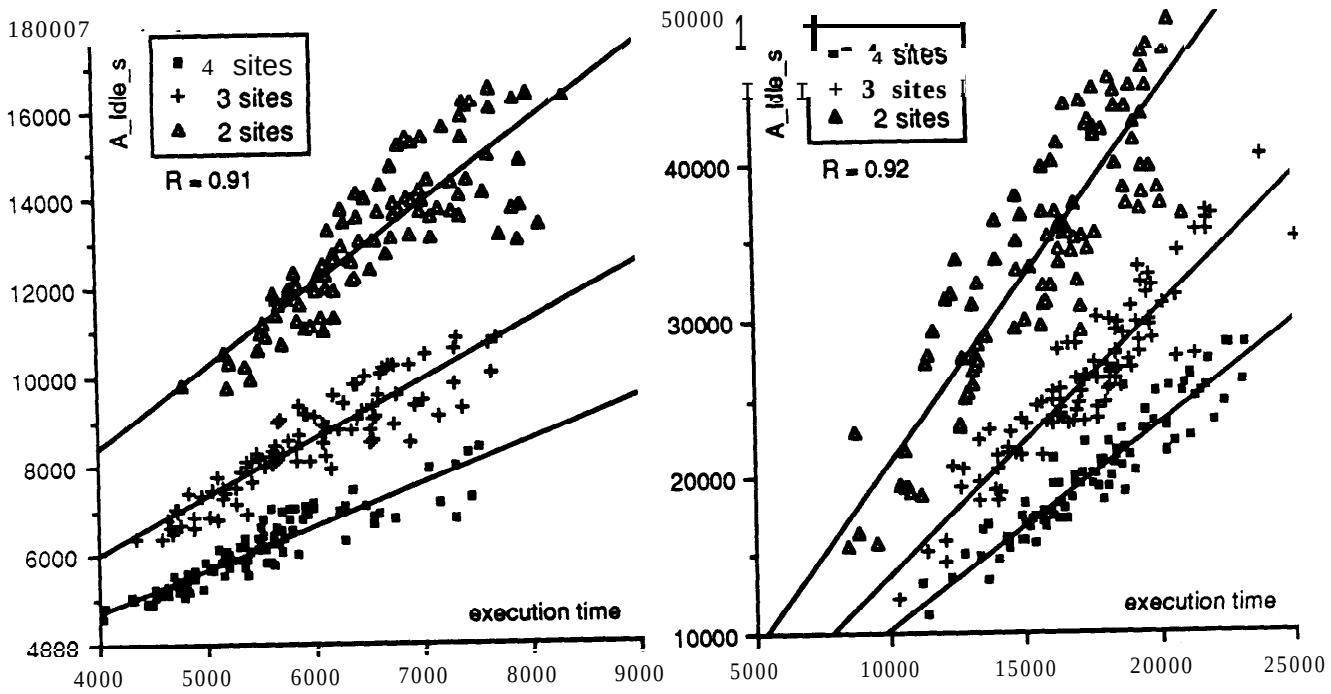
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-16. Average Idle Time between a Pair of Players



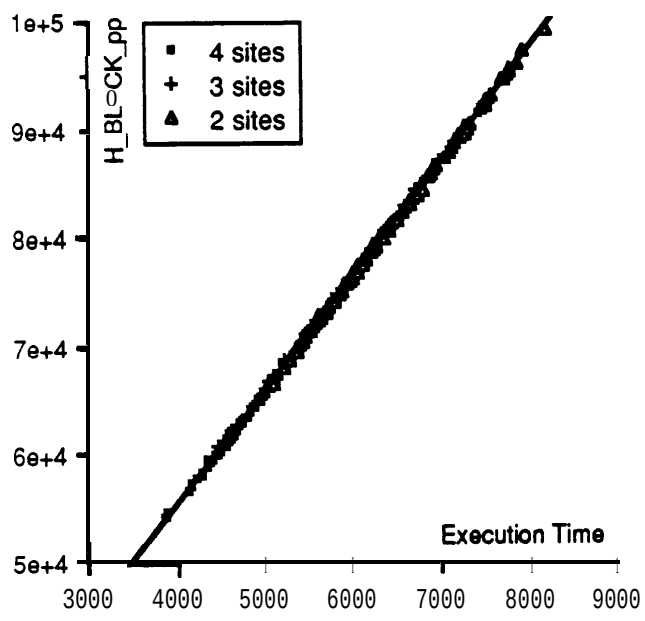
(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-17. Largest Sum of Idle Time for Players at a Site

