MEASURING THE OVERHEAD IN CONSERVATIVE PARALLEL SIMULATIONS OF MULTICOMPUTER PROGRAMS: DETAILED MEASUREMENTS

Mary L. Bailey
Michael A. Pagels

TR 91-14

MEASURING THE OVERHEAD IN CONSERVATIVE PARALLEL SIMULATIONS OF MULTICOMPUTER PROGRAMS: DETAILED MEASUREMENTS

Mary L. Bailey
Michael A. Pagels

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

ABSTRACT

In this paper we show that it is feasible to characterize the overheads present in conservative parallel simulations of multicomputer programs. We use a modified version of the parallel simulator from the Poker Programming Environment to empirically measure the overhead in two parallel algorithms which use three different interconnection structures. We discuss the sources of overhead and qualitatively discuss their relative importance.

1 INTRODUCTION

There has been a great deal of interest over the past few years in comparing conservative and optimistic strategies for parallel discrete-event simulations. The work in this area can be categorized as empirical studies and analytical or formal models. In the empirical studies, specific experiments are run on both conservative and optimistic simulators to see which strategy results in a faster simulation. Fujimoto (1989) did this for closed queuing networks and found that the optimistic strategy generally outperformed the conservative strategy. Reynolds and Dickens (1989) have developed a test bed for comparing the two strategies and are currently using it to compare the synchronization strategies with various applications.

In addition to these empirical studies, there has been a flurry of activity in formal or analytical models for comparing the two synchronization strategies. Here different assumptions are made to keep the analysis tractable, such as requiring one process per processor, or vastly simplifying the overhead costs. In contrast to the empirical experiments, where a single application domain is investigated, most of the analytical studies consider all domains (Felderman and Kleinrock 1991, Nicol 1990, Lin and Lazowska 1990, Madisetti, Walrand and Messerschmitt 1990, Lipton and Mizell 1990, Mitra and Mitrani 1984). There has been some work in domain-specific formal models, but this seems to be the exception, rather than the rule, and simplifying assumptions are still made for overhead costs and processes per processors (Lin, Lazowska and Bailey 1990).

The focus of our work lies between these two traditional approaches. We have performed an empirical study using the conservative strategy in which we examine its performance and more importantly investigate whether it is feasible to characterize the overheads in the simulation so that they can be used in analytical models. The application domain which we have chosen is simulating multicomputer programs, i. e., programs written for non-shared memory parallel processors. In particular, we have taken a well-established multicomputer programming environment, the Poker Programming Environment, to use for our work (Snyder 1984). We have modified a parallel version of the Poker simulator by adding a conservative communication strategy based on the Chandy-Misra paradigm, and have characterized the overheads using two Poker programs. Our characterizations are not sufficiently tuned for use in analytic models, but we believe we demonstrate the feasibility of this approach, and its future efficacy.

The organization of this paper is as follows. We first describe the parallel multicomputer simulator which we used in the experiments, including brief overviews of the original Poker simulator and the conservative algorithm. Then we discuss the programs that we used for our measurements. Next is a discussion of the results of the empirical study, including both general overhead costs together with a more specific breakdown of the overhead costs. Finally, we conclude and discuss future directions.

2 THE PARALLEL SIMULATOR

At the core of our empirical work is the Poker simulator and a Chandy-Misra algorithm. We will briefly discuss each of these separately, and then will describe the interactions between the two when we created the Chandy-Misra version of the Poker simulator.

2.1 The Chandy-Misra Paradigm

In synchronous event-driven simulation, two independent events may not execute in parallel if they have different time stamps. Asynchronous strategies attempt to increase the number of events available for parallel evaluation by allowing independent events to be executed in parallel. These parallel simulations must produce the same result as an equivalent sequential simulation, so the focus is on developing strategies for ensuring this correctness while completing the simulation as quickly as possible. There are two general strategies that are most prevalent in the literature: conservative and optimistic. In both asynchronous strategies, as in the synchronous strategy, the processes are divided among the simulation processors, with each executing events for its partition of the problem space. Each process also maintains a local clock and one or more local event queues. Events queued for this process can be executed if their time stamps equal the value of the local clock. The two strategies differ in the way the local clocks advance. We focus on the conservative strategy in the remainder of this paper. The interested reader is referred to Fujimoto (1990) for a more complete description.

In the conservative strategy, local clocks can advance only if it can be guaranteed that the process will not receive an event with a time stamp less than the new value of the local clock. In other words, no events can arrive that are in the "past". Chandy and Misra (1979) and Bryant (1977) pioneered this strategy. We will summarize the key ideas in the asynchronous strategies. To simplify the explanation, we will assume that there is one process per simulation processor. This is not a requirement, and we do not have this situation in our simulations, since we expect that there will be many more processors in the multiprocessor than in the simulation engine.

Another requirement for the conservative strategy is a static process communication graph. In this graph, there is a directed arc from a process to another if and only if the first process will send message(s) to the second one. The graph may not change as the simulation progresses as the graph is used to determine when to increment the local simulation clocks. Each simulation process maintains an input queue for each incoming arc in the communication graph and sends outgoing events to the appropriate process queue as determined by the process communication graph. The conservative strategy requires that for each arc in the process communication graph, events arrive in increasing time stamp order. This enables the receiving process to consider only a single event from each incoming edge in deciding whether to increase its simulation clock.

There are two rules that are used to ensure the "conservative" requirements in the algorithm. The first, the input waiting rule, states that a process must wait for an event on each incoming edge in the corresponding communication graph before advancing the clock. The clock time is then advanced to the minimum of the time stamps of the events in all queues. Because we know that on each arc, events arrive in time stamp order, we know that there will be no event arriving earlier than the minimum time stamp. The second rule, the output waiting rule, states that output messages cannot be sent until the simulation clock time equals the time of the outgoing message. This guarantees that output messages are sent in time stamp order. There is an explicit assumption that the hardware maintains this message ordering when transmitting messages. The output waiting rule is often relaxed if there is a minimum delay between any event and resulting output messages. Its function is to ensure that no later event will generate output messages with time stamps less than those already transmitted.

As a consequence of these two rules, there can be substantial idle time while a process waits for input messages, and there can be substantial delays between creating an output message and its transmission. In particular, the system can deadlock because of both the input and output waiting rules. Thus the simulation strategy must be able to detect and recover from deadlock or to avoid deadlock. The most popular deadlock avoidance mechanism is to use "null messages," messages which only transmit timing information, to ensure that the simulation can proceed. Thus the overhead in this system must account for deadlock detection and recovery or transmission of null messages. In these experiments we primarily use null messages because it seems to be the most popular implementation of the conservative strategy. However, we also have a deadlock detection mechanism, since the simulated system can deadlock.