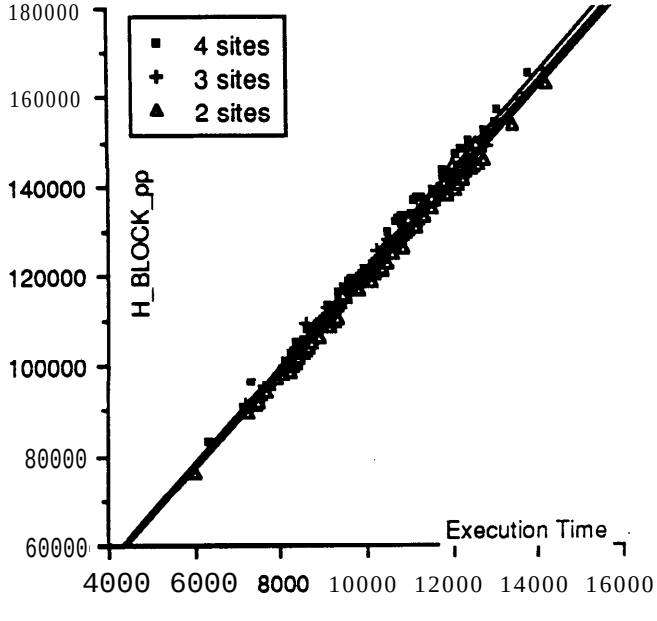


(a) T_{hop} = 10 (b) T_{hop} = 100

Figure 4-18. Average Sum of Idle Time for Players at a Site

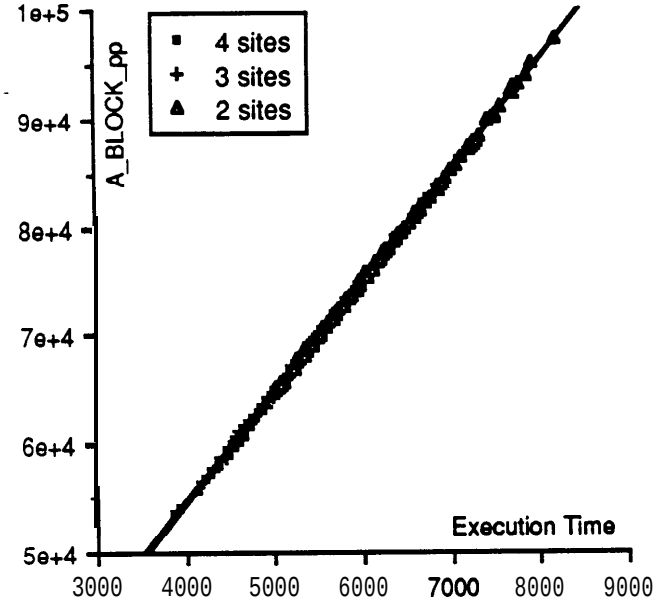


(a) T_{hop} = 10

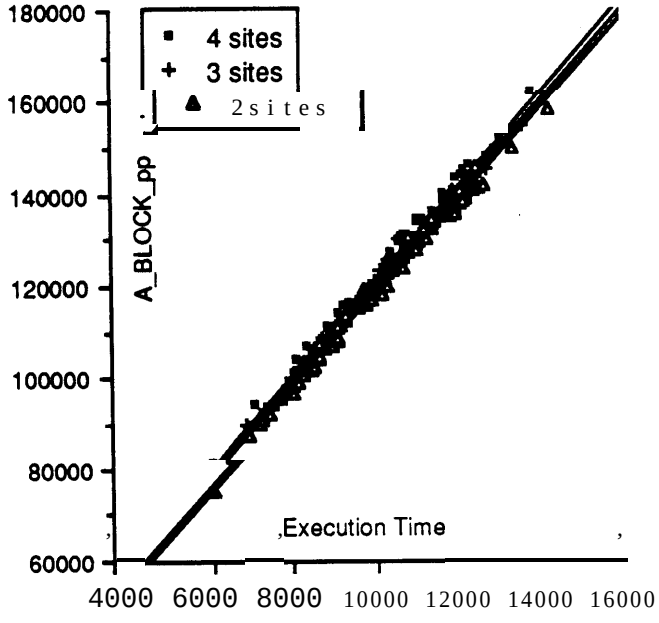


(b) T_{hop} = 50

Figure 4-19. Longest "Block Time" between Player Pairs



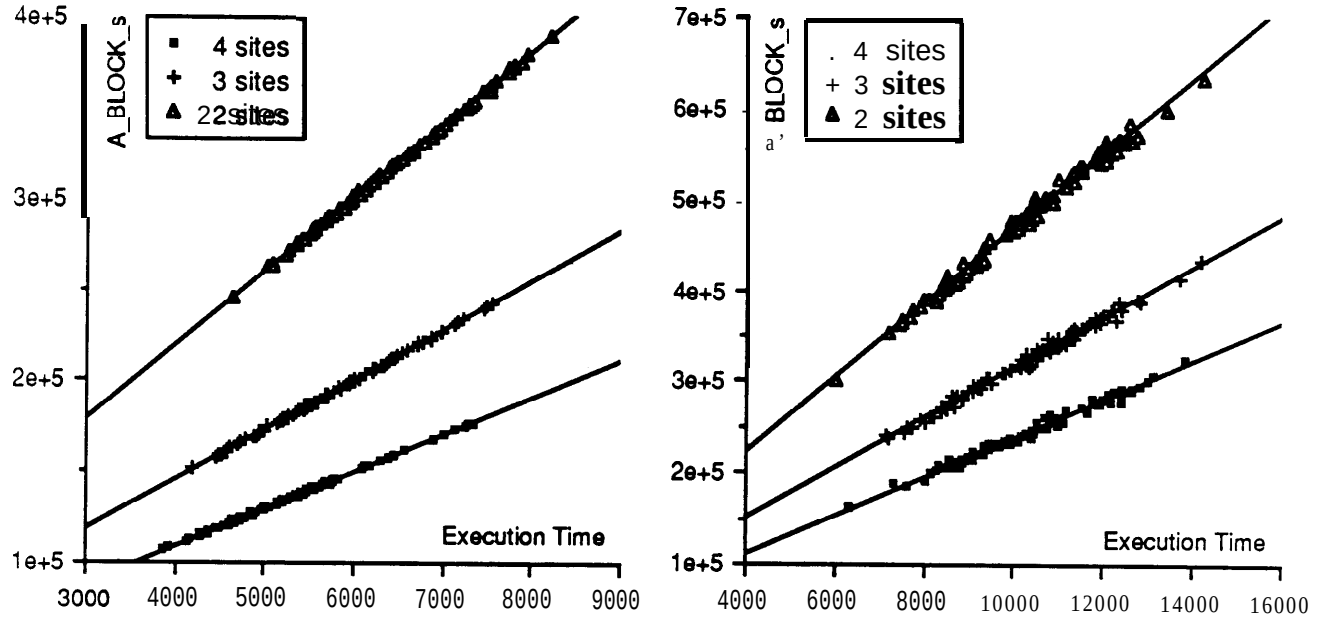
(a) T_{hop} = 10



(b) T_{hop} = 50

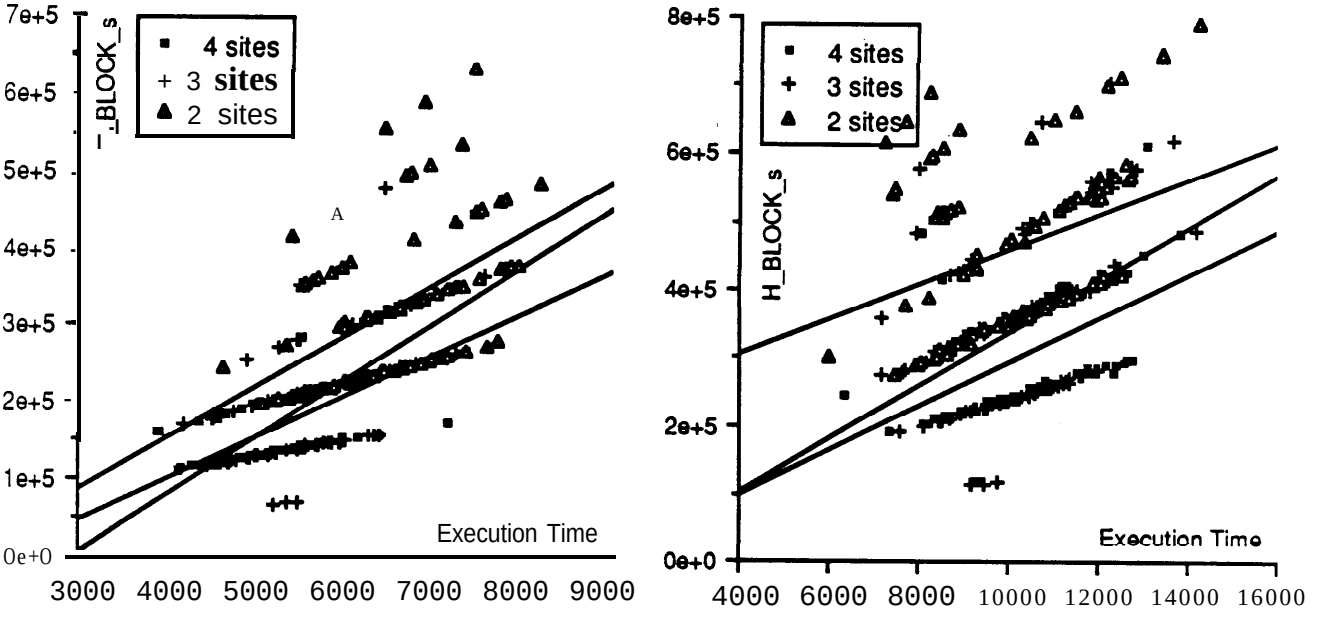
Figure 4-20. Average "Block Time" between Player Pairs

For this benchmark, the parameters A_BLOCK_{pp}, H_BLOCK_{pp}, A_BLOCK_{S}, and H_BLOCK_{S} are defined and plotted in Figures 4-19, 4-20, 4-21 and 4-22. Their high co-relation suggests that a mapping that minimizes the idle time between players also minimizes the execution time.



(a) T_{hop} = 10 (b) T_{hop} = 50

Figure 4-21. Average Sum of"Block Time" for Player per Site



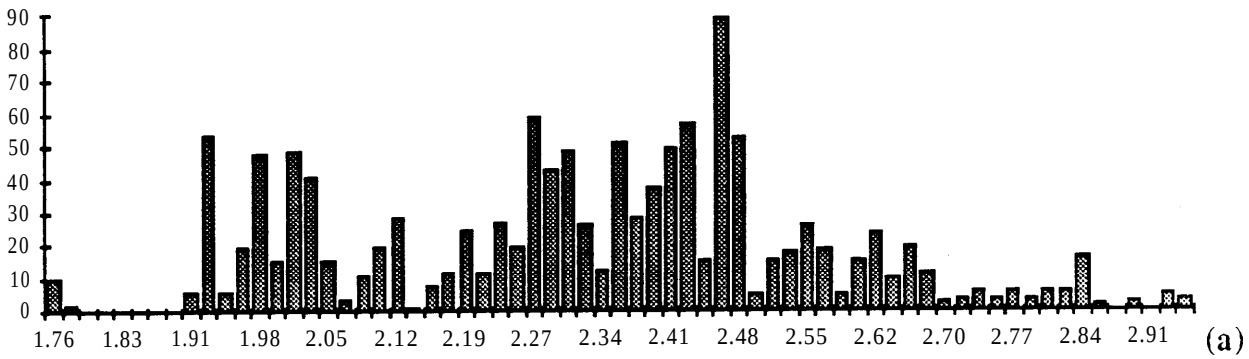
(a) T_{hop} = 10 (b) T_{hop} = 50

Figure 4-22. Highest Sum of"Block Time" for Player per Site

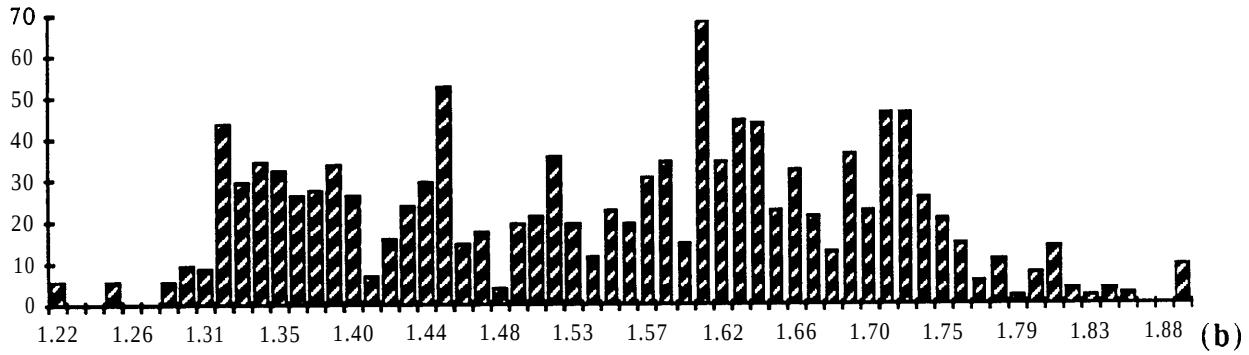
5. Conclusions and Future Directions

5.1 Summary of Major Findings and Their Interpretation

A. For both benchmarks, the speed-up obtained varies according to how players are mapped to sites. Randomly generated mappings can be far from the optimal, thus making the success of the resource management system critical to fully exploiting available parallelism in a concurrent system. Even when all sites are loaded with almost the same number of players (i.e. "balanced"), execution speed is not necessary minimized. Figure 45 below illustrates that when the sites are loaded in a 2-2-2-3 configuration[1], speed-up is still distributed over a very wide range irrespective of communication cost. The same is true for high communication cost.



"Pipeline" Benchmark



"Divide-and-Conquer" Benchmark

Figure 5-1. Speed-up Distribution with the 2-2-2-3 Configuration (T_{hop} = 10)

B. The following parameters are found to be "good descriptors" for the execution environment — these parameters correlates well with execution time:

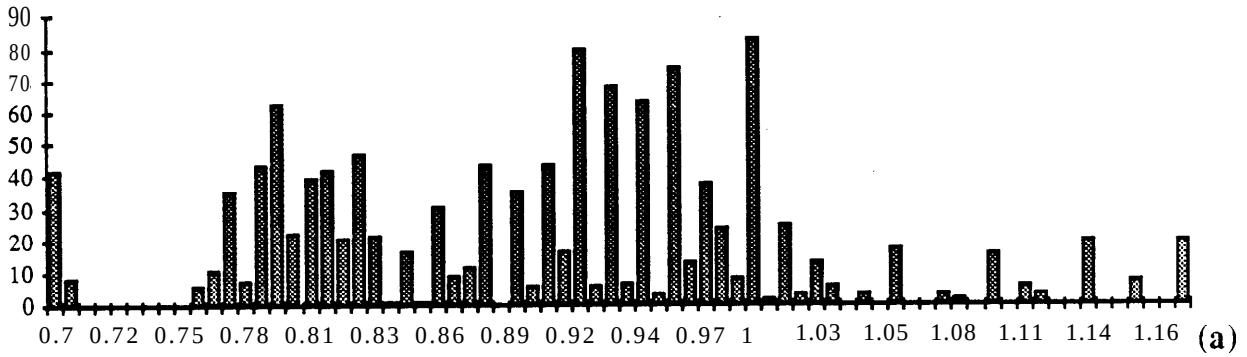
---

[1] A "w-x-y-z" configuration indicates that there are "w" players in the first site, "x" in the second... etc.