2.2 Poker

The Poker Programming Environment consists of a programming language, together with a simulation/debugging environment used to simulate the programming language (Snyder 1984 and Notkin et al. 1988). We will provide a brief summary of the programming environment here; for more information, see Snyder (1988). A Poker program is not a monolithic text file, but is represented by a database. The execution of a Poker program occurs in one or more phases. Different phases often have different interconnection structures.

There are two languages that are currently supported in Poker which are used for the sequential process code, XX (dos equis) and Poker C. Poker C is the more robust language and uses a faster, more generic simulator; we use it as our experimental platform.

The current version of Poker C supports two message primitives, one for sending a message and another for receiving messages. In both of these a single message is sent/received on a specific port. Processes block on a receive until the message arrives; sends are non-blocking. Events in the Poker simulator are generated by sending and receiving messages. The current parallel implementation of the Poker simulator uses a data-driven model since receives are blocking and only a single port can supply the data to be received. Thus there is no need for a more general synchronization strategy to insure that the simulation is correct.

In order to test the viability of the conservative synchronization primitives, a second type of receive primitive was added to the Poker C language, *MultiRead*. Here one can receive a message from one of several specified ports, and whichever message arrives first will be the one which is delivered [1]. The addition of *MultiRead* eliminates the possibility of using the data-driven paradigm for the parallel version of the Poker simulator. The simulator now must insure that the event corresponding to the message with the least time stamp among the specified ports is actually simulated first; otherwise the simulation is not accurately reflecting the performance of the sequential processes.

The addition of the *MultiRead* construct is actually useful in many Poker programs. There are often cases where one needs to get values from several ports and the order that they arrive is immaterial [2]. To make the construct more meaningful, we allow the user to check to see which port provided the message that was read during the *MultiRead*.

2.3 Chandy-Misra Poker Simulator

The Chandy-Misra version of the Poker simulator uses much of the original Poker simulator code intact. There were two major modifications to the data-driven parallel version of the Poker simulator which were necessary to create the Chandy-Misra version, adding the *MultiRead* and adding null messages to avoid deadlock. In addition we modified the parser for Poker C to accept the new *MultiRead* construct.

Adding *MultiRead* to the simulator was relatively straightforward. Because messages are generated and sent in time stamp order, the output waiting rule is not needed in this environment. The code for the input waiting rule is isolated in the *MultiRead* function. We basically execute a loop waiting for the input waiting rule to be satisfied. In this loop we perform a context switch if the waiting rule is not satisfied. Thus we use the number of context switches when the desired message is already present to provide an estimate of the overhead from the input waiting rule, since it is a measure of the substantive source of overhead from the input waiting rule. See Appendix A for additional details.

The other major change is the implementation of null messages to the system to avoid deadlocks. Null messages are implemented as "pokes" in shared memory, which generates less overhead than if they are implemented as full-blown messages. When a process performs a *Read,* it "pokes" all other processes which are connected to it [3]. In our experiments 79% to 99% of simulated time is spent in I/O. In programs with lower *I/O* to compute ratios, or where the ratio is very asymmetric between processes, a method to "poke" processes during compute time would likely be beneficial.

Besides these two changes, we modified the Poker C deadlock detection algorithm. The original deadlock detection algorithm had some timing problems, causing it to report that the simulator was deadlocked

1. Note that the message delivery ordering must be in terms of the local clocks on each multicomputer process, as opposed to the simulator's time.

2. In an earlier version of Poker, the XX language supported a similar construct, where the user could read from multiple ports and these reads were done in order of message arrival time. This construct was not implemented in the original Poker C.

3. Since all arcs in the graph are bidirectional, this is equivalent to "poking" all outgoing arcs.

when it wasn't. These timing problems were exacerbated with the addition of *MultiReads*.

3 THE TEST PROGRAMS

Two test programs have been run to test the efficiency of the Chandy-Misra strategy on Poker programs. One is a systolic Matrix Multiply algorithm. The other program implements the Jacobi iterative method for solving Laplace's equation on a rectangle. The Matrix Multiply algorithm and a single-point version of the Jacobi Iteration are described in Nelson (1987). Both are implemented on 64 Poker processing elements (PPEs). Because the structure of the algorithm impacts the resulting performance and overhead, we will describe each of the algorithms briefly.

3.1 Matrix Multiply

The matrix multiply algorithm is a well-known wavefront algorithm first proposed by Kung et al. (1982) to multiply two $n \times n$ matrices, A and B together. Our implementation uses two 8×8 input arrays. It differs from the original algorithm in that the matrix elements begin in the PPEs instead of being fed into the matrix from the edges. The 64 PPEs are connected in a 8×8 torus, created from an 8×8 matrix by connecting the ends of each row and column. The matrix elements flow vertically or horizontally around the torus. The result matrix is also stored in the processors.

In this implementation, the elements $A(i,j)$ and $B(i,j)$ are pre-routed to the appropriate processor so that the data is staged for the systolic portion of the algorithm. The result array, C, has element $C(i,j)$ stored in processor (i,j). This algorithm is implemented in two phases. In the first phase we load random numbers into the appropriate processors to form the matrices A and B. In the second phase the matrix multiply is performed. All measurements reported in this article pertain only to the second phase, the matrix multiply.

3.2 Jacobi Iteration

The Jacobi Iteration is a parallel implementation of the Jacobi iterative method for solving Laplace's equation on a rectangle. The rectangle is represented by n discrete values which correspond to the voltages at points in the rectangle. The boundary and the voltage sources are fixed constants. In the algorithm, an initial guess is computed for each of the points, and then new values are iteratively obtained by averaging the values of its four neighbors until the voltage stabilizes.

In the Poker implementation, this process is represented as two phases, one for the iterations, and another to determine whether the system has stabilized. The iterate or compute phase uses a mesh interconnection structure, since each processor must obtain values from each neighbor. The aggregate phase uses a tree interconnection structure since it must determine whether all processes have stabilized. In our implementation a rectangular set of points is maintained in each processor, and in the compute phase, a single message is sent to each of the four neighbors containing the appropriate array of points necessary for communication. Additionally, we iterate 10 times in this phase to decrease the overhead of changing phases. Increasing the number of iterations also increases the resulting simulation time, so we obtain more accurate measurements.

We took measurements on both phases of this problem since both phases are pertinent to the algorithm and can use the *MultiRead* operator. Since in the compute phase the algorithm receives a set of values from each of its 4 neighbors and then performs the averaging, it doesn't matter which values are received first, so we used the *MultiRead* construct on the four ports. Because four array boundaries of elements were being received, it was necessary to understand which port was being read, which complicated the code somewhat. Likewise in the aggregate phase we are simply taking the maximum of a set of values stored one in each processor (the maximum voltage change in the last ten iterations), so order is not important here.

4 EMPIRICAL RESULTS

The goal of the empirical tests is to determine the amount and types of overhead present due to the Chandy-Misra algorithm we added to the Poker simulator. We will first discuss the general experimental methodology used to take the measurements, and then will discuss the actual experiments performed.

4.1 Methodology

All programs were run on a Sequent Symmetry, with eight processors running at 16 MHz. On this machine, one processor must be reserved for the operating system, so the maximum number of processors available for measurements is seven. All experiments were run when the machine was in single user mode, with "tmp_affinity" set so that processes are bound to specific processors[1].

1. This decreases the operating system overhead by reducing cache conflicts.

Two versions of the Poker Simulator and two versions of each test program were used in the measurements. Table 1 shows the names that we use for each of the measurements.

Table 1: Naming Conventions used in the Measurements

	No MultiRead	MultiRead
Original Simulator	Original	
Modified Simulator	Plain	MultiRead

Original is the original Poker simulator, with the algorithm written *without* the *MultiRead* construct. This provides a baseline for our comparison, since there is no overhead here due to the Chandy-Misra algorithm. **Plain** uses the Chandy-Misra version of the Poker simulator but run on the version of the algorithm which does *not* use *MultiRead*. Thus there is no input waiting rule overhead in this version, but there is overhead due to null messages. **MultiRead** uses the Chandy-Misra version of the Poker simulator run on the version of the algorithm which uses *MultiRead*. Both null message and input waiting rule overheads will be present in this version.

In both versions, there is a heavy-weight UNIX process associated with each running Symmetry processor. Poker PPEs are implemented as light-weight processes using a light-weight threads package provided in the original Poker parallel simulator. The heavy-weight processes are actually just parked (suspended) after the first phase of a Poker program; in subsequent phases these processes are simply signaled

We begin by presenting the execution times of the three simulators on each of the algorithms. These are shown in Figure 1, Figure 2, and Figure 3. The average execution times are plotted in the graphs. Standard deviations are shown using error bars.

Each of the points in the figure represents the average of 22 runs. The execution times were printed out as part of the program and measures the execution time for each phase from the point just before the processes are forked to the point when the last forked process has completed. The measurements do not include any overhead time to print the statistical information that was gathered during the run [1].

1. We compared the execution times both with and without measurements and found no significant difference. Thus all results here are with measurements turned on. Appendix A details the measurement points added to the simulator

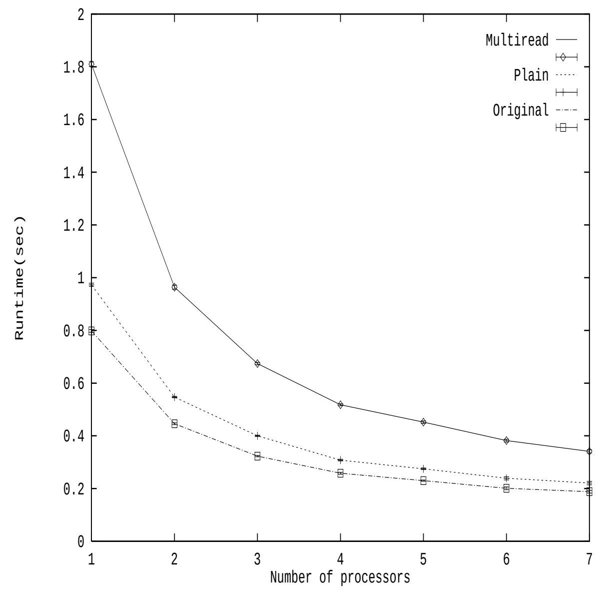


Figure 1: Executation Times for Matrix Multiplication

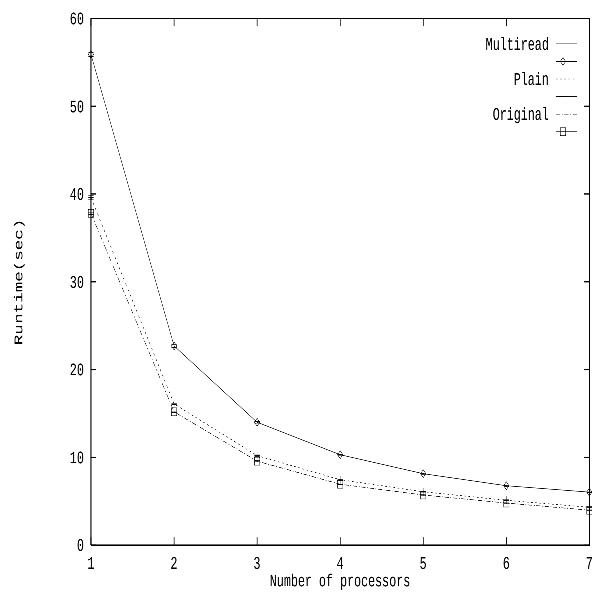


Figure 2: Executation Time for the Jacobi Computation Phase

In the following sections we will discuss the overheads found in each of the two algorithms. See Appendix A and Appendix B for more detailed experimental data.

4.2 Matrix Multiply

The easiest way to visualize the overheads in the modified Poker simulator is to consider the difference in execution times between the three experimental

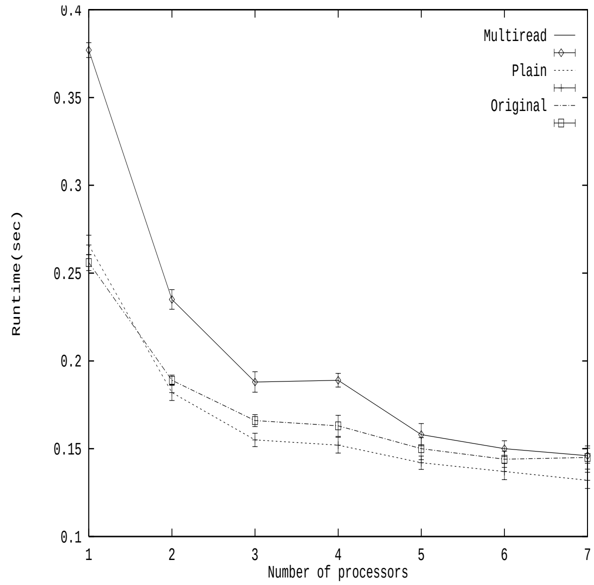


Figure 3: Executation Time for the Jacobi Aggregate Phase

models: **Original**, **Plain**, and **MultiRea**d. Figure 4 shows these for the Matrix Multiply algorithm.