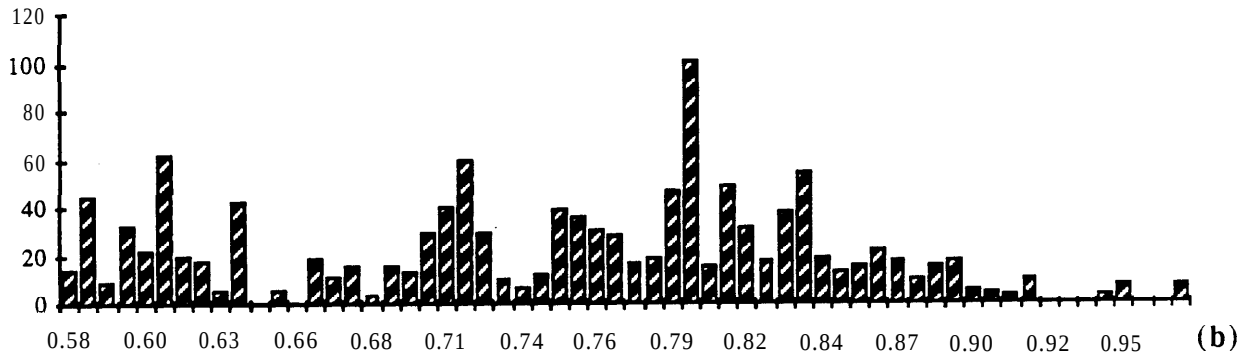
Benchmark I Pine-line):

With low/high communication costs

- H_CPU_S (Figure 3-6), the CPU utilization of the "busiest" site. This indicates that the processor at the "busiest" site is the bottle-neck for the execution environment.
- $H\_CONT\_RT_p$ (Figure 3-13) describes the player which spent the longest time in the ready-queue. There are three possible reasons that make ready-queue "long": (i) all resident players require a lot of processing, (ii) players at a site always execute at the same time and/or (iii) the CPU is busily delivering messages. In this benchmark, where players execute as soon as the CPU is available, the "busiest site" presents the longest ready-queue to its residents.



" Pipeline" Benchmark (T_{hop} = 100)



"Divide-and-Conquer" Benchmark (T_{hop} = 50)

Figure 5-2. Speed-up Distribution with the 2-2-2-3 Configuration

With low communication cost

. H_CONT_{pp} (Figure 3-9a) represents the highest contention experienced between a pair of players. When communication cost is negligible, the

processor is mostly used for program execution. Players, therefore, has to wait for the processor mostly because of other players executing. Therefore, this quantity reflects the **CPU** utilization at the busiest site. However, with high communication cost, the player has to wait for message delivering as well. H_CONT_{pp} therefore, does not necessarily reflect the bottle-neck of the system.

Benchmark II (Divide-and-Conauer):

With low/high communication costs

- H_IDLE_{pp} (Figure 4- 15) and H_BLOCK_{pp} (Figure 4- 19) represent the longest time a player has to wait for a request to arrive and finished respectively. Because this benchmark is highly structured, the performance of the system depends critically on the specific sequence in which players are invoked. These two quantities represent the 'bottle-neck" in the system
- . A_IDLE_{pp} (Figure 4-16) and A_BLOCK_{pp} (Figure 4-20) represent the average time a player has to wait for a request to arrive and finished respectively. The good correlation of these two parameters (with the execution time) merely reinforces the point above — that the performance of programs which exploit parallelism explicitly depends critically on the way in which it is orchestrated.
- A_IDLE_S (Figure 4-18) — the average sum of "idle time" per site

With high communication costs

- A_CPU_S (Figure 4-4b) and H_CPU_S (Figure 4-5b) describe the utilization of the processor at each site. The fact that they correlate well (with execution time) only at high communication costs again suggests that CPU usage is not a critical factor for execution with low communication cost. It is the sequence, not the amount, in which different players utilizes the **CPU** that matters (c.f. H_CONT_{pp} in Figure 4-9a).
- A_MSG_S (Figure 4-7b) and H_MSG_S (Figure 4-8b) represent the time spent delivering messages. At high communication cost, message routing takes a relatively larger proportion of time and thus CPU usage becomes a bottle-neck. These parameters reflect such bottle-necks.

With low communication costs

- $A\_CONT\_RT_p$ (Figure 4-14a) describes the average time a player has to wait in the "ready queue" before execution. The good correlation again indicates that it is not the amount of processing power available that matters but that

the processor should be available when a player requires it. The correlation deteriorates with increased communication cost because increased message deliver time have "skewed" the execution profile

C. The following parameters do not correlate well with program execution time for both benchmarks:

With low/high communication costs

- N_MSG (Figures 3-4 and 4-3) — the total number of messages routed to remote players does not describe the execution environment because message routine can occur in parallel. Although N_MSG describes the total emote of processor time spent for communication, minimizing this quantity without regard to contention of processor contention at each site does not necessarily guarantee minimization of execution time.
- A_CONT_S (Figures 3-12 and 4-12) — the average sum of "contention" per site — is not a good measurement of contention over the system.
- H_IDLE_S (Figures 3-17 and 4-17) describes the site with the highest sum of "idle time".

With low communication cost:

- A_CPU_S (Figure 3-5a and 4-4a) — the average CPU utilization per site

With high communication cost, the following does not represent the contention of the system:

- H_CONT_{pp} (Figure 3-9b and 4-9b) — highest "contention" between a player pair
- H_CONT_S (Figures 3-l 1 b and 4-l lb) — site with the highest contention experienced by players
- A_CONT_S (Figures 3-12b and 4-12b) — average contention experienced by players per site
- $A\_CONT\_RT_p$ (Figures 3-14b and 4-14b) — average time spent in the ready-queue per player

5.2 Performance of "Post-Game" on these Benchmarks

Execution time cannot be minimized simply by finding mappings with small numbers of remote messages. The effects of CPU contention at each site must also be taken into consideration. In order to improve execution time, "post-game" must:

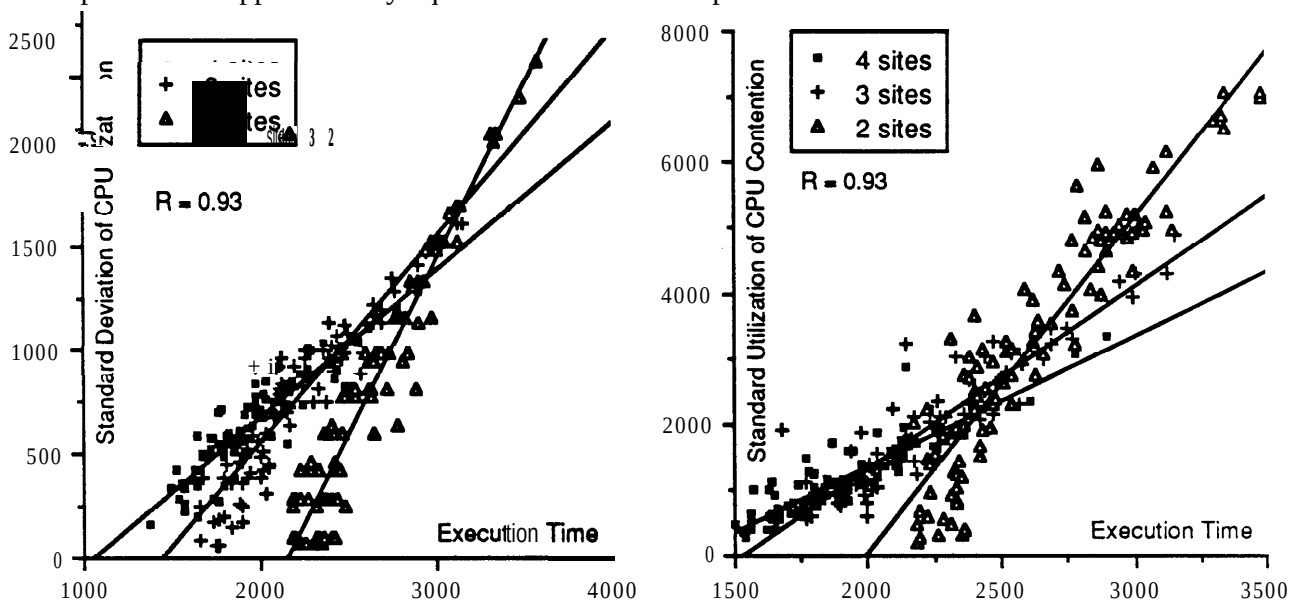
i) properly trade-off the gain obtained from concurrency with the increased cost in remote communication; and

ii) locate (and then, alleviated) the bottle-neck of the system.

Three "post-game" strategies are applied to these benchmarks for performance comparison and evaluation. The first two strategies "Site-Priority" and "Corn-gain" consist of a single heuristic each [Yan 87a]. The third strategy labeled "Post-game" consists consists of 7 heuristics — the application of which is carefully controlled and prioritized. A detailed description and explanation of the performance of the heuristics is given elsewhere [Yan 87b].

52.1 Benchmark I — Low Communication Cost

The first exercise involves managing the "Pipe-line" Benchmark with T_{hop} = 10. Figures 5-3a and 5-3b illustrate that the optimal placement involves a "balanced mapping" which presents an approximately equal demand on all the processors at each site.



(a) USR_{std} (b) $CONT_{std}$

Figure 5-3 Benchmark I at Low Communication Cost (T_{hop} = 10)

The performance of the heuristics is plotted against the distribution of speed-up to demonstrate their performance (Figure 5-4):

1. The maximum speed-up obtained is approximately the same for all three strategies — within 96% of the optimal (top 1% of all possible mappings).
2. The multi-heuristic-strategy (labeled "Post-Game") achieves this value in two (!) steps — while the others take approximately 10 iterations.