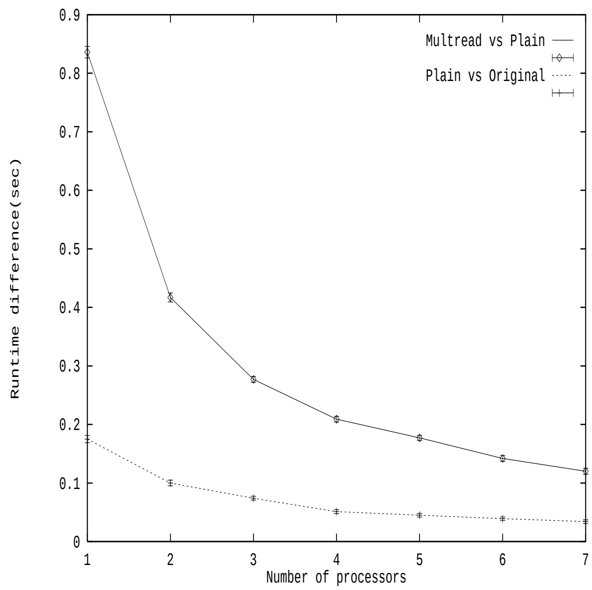


Figure 4: Difference in Simulation Executation Times for the Matrix Multiply

We begin by considering the difference in execution times for the **Original** and **Plain** measurements. Since the same algorithm is used in both measurements, and no *MultiRead*s are used, this difference must be completely due to null message overhead.

We computed the total number of null messages in the runs and found it to be invariant with respect to the number of processors. There are exactly 4.25 null messages sent per read in all cases, and there are a total of 1024 reads, 16 per PPE. Whenever a read is done, all 4 neighbors are poked. Additionally, all neighbors are poked when each PPE finishes. Thus we obtain an average of 4.25 pokes per read. There is no additional overhead from extraneous context switches due to processors waiting for Reads except when the number of processors is five or six. In these cases the total number of extraneous context switches is small. Thus we can conclude that the overhead due to null messages is essentially linear in the number of PPEs, since every PPE generates and receives the same number of null messages. Additionally, the PPE's are mapped to simulator processes in row-major order, so in most cases the same number of PPE's are mapped onto every processor.

If we assume that the original Poker simulator reflects a "typical" parallel execution curve, then we can approximate the execution time of the simulation as $T_{orig} = \varepsilon p + c\frac{W}{p}$ where p is the number of processors, W is the total amount of work, c is a constant related to the extra code due to the original Poker parallel version, and ε is a small constant involved with initialization, such as forking processes. Since we have basically added a constant amount of work to W per processor, we can compute the difference between the Original and Plain versions to be: where δW is the amount of work we added to each processor.

Using the 6 processor value as the basis point we fit $\Delta(p) = c\frac{\delta W}{n}$ to the difference curves in Figure 4. With the exception of the single processor case, the data from **Plain** vs. **Original** fits within one standard deviation of the calculated values. The single processor difference is less than predicted, indicating that based on multiprocessor performance, single processor performance is faster than expected. More analysis is required to exactly determine the cause of this effect.

Now we consider the overhead differences between the **Plain** and **MultiRead** execution times. In the one to four processor runs, the total number of null messages did not change. In addition, there were no extraneous context switches due to *MultiRead*s. For the larger numbers of processors, there were additional null message and context switches. These all occurred on the processor with the least balanced load. For instance, when 5 processors are used for 64 PPEs, four processors contain code for 13 PPEs and the other processor contains code for only 12 PPEs. It is this fifth processor which generates all of the context switches and additional null messages. An analogous situation holds for the six and seven processor case, although in the six processor case there are two processors which are unbalanced so both are performing context switches and sending extra null messages, causing the total number of context switches in the six processor case to be much larger than in the five and seven processor cases. These extra context switches have a minimal effect on the overall simulation time, since it is the fast processors which are wasting time. They are likely to have higher idle rates in **Original** and **Plain**.

Thus the difference in execution time is due to the overhead of performing *MultiRead*s instead of Reads, together with additional simulation time required to process the additional code present in the *MultiRead* version of the algorithm. In fact, we see that the **MultiRead** vs. **Plain** curve fits the predicted difference formula, implying that there are no additional overheads due to the input waiting rule which impact the execution time of the simulation.

In summary, we found that in this example, the overheads due to the addition of Chandy-Misra were completely accounted for by the typical execution curve for any parallel algorithm. There are no additional factors related to the Chandy-Misra algorithm, except for computing the increment in the total work. Thus it is straightforward to compute the extra work in this problem which is due to the Chandy-Misra algorithm.

4.3 Jacobi Iteration

We will discuss each phase of the Jacobi Iteration separately since each has a different interconnection structure. We begin with the compute phase, which uses a mesh interconnection structure. As in the matrix multiply, we begin by considering the difference in execution times in the **Original** and **Plain** measurements (see Figure 5).

Once again, the total number of null messages is independent of the number of processors. Here there are a total of 3.63214 null messages sent per read. This number is lower than four because the PPEs on the edges of the mesh send only three messages (the corner PPEs send two), and there are many more Reads, an average of 140 per PPE. Unlike the matrix multiply, there is some context switch overhead in all versions from processes waiting for Reads being awakened by receiving null messages. These seem to be mainly in the PPEs on the edges of the mesh when the number of processors is

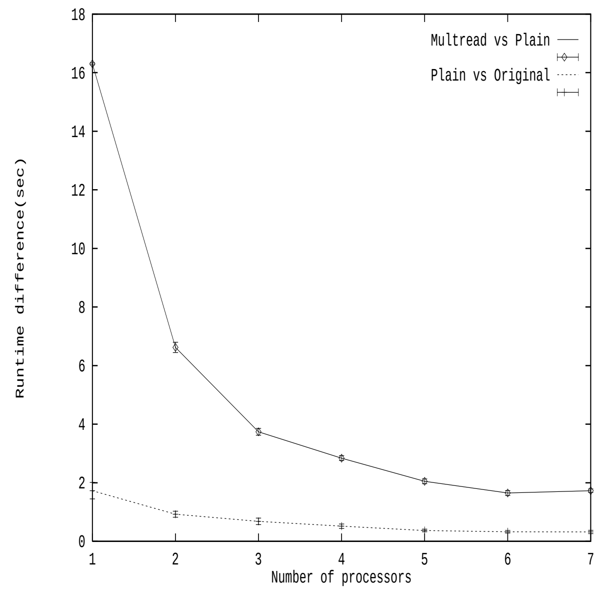


Figure 5: Difference in Simulation Executation Times for the Jacobi Compute Phase

small, but becomes widespread as the number of processors increases. The largest number of context switches occur when five, six, or seven processes are used. Unlike the matrix multiply, although there is often more context switch overhead in the processors with unbalanced numbers of PPEs, there is a significant amount of context switching in other processors. This extra context switching doesn't seem to adversely affect the execution time of the simulation, because we can fit the difference curve to the simple parallel execution model. Hence, the extra "thrashing" which occurs in unbalanced processors is not affecting the overall simulation time.

The overheads due to changes between the **Plain** and **MultiRead** execution times are more difficult to characterize. There is a superlinear decrease in overhead going from one to two processors, which implies that the total overhead in the system actually decreased. In fact, we see this for other numbers of processors also. The data points here do *not* fit the simple parallel execution model, implying that there are additional overhead terms due to the dynamic characteristics of the Chandy-Misra algorithm which affect the simulation execution time.

The measurements of conservative algorithm overheads indicate that the number of null messages increased over **Plain**, and additional context switches due to the input waiting rule were also present. In fact, there is on average between 1 and 1.38 context switches per read. More interestingly, the number of context switches per Read has a similar shape to the average number of null messages per Read. The number of null messages per context switches is not constant, but ranged from 3.9 to 5.1. One interesting phenomenon that we observed is that these overheads decrease when three or five processors are used. We saw a “zig-zag” shaped graph for many of the measured overheads, with the magnitude of the peaks and valleys generally increasing with the number of processors. For example, Figure 6 shows the total

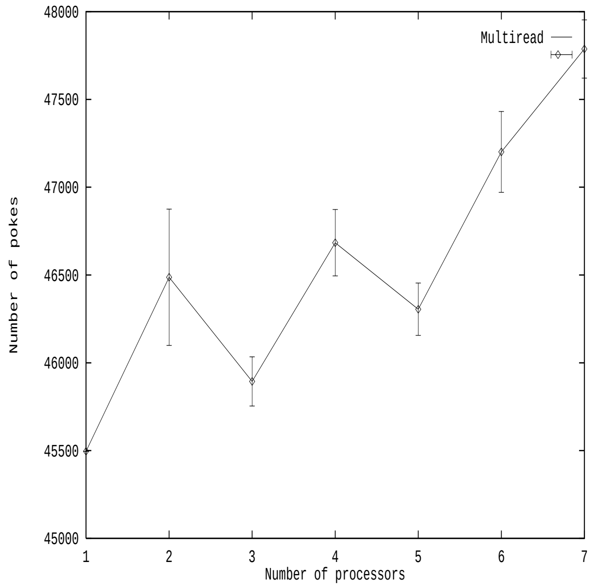


Figure 6: Total Number of Pokes Send in the Jacobi Compute Phase

number of pokes sent as a function of the number of processors.

In summary, we see overheads in the Jacobi compute phase, due to both the presence of null messages and the input waiting rule. This increase is likely due in part to the imbalances caused by edge effects in the mesh, since the PPEs on the edge of the mesh have different communication patterns from those in the interior of the mesh. This problem does not fit the typical parallel execution model, demonstrating that the conservative overheads are impacting the simulation execution time. We did measure the ratio of null messages to context switches and found that there were on average four to five null messages sent per context switch. Thus we can begin to quantitatively compare the overheads spent in the input waiting rule and time spent in processing null messages.

We now consider the overheads in the aggregate phase. This phase uses a binary tree as its interconnection structure, with the modification that the root node has a third child, which is the remaining PPE. Just over half of the PPEs are leaf nodes, which perform no reads, but simply write a value to their parent nodes. The inner nodes of the tree require two *MultiRead*s, and the root node requires three. Thus there are many fewer Reads than in the other phase. Consequently the execution times are much shorter.

As before we begin by considering the difference in execution times between **Original** and **Plain** (Figure 7).

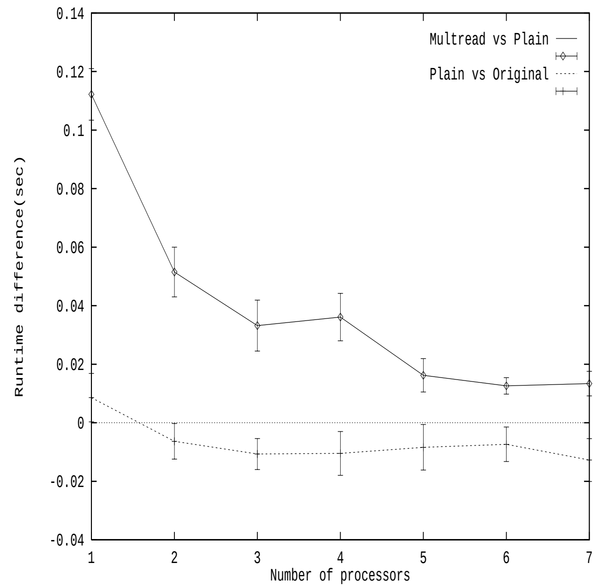


Figure 7: Difference in Executation Times for the Jacobi Aggregate Phase

We have a really unusual case here. If we look at **Original** compared to **Plain**, we see that by adding null messages we actually improved the running time of the simulation in all cases except the uniprocessor case! We once again found the total number of null messages sent was invariant, although the total number sent here was only 315, substantially less than in the other two examples. We see a large number of pokes sent per read, five. As in the matrix multiply, this is due to null pokes being sent as PPEs terminate. The effect is even more dramatic here because half of the PPEs perform no reads, but all terminate. There is also a rather substantial number of extraneous context switches due to null message arrival, an effect seen in all runs. We believe that the increased performance of the **Plain** over **Original** can be attributed to the small amount of additional code which is executed, together with its impact on scheduling of PPEs. Transmission of null messages does impact the scheduler, and in this case, it appears to have increased its efficiency.