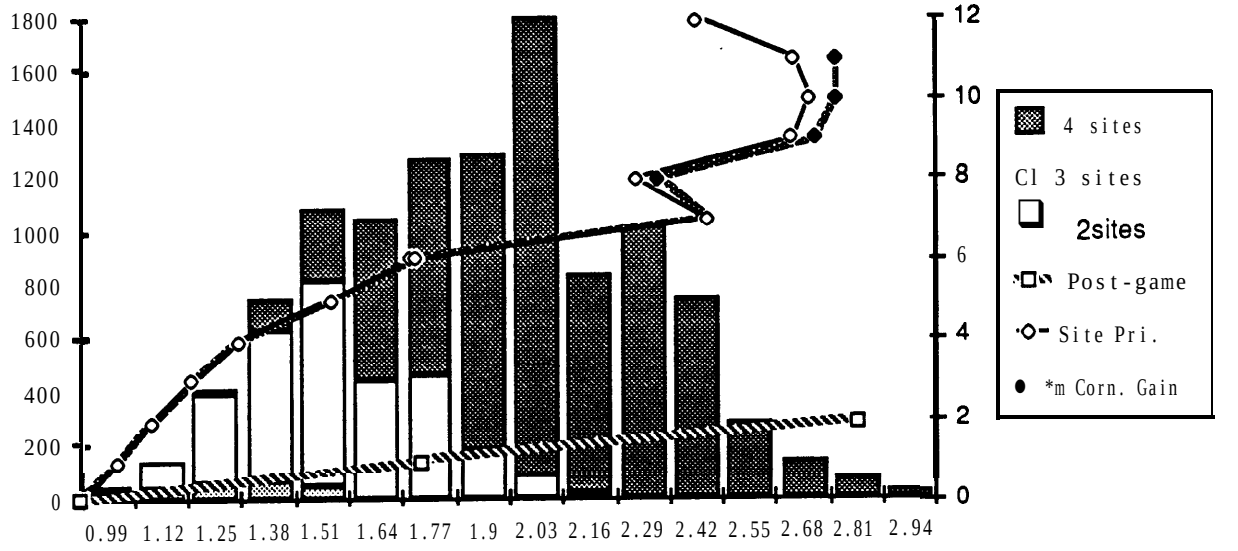
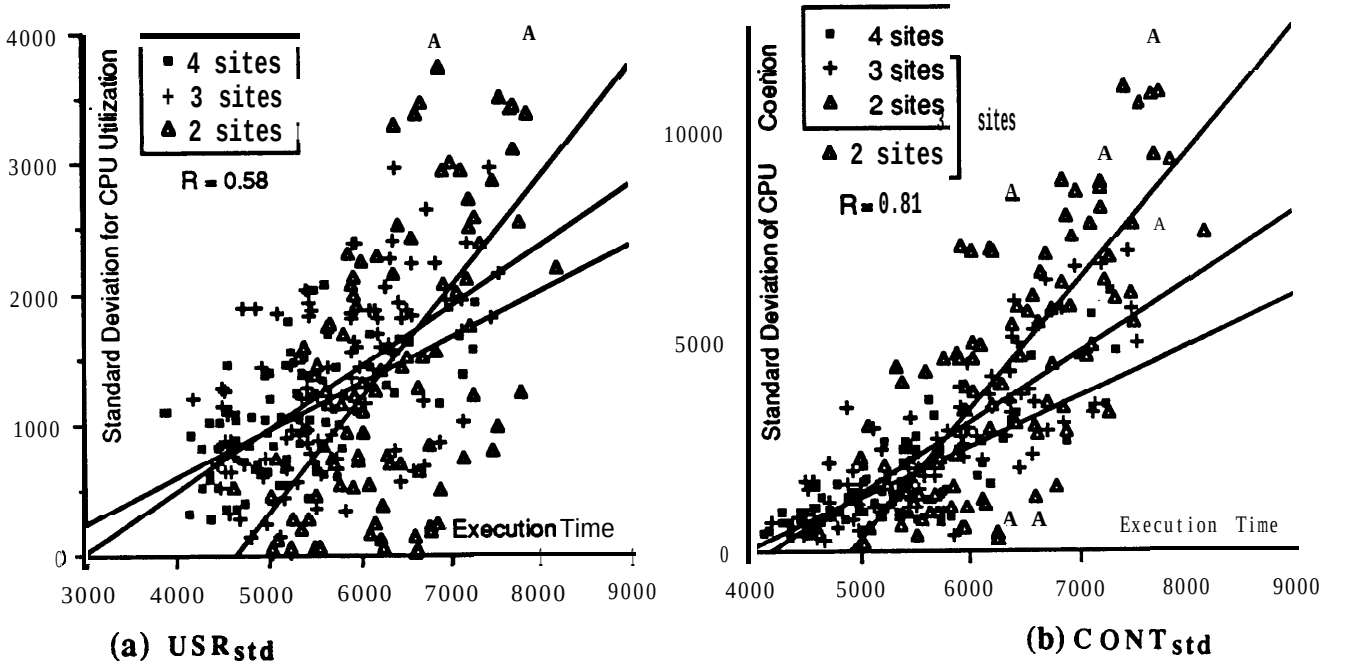


Figure 5-4 Performance of "Post-Game Analysis on Benchmark I with low Communication Cost (T_{hop} = 10)

5.2.2 Benchmark II — Low Communication Cost



(a) USR_{std} (b) $CONT_{std}$

Figure 5-5 Benchmark II at Low Communication Cost (T_{hop} = 10)

The placement exercise for the "Divide-and-Conquer" is a slightly different problem. Besides the evidences offered in section 4, Figures **5-5a** and **5-5b** further suggest that balancing "contention" as opposed to **CPU** utilization is the key towards obtaining speed up. Figure 5-6 shows that:

1. Again, the maximum speed-up obtained is approximately the same for all three strategies — within 96% of the optimal (top 1% of all possible mappings).
2. The multi-heuristic-strategy (labeled “Post-Game”) achieves this value in five steps — while the other two takes approximately 7 iterations.

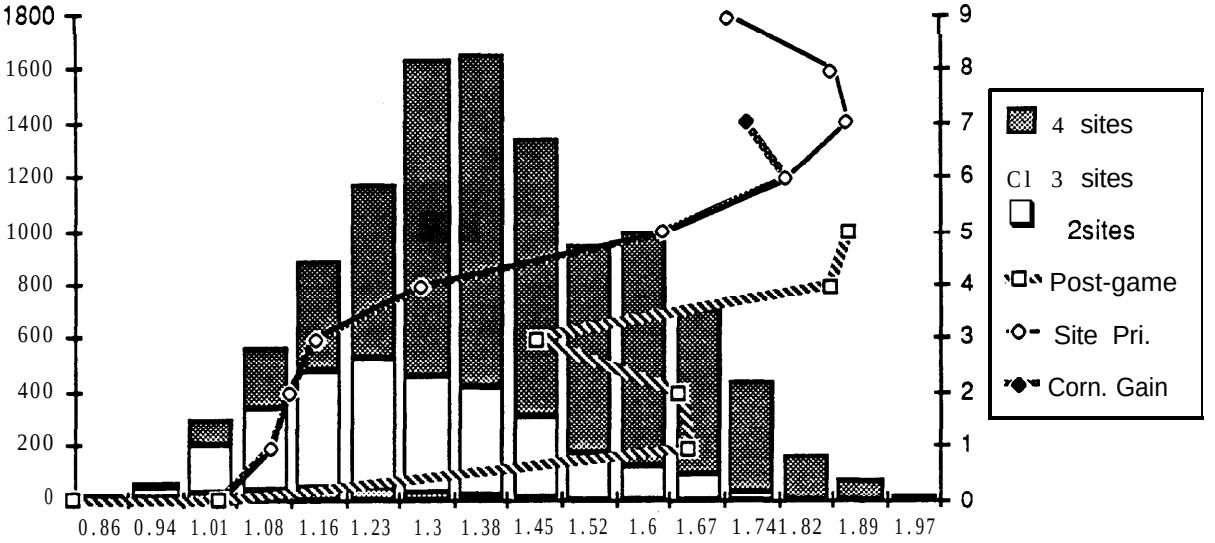
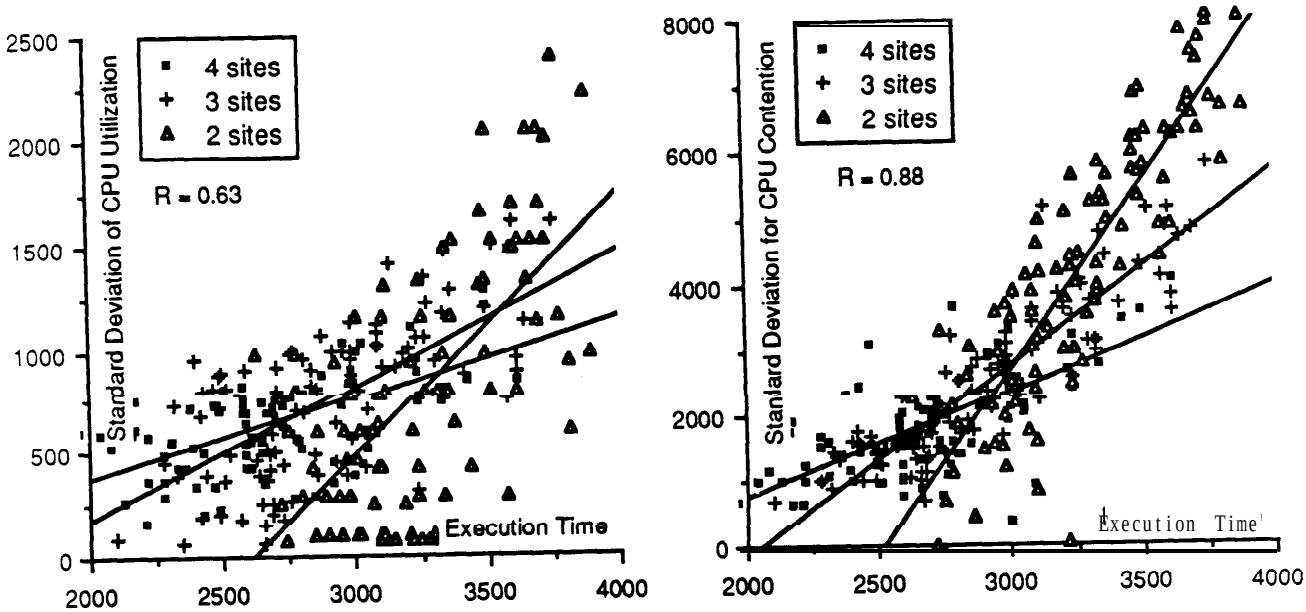


Figure 5-6 Performance of “Post-Game Analysis on Benchmark II with low Communication Cost (T_{hop} = 10)

5.2.3 Benchmark I — High Communication Cost

With higher communication cost, the placement problem becomes less interesting as the range of achievable speed-up decreases. The optimal speed-up attainable also decreases — in fact, with sufficiently high communication cost, mappings utilizing more than one site exhibit a longer execution time. Figures **5-7a** and **5-7b** further suggest even balancing “contention” does not guarantee obtaining the optimal speed-up in this case. Figure 5-8 shows that:

1. The maximum speed-up obtained is is obtained by the heuristic labeled "Com-gain” in 11 steps — within 95% of the optimal (top 1% of all possible mappings).
2. The multi-heuristic-strategy (labeled “Post-Game”) achieved only 85% of the optimal and halted after 3 iterations.



(a) USR_{std} (b) $CONT_{std}$

Figure 5-7 Benchmark I at High Communication Cost (T_{hop} = 100)

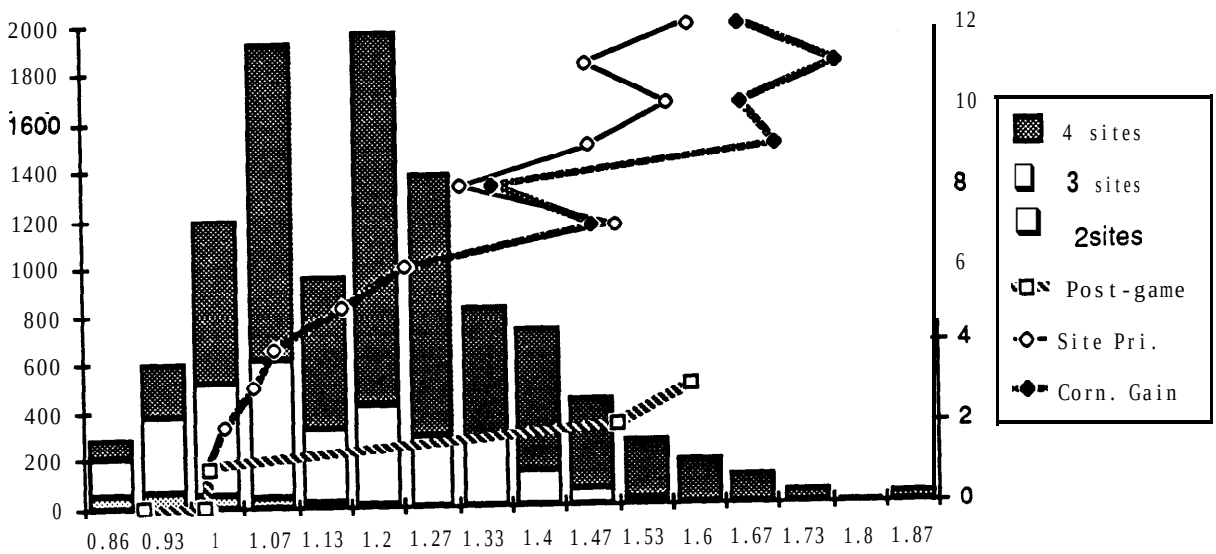


Figure 5-8 Performance of "Post-Game Analysis on Benchmark I with High Communication Cost (T_{hop} = 50)

5.2.4 Benchmark II — Moderate Communication Cost

It should be noted that:

i). obtaining the optimal speed-up has nothing to do with "load-balancing" (Figure 5-9); and

ii). the range of attainable speed-up is small — with half of which less than 1 (Figure 5-10).

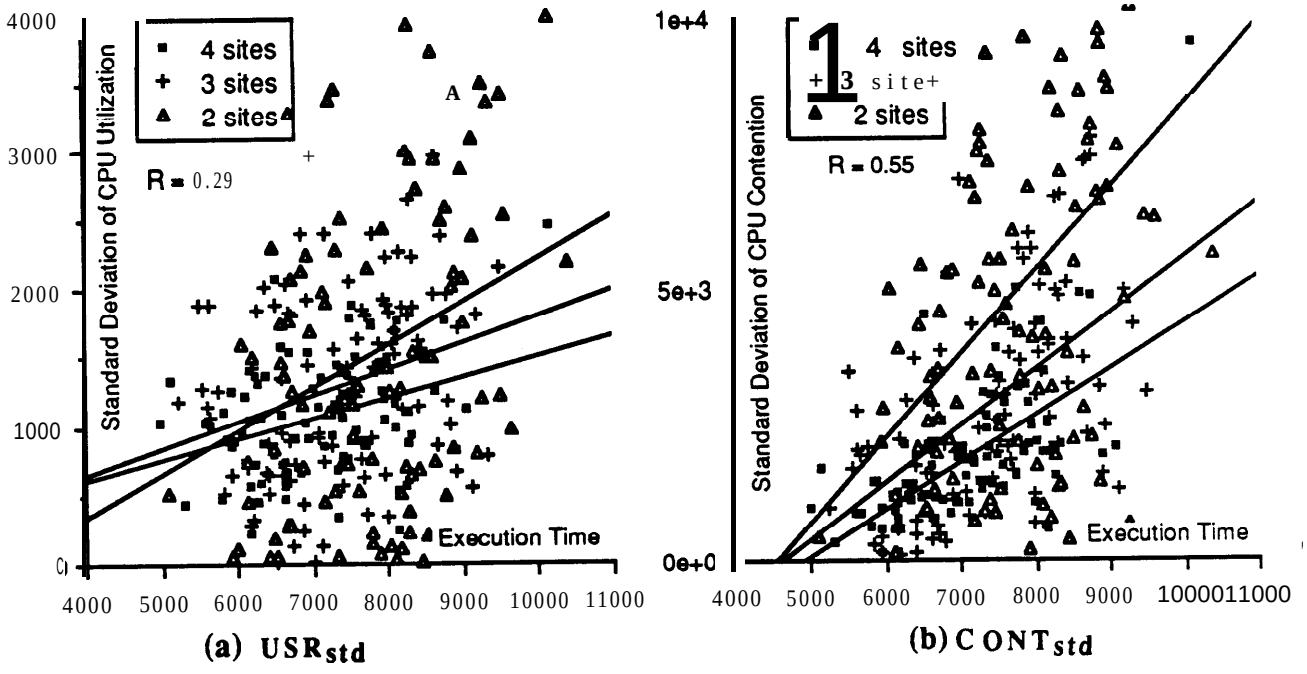


Figure 5-9 Benchmark II at Moderate Communication Cost (T_{hop} = 25)

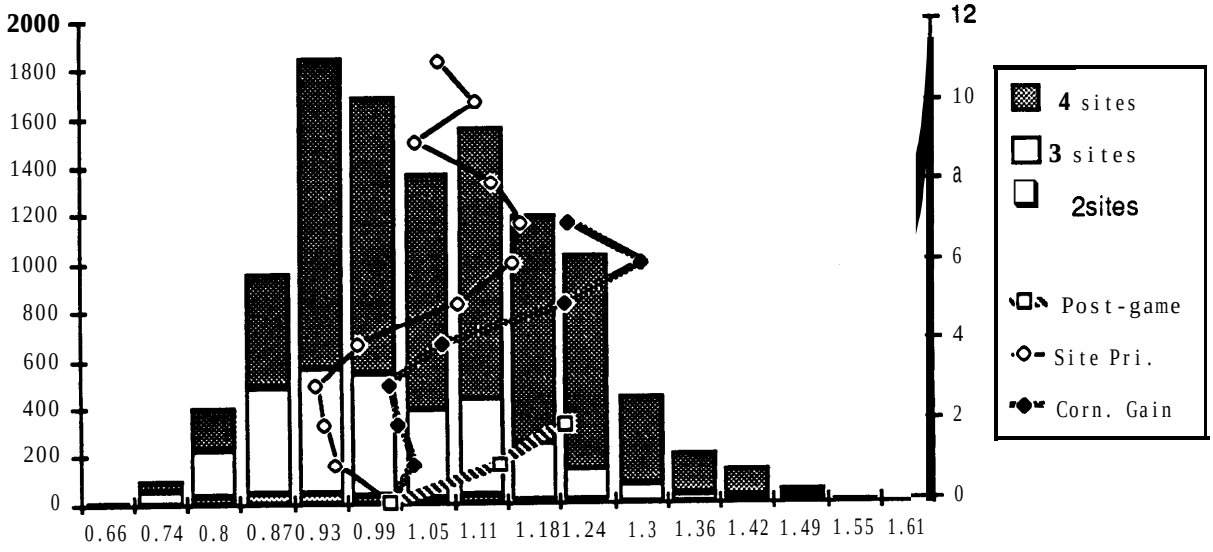


Figure 5-10 Performance of "Post-Game Analysis on Benchmark II with Moderate Communication Cost (T_{hop} = 25)

In spite of these difficulties, the placement heuristics were able to attain to 84% of the optimal speed-up achievable (top 25% of the whole population). Figure 5-10 shows that

for the two single-heuristic strategies, the initial steps in fact "slowed down" the execution time of the program. This, however, was soon corrected.

5.3 Conclusions and Future Directions

An investigation was conducted to find out how system parameters varies with mapping configurations for two small programs (with nine players) on a multiprocessor (with four sites). The sites of the multiprocessor were completely connected, nullifying any effect of routing strategies. The measurement of "locality" in this system involves only two discrete states: "local" or "remote" as opposed to some continuum for other connection topologies such as a "grid" or n-dimensional cube. Even for such a simple system, the range of attainable speed-up is large (a 3-fold difference!) with low communication cost. The two programs selected represented two different ways of exploiting parallelism: data-flow vs. parallel-procedure invocation, unstructured vs. structured exploitation of parallelism, implicit vs. explicit expression of concurrency and processing-bound vs. communication-bound characteristics. Initial experimental analysis of the data suggests that the key factors to obtain speed-up for both cases are similar:

1. Alleviation of bottle-neck — whether it be communication or processing contention, the bottle-neck of the system has to be detected and somehow, alleviated.
2. Minimization of remote communication or balancing the load of each site alone does not necessarily leads to a reduction in execution time. These two factors have to be properly traded off.

The experiment also demonstrate that a good resource management strategy should be able to respond to different programs behavior (or programming paradigms):

- With the first program "pipe-line", the key issue was reducing the processing bottle-neck in the system
- With the second program however, identifying the dependencies between players to allocate resources accordingly is more important

It was also shown that the placement heuristics proposed indeed was able to respond to the different needs of these two small programs and attain acceptable speed-ups. A detailed account of these heuristics is given elsewhere [Yan 87a, Yan 87b].

Regression analysis and related data-interpretation techniques are being employed in order to identify the cross-correlation between some of these variables and extract the few that are "really matters". The findings from these analysis are currently used to refine/improve the heuristics.

References

[Agha 85] Gul A Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", ***PhD Thesis,*** Massachusetts Institute of Technology, AI Lab., TR 844.