The overheads seen in the *MultiRead* experiments are relatively straightforward. The total number of null messages generally decreases with the number of processors, although there is an increase in the cases when the processors are more poorly balanced. We also see that there are a number of instances when null messages are considered but not sent. In our code, we only send null messages if the time stamp differs from the last null message that was sent. In this phase, there seem to be a large number of times when the null message being sent has the same time stamp as the last null message the PPE sent. This seems to be most pronounced in the three and five processor experiments. We also see some overhead due to the input waiting rule, although there are many fewer context switches than in the compute phase. In fact, the number of null message per context switch ranges from 20 to 70, a huge increase over the compute phase. Four processors seems to be particularly susceptible to extraneous context switches. It is the instance where poor PPE balancing is most evident.

To summarize the results for the aggregate phase, we see very different behavior in this phase than in either of the two other experiments. This is likely due in part to the small execution time of this phase. Before any concrete conclusions can be drawn concerning the effect of the tree interconnection structure on the overheads in the conservative algorithms, a more time-consuming example needs to be used.

5 CONCLUSIONS

We were able to measure many of the overheads present in the conservative algorithm. In the first example, the matrix multiply, we saw that a simple execution model was generally able to explain the difference in the **MultiRead** and **Plain** versions, demonstrating the minimal effects of overhead due to the conservative algorithm. The major overhead present in this case was the addition of null messages which increased the overall workload. The absence of other effects is likely due to the well-balanced algorithm.

In the mesh interconnection structure, we saw some impact of edge effects. Here, the simple execution model no longer explained the difference in execution times. The total number of pokes increased, as well as the number of context switches due to the input waiting rule. The number of pokes per Read ranged from 5.08 to 5.32 and generally increased with the number of processors. The number of pokes per context switch ranged from 3.9 to 5.1 and generally decreased with the number of processors.

The binary tree interconnection structure behaved quite differently from either of the other two. We saw that the addition of null messages actually decreased the simulation time. Unfortunately, the run times were so small, that the data drawn from this experiment is inconclusive. However, we did see that poor balance of PPEs to processors had an large impact on both execution time and context switch overhead.

There is still a great deal of work to be done in characterizing the overheads in conservative simulations. The interconnection structure seems to have a major impact on the resulting overheads. We were able to discuss only a single algorithm using each interconnection structure in this paper. Imbalances in communication patterns and structures appear to result in increased overheads.

In the future, we hope to quantify the effects of the dynamic overheads, such as the effects of the input waiting rule, so that they can be used in analytical models. The data here suggests that this is a viable task, that it is feasible to characterize the overheads in the conservative algorithm for this class of programs.

ACKNOWLEDGEMENTS

We would like to thank John Luiten and the Lab Staff in our department for providing support and single-user access to the Sequent Symmetry. We would also like to thank Larry Snyder for providing access to the Poker programming environment and sample programs, and Richard Fujimoto for giving us his parallel simulators which provided an excellent point of reference for our work. This work is funded in part by National Science Foundation Grant CCR-9110443.

REFERENCES

[1] Bryant, R.E. 1977. Simulation of packet communication architecture computer systems. Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology.

[2] Chandy, K.M. and J. Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* **SE-5(5)**: 440-452.

[3] Felderman, R.E. and L. Kleinrock. 1991. Two processor time warp analysis: Some results on a unifying approach. In *Proceedings of the SCS Multiconference on Distributed Simulation*, eds. V. Madisetti, D. Nicol, and R. Fujimoto, 3-10. SCS, San Diego, California.

[4] Fujimoto, R.M. 1989. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation* **6(3)**: 211-239.

[5] Fujimoto, R.M. 1990. Parallel discrete event simulation. *Communications of the ACM* **33(10)**: 30-53.

[6] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. B. B. Rao. Wavefront array processor: Language, architecture, and applications. *IEEE Transactions on Computers*, C-31(11): 1054-1065, 1982.

[7] Lin, Y.-B. and E.D. Lazowska. 1990. Optimality considerations for "time-warp" parallel simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, ed. D. Nicol, 29-34. SCS, San Diego, California.

[8] Lin, Y.-B., E.D. Lazowska, and M.L. Bailey. 1990. Comparing synchronization protocols for parallel logic-level simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, ed. P.-C. Yew, III-223–III-227. Penn State, University Park, Pennsylvania.

[9] Lipton, R.J. and D.W. Mizell. 1990. Time warp vs. chandy-misra: A worst-case comparison. In *Proceedings of the SCS Multiconference on Distributed Simulation*, ed. D. Nicol, 137-143. SCS, San Diego, California.

[10] Madisetti, V., J. Walrand, and D. Messerschmitt. 1990. Synchronization in message-passing computers – models, algorithms, and analysis. In *Proceedings of the SCS Multiconference on Distributed Simulation*, ed. D. Nicol, 35-48. SCS, San Diego, California.

[11] Mitra, D. and I. Mitrani. 1984 Analysis and optimum performance of two message-passing parallel processors synchronized by rollback. *Performance'84*, 35-50.

[12] Nelson, P.A. 1987. Parallel Programming Paradigms. Ph.D. thesis, Computer Science Department, University of Washington, Seattle, Washington.

[13] Nicol, D.M. 1990. Performance bounds on parallel self-initiating discrete-event simulations. Technical Report 90-21, ICASE.

[14] Notkin, D., L. Snyder, D. Socha, M.L. Bailey, B. Forstall, K. Gates, R. Greenlaw, W.G. Griswold, R.J. Holman, R. Korry, G. Lasswell, R. Mitchell, and P.A. Nelson. 1988. Experiences with poker. In *Proceedings of the ACM/SIGPLAN PPEALS*, 10-20. Association of Computing Machinery, New York, New York.

[15] Reynolds, P.F. and P.N. Dickens. 1989. SPECTRUM: A parallel simulation testbed. In *Proceedings of the 4th Annual Hypercube Conference*.

[16] Snyder, L. 1984. Parallel programming and the Poker programming environment. *Computer* **17(7)**: 27-36.

[17] Snyder, L. 1988. Poker (4.2) programmer's reference guide. Technical Report TR88-10-05, Computer Science Department, University of Washington, Seattle, Washington.

AUTHOR BIOGRAPHIES

MARY L. BAILEY is an assistant professor in the Department of Computer Science at the University of Arizona. Her research interests include parallel and distributed simulation, computer-aided design for VLSI, and parallel computation.

MICHAEL A. PAGELS is a research assistant in the Department of Computer Science at the University of Arizona. His research interests include multi-processor architecture and operating system simulation.