[Bokhari 77] S H Bokhari, "Dual Processor Scheduling with Dynamic Reassignment", ***IEEE Transactions on Software Engineering, July, 1979***

[Conway 63] M Conway, "A Multiprocessing System Design", *Proceedings of the* ***AFIPS Fall Joint Computer Conference, 1963.***

[Chow 79] Yuan-Chieh Chow, Walter Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System", *IEEE* ***Transactions on Computers,*** May, 1979

[Davis 82] A.L. Davis and R.M. Keller, "Data flow program graphs", *IEEE* ***Computer,*** February, 1982.

[Dennis 66] J B Dennis and E C Van Horn, "Programming Semantics for Multiprogrammed Computations", ***Communications*** *of the ACM,* March 1966.

[Dijkstra 65] E W Dijkstra, "Cooperating Sequential Processes", Tech. Report EWD-123 Technological University, Eindhoven, The Nethelands.

[Efe 82] Kemal Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems", ***IEEE Computer Magazine,*** June, 1982.

[Gao 84] Chuanshan Gao, Jane WS Liu, Malcolm Railey, "Load Balancing Algorithms in Homogeneous Distributed Systems", Dept. of Computer Science, University of Illinois, Technical Report UICDCS-R-84-1168.

[Gylys 76] V B Gylys, J A Edwards, "Optimal Partitioning of Workload for Distributed Systems", ***Proceedings*** *of* ***Compcon Fall, 1976.***

[Hanan 70] M Hanan and J M Kurtzberg, "Force-vector Placement Techniques," International Business Machines Cooperation, IBM Report RC 2843, April 1970.

[Hoare 78] C A R Hoare, "Communicating Sequential Processes", *Communications* *of* ***the ACM,*** August, 1978.

[Rung 82] H T Kung, "Why Systolic Architectures?", ***IEEE Computer*** *Magazine,* January, 1982.

[Lo 81] Virginia Mary Lo, Jane W S Liu, "Task Assignment in Distributed Multiprocessor Systems", ***IEEE Proceedings*** *of* ***the 1981 International Conference*** *of* ***Parallel Processing,*** August 198 1.

[Miller 85] Barton P Miller, "Parallelism in Distributed Programs: Measurement and Prediction", ***Computer Sciences Technical Report*** 574, University of Wisconsin-Madison, 1985.

[Nelson 81] B J Nelson, "Remote Procedure Call", Xerox Palo Alto Research Center, Palo Alto, California, Tech. Report CSL-81-9

[Ni 81] Lionel M Ni, Kai Hwang, "Optimal Load Balancing Strategies for a Multiple Processor System", ***IEEE Proceedings of the 1981 International Conference of Parallel Processing,*** August 198 1.

[Sankar 87] Vivek Sarkar, "Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors", PhD Thesis, Department of Electrical Engineering, Stanford University.

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

[Stankovic 81] John Stankovic, "The Analysis of a Decentralized Control Algorithm for Job Scheduling Utilizing Bayesian Decision ***Theory", IEEE Proceedings*** *of* ***the 1981 International Conference of Parallel Processing,*** August 1981.

[Steele 85] Craig Steele, "Placement of Communicating Processes on Multiprocessor Networks", MS Thesis, Department of Computer Science, California Institute of Technology, Technical Report 5184:TR:85

[Stone 77] Harold Stone, "Multiprocessor Scheduling with the Aid of Network ***Flow*** Algorithms", ***IEEE Transaction on Software Engineering,*** January 1977.

[Stone 78] Harold Stone, "Control of Distributed ***Processes", Computer Magazine,*** July 1978.

[Su 85] W-K Su et al, "C Programmers' Guide to the Cosmic Cube", California Institute of Technology, M.S. Thesis (CITCS) 5 129:TR:84.

[Weng 75] K-S Weng, "Stream-Oriented Computation in Data Plow Schemas", TM 68, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1975.

[Yan86a] J.C. Yan, "Parallel Program Behavior Specification and Abstraction using BDL", Computer System Laboratory, Stanford University, CSL-TR-86-298

[Yan86b] J.C. Yan and S.F. Lundstrom, "AXE: A simulation environment for actor-like computations on ensemble architectures," *Proc. **I986 Winter Simulation Conference,*** Washington DC.

[Yan87a] J.C. Yan, "Post-Game Analysis — An Initial Experiment for Heuristic-Based Resource -Management in Concurrent Systems", Computer System Laboratory, Stanford University, CSL-TR-87-3 14.

[Yan87b] J.C. Yan, "Multi-Perspective Performance Evaluation and Prediction of Concurrent Execution Environments", Computer System Laboratory, Stanford University, to be published.

Appendix

I. Solving for "All" Possible Assignments for Completely Connected Sites

The problem of generating all possible assignments for completely connected sites turned out to be more difficult than expected. If all the sites were different or are connected in an irregular topology, the number of possible assignments is large ($N\_S^{N\_P}$)— but generating the placement configurations is straight forward! When the sites are connected in a regular topology, many of the $N\_S^{N\_P}$ mappings are in fact identical. They are "mirror images" or can be obtained via similar transformations from some other mappings. Completely connected nets present a easier placement problem to be solved among the many regular topologies.

The problem of deciding "how many ways there are to place N_P players into **N-S** completely connected site" is broken into two major sub-problems -'both of which can be solved by recursive methods.

A. Classification of Placement Configurations.

Although **N-S** sites are available, it does not necessarily follows that there is at least one player in each site for each of the possible configuration. So if we define:

§ 1. $A_{N\_S}^{N\_P}$ = the number of ways to place **N-P** players into **N-S** completely connected sites (such that **N-P ≥ N-S), then**

§ 2. $A_{N\_S}^{N\_P} = \sum_{i=1}^{i=N\_S} B_{i}^{N\_P}$ ..... where

§ 3. $B_{i}^{N\_P}$ = the number of ways to place **N-P** players into exactly "i" completely connected sites. In other words,. $B_{i}^{N\_P}$ describes the number of ways to place **N-P** players into "i" completely sites such that there is at least one player in each site.

§ 4. Each mapping that utilizes exactly "i" sites can be described by a set $\beta_{i,N\_P} = \{n_1, n_2, \ldots n_i\}$ where "n_k" describe the number of players in the "k^{th}" site (e.g. when **N-P** = 6 and i = 3, the possible β_{i,N_P}'s include: {1,1,4}, {1,2,3} and {2,2,2}). It should be noted that

a. since all sites are homogeneous and are connected to one another, the order of the elements "n_k" does not matter (hence β_{i,N_P} is a set); and

b. $\sum_{k=1}^{k=i} n_k =$ **N-P** for all $\beta_{i,N}$ p

§ 5. The sets $\beta_{i,N\ P}$ can be further classified into three kinds:

a. simple-p: where all "n_k"s are different;

b. even-p: where all "n_k"s are the same; and

c. normal-~: otherwise (only some "n_k"s are identical)

The method of obtaining $B_i^{N\_P}$ for all three cases are similar but not identical

B. Generating Placement Configurations.

A simple recursive algorithm can be used to generate β_{i,N_P} for any "i". The total number of possibilities can be enumerated by further classifying the β's into the three classes shown in §5 above. The difficult part of the problem is not finding a method to "count" but to "generate" all placement configurations without dup ication. In combinatorial theory, many enumeration techniques first generate all possible configurations in a similar situation and then divide the result obtained with an appropriate value argued from symmetry[1]. With millions of possibilities, an algorithm that needs to check each configuration with the ones already generated is not practical.

This section describes the recursive algorithms used to

i) generate all possible β_{i,N_P}'s

ii) generate all possible placement configuration for simple-P's, even-β's and normal-β's.

B.1 The Algorithm to Generate β_{i,N_P}

For any set β_{i,N_P}, a corresponding ordered i-tuple (or vector) β'_{i,N_P} can be obtained by sorting the "n_k"s in an increasing order. "**generate**β(i, 1, N-P)" can be called to obtain the "n_k"s of the vector $\beta'_{i,N\ P}$ (where β'[k] represents the k^{th} element of the i-tuple). The simplified procedure is listed as follows following in a "C"-like syntax.

```
generateβ(k, min-val, players-left)        /* The procedure takes 3 arguments */
  int k,                                   /* points to (i-k)th element of array */
      min-val,                             /* β'[k]≤β'[k+1] ● */
```

[1] e.g. the total number of ways to choose "r" balls out of "n" $C_r^n = \frac{n!}{(r!)(n-r)!}$ can be thought of

i) place "n" balls into "n" slots (n!);
ii) then merely look at the first "r". slots (hence divide by (n - r)!); and finally
iii) ignore the ordering of the first "r" slots (hence divide by r!)

```
    players-left;                                  /* since Σ (k=1 to k=i) nk = N-P */

{ int result, flag;                                /* internal variables'/
  flag = 0;                                        /* initialization ● /
  if (players-left < min_val) return (0) ; /* since β'[k] ≤ β'[k+1], this call*/
                                                   /* call to assign fails ● /
  else {                                           /* otherwise...*/
    if (k == 1) {                                  /* if this is the last (k^th) element of β' ● /
      β'[i - 1] = players-left;                    /* put all the players left into β'[i-1]*/
    configureβ(β')                                 /* record configuration and generate*/
                                                   /* the actual placement configuration ● /
    return(l);                                     /* a successful configuration is found*/
    } else do {                                    /* if this is not the (k^th) element of β' */
      β'[i - k] = min_val;                         /* assign the minimum value to β'[i-k]*/
      result =                                     /* recurse down the array... ● /
        generateβ(k-1,                             /* assign the next element... ● /
                min_val,                           /* with the same minimum value... ● /
                players-left-min-val);             /* but with fewer players left */
      if (result) flag = 1;                        /* if any successful configuration was ● /
      ++min_val;                                   /* found , increment value of β'[i-k]...*/
    } while (result););                            /* and try again until no successful ● /
                                                   /* mapping can be located */
    return(flag);}                                 /* tell caller whether any successful */
                                                   /* configuration was ever found ● /
```

B.2 The Algorithm to Configure β_{i,N_P}

Generating the actual mapping between players and sites is simplest when all the "n_k"s of $\beta'_{i,N\ P}$ are different from another. This problem is analogous to enumerating the sequential selection of balls from a pool of N-P with n_k balls taken at a time (in any order). For example, for $\beta_{3,10} = \{2,3,5\}$, the total number of possible placements = $C_2^{10} * C_3^8 * C_5^5 = C_5^{10} * C_2^5 * C_3^3 \ldots$ where C_x^y = the number of ways to select a subset of "x" items from a pool of "y". However, when all "n_k"s are identical, this simple enumeration approach has to be slightly modified. The interested ready is encouraged to prove the following statements:

§ 6. The total number of ways to configure simple-$\beta_{i,N\ p}$ = {no, $n_1, n_2, \ldots n_i$} is:

$$C^{N\_P}_{n_0} * C^{N\_P-n_0}_{n_1} * C^{N\_P-(n_0+n_1)}_{n_2} \ldots * C^{N\_P-(n_0+n_1)}_{n_{i-1}} * C^{n_i}_{n_i}$$

§ 7. The total number of ways to configure even-$\beta_{i,N\ p}$ = {n, n, n, . . . n} is:

$$C^{N\_P-1}_{n-1} * C^{N\_P-(n+1)-1}_{n-1} * C^{N\_P-2(n+1)-1}_{n-1} \ldots * C^{n-1}_{n-1}$$

Because the enumeration methods for even-p is different from that for simple-β, the process of configuring normal-P's (e.g. $\beta_{7,30}$ = {2,3,3,5,5,5,7}) involves two steps:

i) Sectioning β' into even-β's with the smallest possible length (in this case for $\beta_{7,30}$: {2}, {3,3}, {5,5,5}, and (7)) and then

ii) sequentially generating the configurations of these even-β's.