Appendix A: Measurement Modifications

The following code templates identify the locations where measurements were taken during the executation of the simulator. Each Poker Processing Element maintained its own set of counters. The counters were "wrapped" in conditionally compiled code allowing for executation time measures with and without the additional counter code.

```
/*
 * Procedure SendNullMessage
 *
 * Go poke my connections simulating NULL messages
 * The calling process will never send a message earlier than.... time
 */

SendNullMessage( time )
{
   foreach connected port {

      ++CurProcs->pokesConsidered;

      if time is greater than the last time we poked this port {

         ++CurProcs->pokesSent;

         poke the port;

         if the process is waiting for IO from this port {

            ++CurProcs->wakeUpsSent;

            wake the process up;

         }
      }
   }
}


/*
 * Procedure ReadPort
 *
 * Read a message of a specified size on a specified port
 */

ReadPort( ... )
{
   ++CurProcs->reads;

   while there is no data on the requested port {

      ++CurProcs->nullReads;

      put this process to sleep;
   }

   read an process the waiting message;

   SendNullMessage( processes current timestamp );
}


/*
 * Procedure ReadMultiPort
 *
 * Read a message from anyone of a set of ports
 * Apply the input waiting rule to insure that the earliest
 *  possible message is read first.
 */

ReadMultiPort( ... )
{

   ++CurProcs->multiPortReads;

   loop until done {

      foreach port requested in the MultiRead {
```

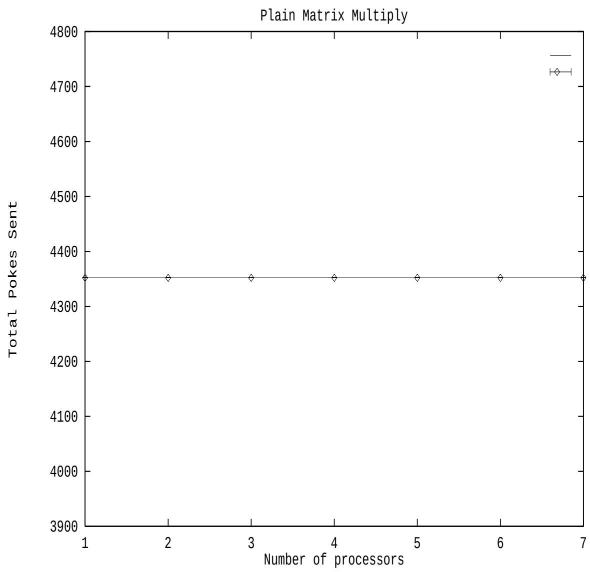
```
         identify the port with the earliest possible message arrival;

      }

      if there is no message ready on the required port {

         SendNullMessage( earliest arrival time );

         forevery port we have a connection too {

            if there is a message waiting at that port {

               ++switchCount[ port ];

            }

            ++CurProcs->switches;

            put this process to sleep;
         }
      }

      else {

         CurProcs->waitedOnWrong += switchCount[ identified earliest arrival ];

         done;
      }
   }

   ReadPort( identified earliest arrival );
}

/*
 * Procedure wakeupOldPokes
 *
 * It is possible for a poke to occur after a process has last examined
 * its input queue and decided to go to sleep.  This routine awakens
 * processes that are asleep with waiting pokes.
 */

wakeupOldPokes()
{

   for each sleeping process {

      if process went to sleep with an unprocessed poke {

         ++Process->sleeperWakeUpsSent;

         wait the process up;

      }
   }
}

/*
 * Procedure ContextSwitch:
 *
 * Suspends the current process.  Starts the next runable process
 */
ContextSwitch( ... )
{                       /* Numbers are indexes for sequent */

    ++CurProcs->contextSwitches;

    suspend the current process;

    activate the next runnable process;
}
```

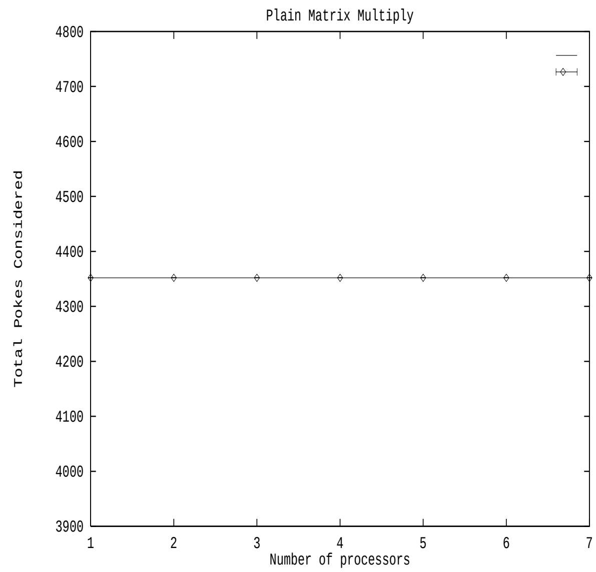
Appendix B: Performance Graphs

Plain Matrix Multiply

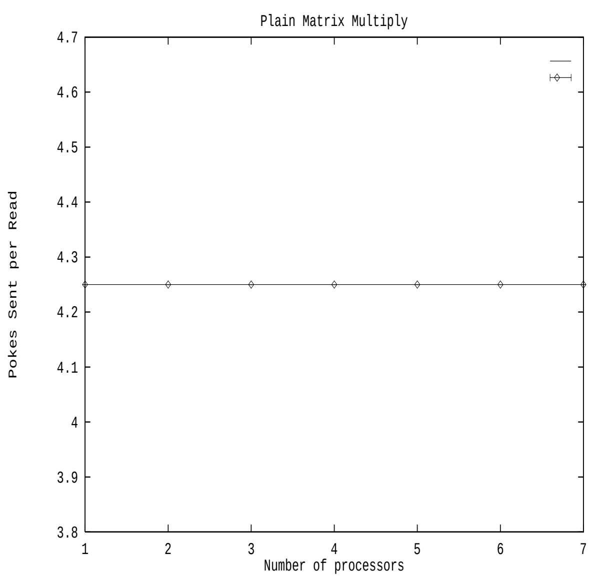
The graphs in this section correspond to the Chandy-Misra version of the Poker simulator run on the Matrix Multiply code which does *not* use *MultiRead*. Thus these graphs measure the effects of adding null messages to the original Poker simulator. In the graphs, the term “pokes” is used in lieu of “null messages”.