C. Distribution of the Number of Configurations

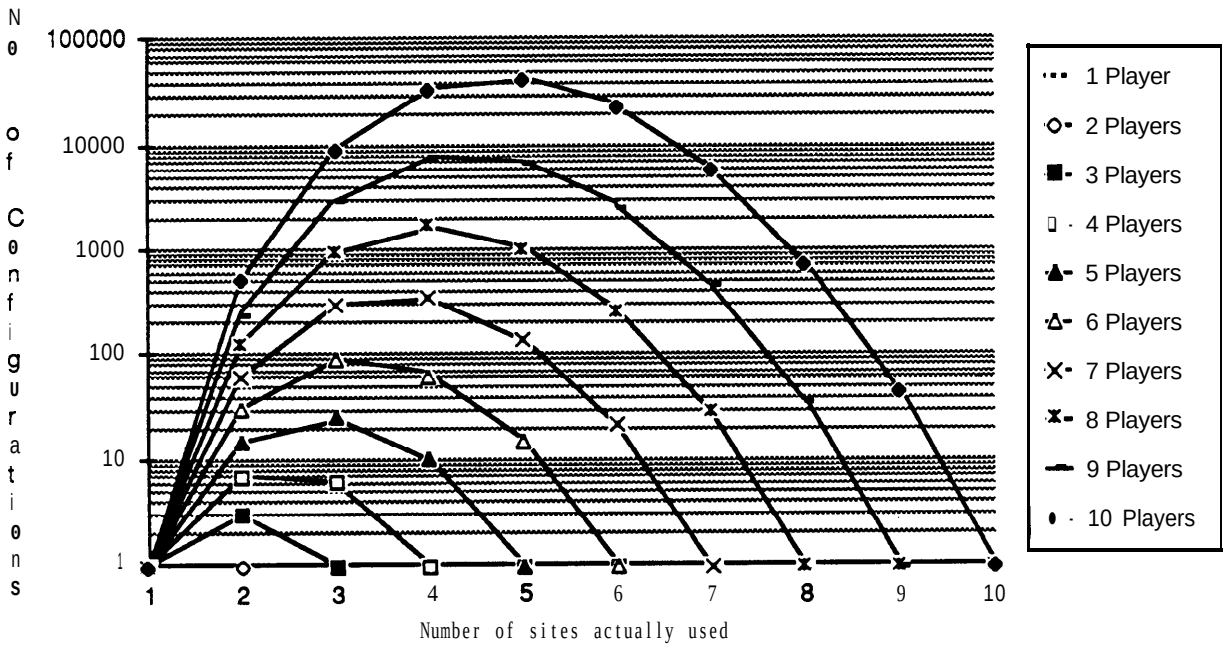


Figure A-l Distribution of the No. of Configurations against the No. of Sites Actually Used

It has already been shown in Figure 2-4 that the total number of possible configurations increases exponentially when an infinite number of sites is available. Figure A-l below illustrates how the total number of configuration varies against the number of sites actually used. It can be seen that the number of possibilities is the greatest when N_P ≈ 2 * N_S.

This explains another reason for choosing the 9-player-4-site configuration — to make it harder for the placement heuristics to locate a suitable mapping!

D Actual Program Listing

```
#include <stdio.h>
/* #define    G 'generate data structures to give correct output */
/*  #define   P  output placement configuration on screen */
/* #define    Q output player id's on screen */
/* Allowable Configurations G-P-Q-, G+P-Q+, G+P+Q- and G+P+Q+ */
/* The output for the various options are listed as follows for N-S = 2,
N_P = 5
```

G-P-Q- merely counts the number of possible configurations for different β

```
##type: 14          /* i.e. 1 player in a site and 4 in the other */
## 5 ##
##type: 23          /* i.e. 2 player in a site and 3 in the other */
## 10 ##
### Total(5, 2): 15 ###
```

G+P-Q+ generate the id's of the players that actually reside in each site

```
##type: 14
0123 4             /* player <4> in one site and the rest in the other site..*/
0124 3 0134 2 0234 1 1234 0
## 5 ##            /* total count*/
##type: 23
012 34             /* players <3>,<4> in one site and the rest in other site..*/
013 24 014 23 023 14 024 13
034 12 123 04 124 03 134 02 234 01
## 10 ##
### Total(5, 2): 15 ###
```

G+P+Q- generates the placement locations (i.e. site id's) for players

```
##type: 14
1 1 1 1 0          /* player <4> in site "0" and the others in site "1"
1 1 1 0 1          1 1 0  1 1
1 0 1 1 1          0 1 1 1 1
## 5 ##
##type: 23
1 1 1 0  0         1 1 0 1 0          1 1 0 0 1
1 0 1 1 0          1 0 1 0 1          1 0  0 1 1
0 1 1 1 0          0 1 1 0 1          0 1 0 1 1
0 0 1 1 1
## 10 ##
```

G+P+Q+ generates "everything"

```
##type: 14
0123 4 ** 1 1 1 1 0 ** 0124 3 ** 1 1 1 0 1 **
0134 2 ** 1 1 0 1 1 ** 0234 1 ** 1 0 1 1 1 **
1234 0 ** 0 1 1 1 1 **
```

```
        ## 5 ##
        ##type: 23
        012 34 ** 1 1 1 0 0 ** 013 24 ** 1 1 0 1 0 **
        014 23 ** 1 1 0 0 1 ** 023 14 ** 1 0 1 1 0 **
        024 13 ** 1 0 1 0 1 **
        034 12 ** 1 0 0 1 1 ** 123 04 ** 0 1 1 1 0 **
        124 03 ** 0 1 1 0 1 ** 134 02 ** 0 1 0 1 1 **
        234 01 ** 0 0 1 1 1 **
        ## 10 ##
        ### Total(5, 2): 15 ###
 */

/* --------------------- Cosmetic adjustments ---------------------- */
#ifdef  Q
#define  COLUMN  5
#else
#define  COLUMN  1
#endif

/* ---------------------------  Problem Size ---------------------- */
#define  DIM  15     /* maximum size of N-P */
#define  VDIM  10          /* maximum no. of sections in each β */

/* ------------------------ Useful Macros ------------------------ */
#define BU(Z2,Z1) for (i = 0; i < DIM; i++) Z1[i] = Z2[i]
#define ml          for (i-0; i < rr; i++) dd[i+used] = bb[used+D_aray[i]]
#define  m2         for (i = 0; i < rr + used; i++) bb[i] = dd[i]
#define  ddAbb      ml; m2; fill_up(N_P, used + rr, bb)
#define  m3         for (i=0; i < rr; i ++) D[i + used] = B[used+D aray[i]]
#define  m4         for (i = 0; i < rr + used; i ++) B[i] = D[i]
#define  ddAbb2     m3; m4; fill_up(n_p, used + rr, B)

/* ----------------------- Global Variables ------------------------ */
int AA[VDIM][DIM], BB[VDIM][DIM],   /* used for solving config2β */
    CC[VDIM][DIM], DD[VDIM][DIM];
int new_A[DIM], A[DIM], B[DIM],           /* used for solving configβ */
    C[DIM], D[DIM];
int E[DIM], F[DIM];
int stack[VDIM][2];                       /* used for sectioning each β */
int n_p, r, n_s;
long int total, sub_total;
main()                                /* main program */
{
  printf("input n_p, n_s\n");
```

```
scanf("%d %d", &n_p, &n_s);                /* first input N-P and N_s*/
sub_total = total = 0;                     /* Initializations */
total = sub_total = 0;
generateβ(n_s, 1, n_p);                    /* generate the β's */
if (sub-total > (long) 0) printf("## %ld ##", sub_total);
total += sub_total;
printf("\n### Total(%d, %d): %ld ###", n_p, n_s, total);


generateβ(dim, start, left)
int dim, start, left;
{
  int i, result, flag;
  flag = 0; if (left < start) return(0);
  else {
    if (dim == 1) {
      if (sub-total > 0) printf("## %ld ##", sub_total);
      total += sub_total; sub_total = (long) 0;
      A[n_s - dim] = left;
      printf("\n##type: "); display(stdout, n_s,A); printf("\n");
      new_vals();
    for (i = 0; i < DIM; i++) B[i] = i;
    gen(r, C, 0);
    return(l);
    } else do {
      A[n_s - dim] = start; result = generateβ(dim-1, start, left-start);
      if (result) flag = 1; ++start;
    } while (result);
  };
  return(flag);


gen(a_pos, ar, used)
int apos, ar[], used;

  int i, num, count, ss_pos;

  ss_pos = a_pos;
  num = new_A[--ss_pos];
  configureβ(0, 0, n_p - used, num, ar, a_pos-1, used);
}

configureβ(start_val, start_pos, nn, rr, D_aray, apos, used)
```

```
int start-t-al, start_pos, nn, rr, D_aray[], a_pos, used;
{
  int i, num, b[DIM];

  if (start_pos == rr - 1) {
    while (start_val < nn) {
      D_aray[start_pos] = start_val++;
#ifdef G
      BU(B,b); ddAbb2;
#endif
      if (a_pos) {
        gen(a_pos, &D_aray[rr], used + new_A[a_pos]);
      } else {
        if (n-s == r) report(D, new_A, r);
        else new-section(0);
        };
#ifdef G
      BU(b,B);
#endif
    },
    return(l);
    } else while (nn > start-val) {
    D_aray[start_pos] = start_val++;
#ifdef G
    BU(B,b);
#endif
    configureβ(start_val, startpos + 1, nn, rr, D_aray, apos, used);
#ifdef G
    BU(b,B);
#endif
  };
}

new_vals()
{
  int i, old, new;

  old = new = -1; r = 0;
  for (i = 0; i < n-3; i++) {
    new = A[i];
    if (old == new) {
      stack[r-1][1]++; stack[r-1][0] += new;
      new_A[r-1] += new;
    } else {
      stack[r][1] = 1; stack[r][0] = new;
```

```
      new-A[r++] = new;
    };
    old = new;
  };


fill-up (nn, rr, array)                    /* fill up rest of the array */
int nn, rr, array[];                       /* with no.s not already there */

  int i, index;
  index = rr;
  for (i = 0; i < nn; i++) if (absent(i, array, rr)) array [index++] = i;
}

absent(element, array, rr)                 /* check whether a particular value */
int element, array[], rr;                  /* is missing in the "array" */
{
  int i;
  for (i = 0; i < rr; i++) if (element == array[i]) return(0); return(l);
}

report (ar, in, the-r)                     /* display the particular configuration */
int ar[], in[];
{
  int i, j, u;
  sub-total++;                             /* increment no. of configurations found */
#ifdef G
#ifdef Q
  u = 0;   if (1 != sub_total%COLUMN) printf(" ");
  for(i = the-r-l; i >= 0; i--) {
    if (i != the-r-l) printf(" ");
    display(stdout, in[i], &ar[u]); u +=in[i];
  }
#endif
#ifdef P
#ifdef Q
  printf(" ** ");
#endif
  u = 0 ;
  for(i = the-r-l; i >= 0; i--) {
    for(j = 0; j < in[i]; j++) F[ar[j+ u]] = i;
    u +=in[i];
  };
  for(j = 0; j < n_p; j++) printf("%d ", F[j]);
#ifdef Q
  printf("**");
```

```
#endif
#endif
#ifdef Q
  if (0 == sub-total%COLUMN) printf("\n");
#else
  printf("\n");
#endif
#endif


display(where, end, array)          /* Low level Display routine */
int end, array[];
FILE *where;
{ int i; for (i = 0; i < end; i++) fprintf(where, "%d", array[i]);}

new_section(level)
int level;

  int i, size, N-P, R;
  N_P = stack[level] [O]; R = stack[level][1];
  size = N-P/R;
  for (i = 0; i < R; i++) AA[level][i] = size;
  for (i = 0; i < DIM; i++) BB[level] [i] = i;
  if (N-P == R) {
  for (i = 0; i < DIM; i++) DD[level][i] = i;
    if (level == r-l) xreport(); else new_section(level+1);
  } else {
    gen3(R, CC[level], 0, AA[level], BB[level],. DD[level], level, N-P, R);
  };
}

gen3(a_pos, ar, used, aa, bb, dd, level, N-P, R)
int apos, ar[], used, aa[], bb[], dd[], level, N-P, R;
{
  int i, num;
  num = aa[--a_pos];
  configure2β(0, 0, N-P - used, num, ar, apos,
              used, aa, bb, dd, level, N-P, R);


configure2β(start_val, start_pos, nn, rr, D_aray, apos,
              used, aa, bb, dd, level, N-P, R)
int start-val, startpos, nn, rr, D_aray[], apos,
              used, aa[], bb[], dd[], level, N-P, R;
```

```
 int i, num, b[DIM];

 if (start_pos == rr - 1) {
   while (start-val < nn) {
     D_aray[start_pos] = start-val++;
#ifdef G
     BU(bb,b); ddAbb;
#endif
     if (a_pos) {
       gen3(a_pos, &D_aray[rr], used + aa[a_pos],aa,bb,dd,level,N_P, R);
     } else {
       if (level == r-l) xreport(); else new_section(level+1);
       };
#ifdef G
     BU(b,bb);
#endif
    };
    return(l);
    } else while (nn > start-val) {
      if (start_pos == 0) {
        D_aray[start_pos++] = start-valtt;
  if (start_pos == rr - 1) {
    while (start-val < nn) {
      D_aray[start_pos] = start-valtt;
#ifdef G
     BU(bb,b); ddAbb;
#endif
     if (a_pos) {
       gen3(a_pos, &D_aray[rr], used t aa[a_pos],aa,bb,dd,level,N_P,R);
     } else {
    if (level == r-l) xreport(); else new_section(level+1);
        };
#ifdef G
     BU(b,bb);
#endif
    };
    return(l);
    }      };
      D_aray[start_pos] = start-valtt;
#ifdef G
     BU(bb,b);
#endif
     configure2β(start_val, startpos t 1, nn, rr,
                 D_aray, apos, used, aa, bb, dd, level, N-P, R);
#ifdef G
```

```
      BU(b,bb);
#endif
  };
}

xreport()
{
  int i, u, lev, j;
#ifdef G
  lev = u = 0;
  for (i = 0; i < r; it+) {
    for (j = 0; j < stack[lev][0]; j++) E[j+u] = D[DD[lev][j] + u];
    u+= stack[lev++][0];
  };
#endif
  report(E, A, n_s);
}
```

II. Listing of Benchmark I: Pipe-line

The first benchmark is made up of three player types:

"Stage" defines the operation of players <2> to <8>. They have different service times for three request types. After processing each request, they propagate the request to the next stage. It should be noted that requests are routed to the two acquaintances "downstream" alternately.

```
(DefPlayer stage                                  A"pipe-line stage"
  (MES_A MES_B MES_C I-D I_S)                     It understands 5 message types
  (nxt_stg_1 nxt_stg_2 nxt-stg)                   acquaintances "downstream"
  (dur-a durb dur-c)                              Service times for different requests
  (I-D (record dur-a dur-b dur_c))                hit: record different service times
  (I_S (record nxt_stg_1 nxt_stg_2)               Init: record alternative acquaintances
       (setq nxt-stg nxt_stg_1))                  Set acquaintance to receive next
  (MES_A                                          Request type "MES_A" received
    (run dur-a)                                   Execute for that amount of time
    (post nxt_stg MESA)                           Pass same type of request to next
  stage
    (if (= nxt_stg_1 nxt-stg)                     Set other acquaintance as next
  receiver
      (setq nxt-stg nxt_stg_2)
      (setq nxt-stg nxt_stg_1)))
  (MES_B (run dur_b) (post nxt_stg MES_B)         Same for "MES_B"
    (if (= nxt_stg_1 nxt-stg)
      (setq nxt-stg nxt_stg_2)
      (setq nxt-stg nxt_stg_1)))
  (MES_C (run dur-c) (post nxt_stg MES_C)         Same for "MES_C"
    (if (= nxt_stg_1 nxt-stg)
      (setq nxt-stg nxt_stg_2)
      (setq nxt-stg nxt_stg_1))))
```

"Last-stage" defines the operation of player <9>. It's main function is to terminate simulation after all requests are received and processed.

```
(DefPlayer last_stage                             The last "pipe-line stage"
  (MES_A MES_B MES_C INIT)
```

```
(dur-a dur-b dur-c my-count)
0
( INIT (record dur-a dur-b dur-c my-count)        Record service times and total
msg. count
   (repeat my-Count                               Receive all requests generated at first
     (wait MES_A) (run dur-a)                       stage
     (wait MES_B) (run dur-b)
     (wait MES_C) (run dur-c))
     (terminate)))                                Terminate simulation/execution
```

"First-stage" (i.e. Player **<1>**) carries out the initialization procedures (such as setting up the configuration of the pipe-line) and generate the appropriate number/type of requests.

```
(DefPlayer first_stage                            The first stage (Player <1>)
  (MES_A MES_B MES_C INIT...)
  (da db dc...)
  (... t2 t3...)
  ...
  (INIT
    ...
    (repeat my-Count                              Generate a number of requests
      (run da) (post t2 MES_A)                    and propagate them "downstream"
      (run db) (post t2 MES_B)
      (run dc) (post t3 MES_C))))
```

III. Listing of Benchmark II: Divide-and-conquer

The second benchmark is also made up of three player types:

"Root" (i.e. Player <1>) is the "root" of the tree from which requests are generated.

```
(DefPlayer root                                  The 1st level of the tree
  (INIT... )
  (left-child right-child... )
  (d1 d2 d3 d4 no-of-req duration)
  ...
  (INIT . . .
    (repeat no-of-req (run duration)             Generates "no-of-req" requests
      (post left-child REQUEST dl d2)            . ..to left child
      (post right-child REQUEST d3 d4)           . ..and right child
      (wait ack)  (wait ack)                     Wait for their completion
      (run duration))                            Executes for some time
    (terminate)))                                Terminates after "no-of-req"
  requests
```

"Level2" defines the operation of players <2> and <3>. They have different service times for three request types. After receiving a request, they propagate the request to players at the next level and block to wait for all their replies before replying to their parent.

```
(DefPlayer level2                                The 2nd level of the tree
  (INIT REQUEST)
  (parent child-l child-2 child-3)
  (dur-a dur-b)
  (INIT . ..)                                    Initialization
  (REQUEST (record dur-a dur-b)                  Request received
    (run dur-a)                                  Execute for some time
    (post child-l REQUEST)                       Distribute requests to children
    (post child-2 REQUEST)
    (post child-3 REQUEST)
    (Wait ACK) (wait ACK)  (Wait ACK)            Wait for all their replies
    (run dur-b)                                  Execute for some more and...
    (post master ACK)))                          ...reply
```

"Leaf' defines the operation of players **<4>** to **<9>**. After receiving a request, they carry out some processing before replying to their parent.

```
(DefPlayer leaf                          The 3rd level of the tree
  (REQUEST)
  0
  (dur)
  (INIT (record dur))                    Initialization: record execution time
  (REQUEST                               "Request" received
    (run dur)                            Execute...
    (reply ACK)))                        ...then reply
```