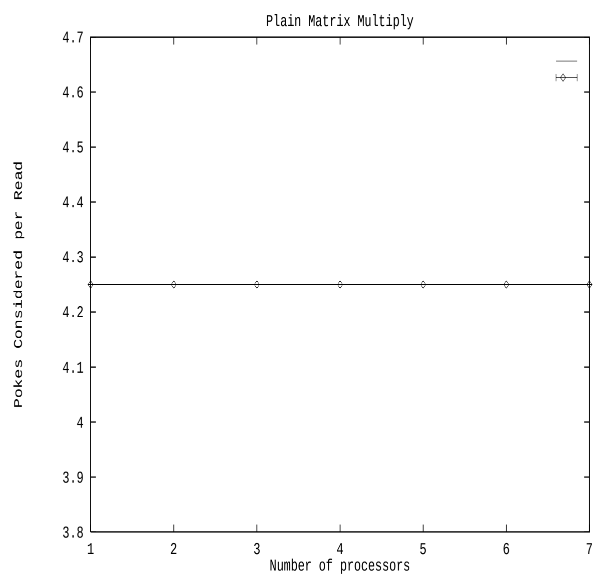
We begin by considering the total number of null messages, or “pokes”. On the left we show the total number of null messages that were sent and see that the number is invariant with respect to the number of processors. It is also interesting to see that the number is invariant over all sampled runs. On the right we show the total number of null messages considered, which equals the number of null messages sent in this algorithm.



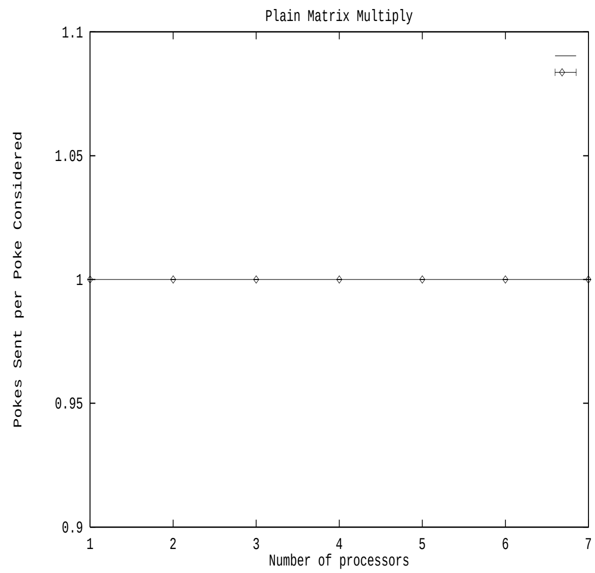


In the next two graphs we simply divide the data in the above graphs by the number of reads, which is also invariant in these simulations, so we once again obtain constant data.

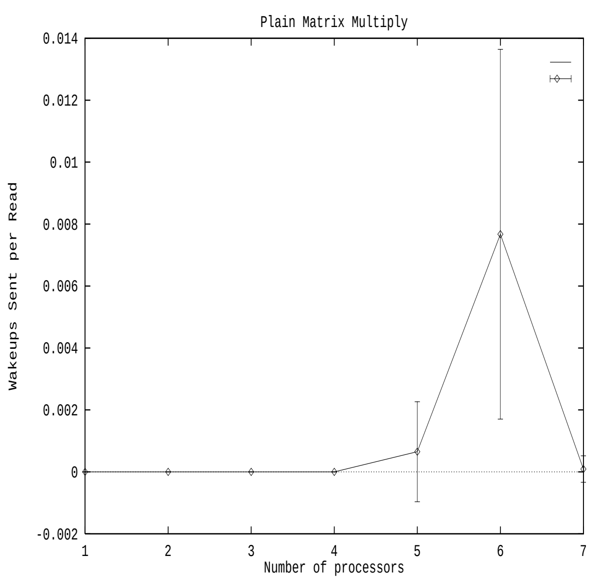


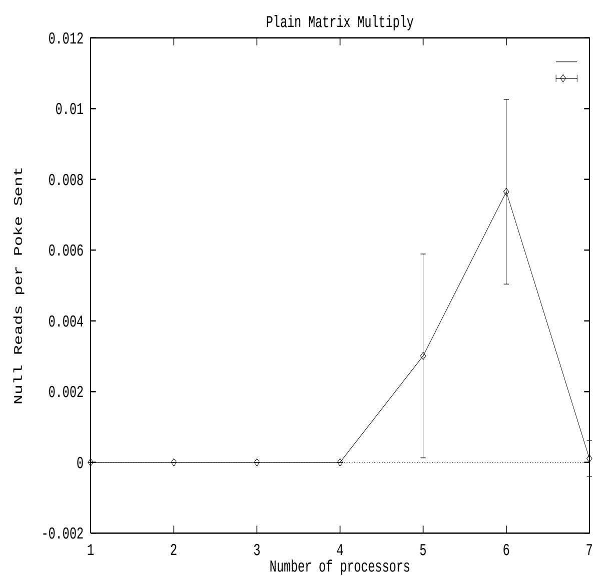


The next graph verifies that all null messages which were considered were actually sent.



The final two graphs for these simulation runs involve the effect of the null messages on the Poker threads package. The graph on the left shows the frequency that reads cause adjacent Poker processing elements to be moved onto the ready queue when they were blocked waiting for reads. The graph on the right shows the frequency that null messages cause extraneous context switches. In these cases the null messages caused the processing elements to be placed on the ready queue, and when they were activated, they were immediately swapped back out because no I/O messages had arrived.

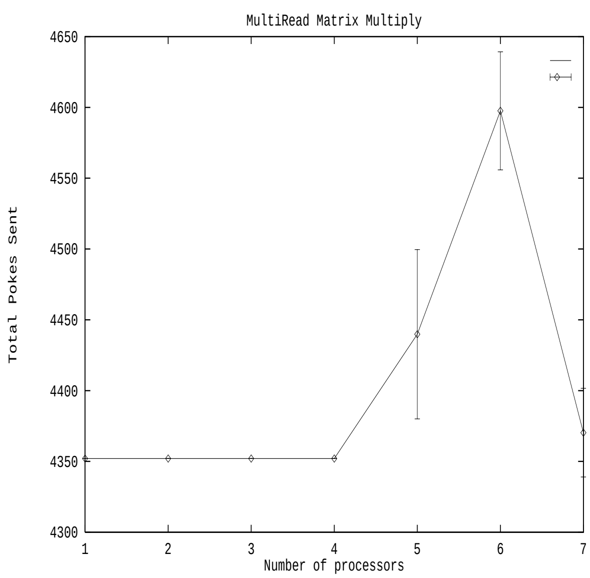


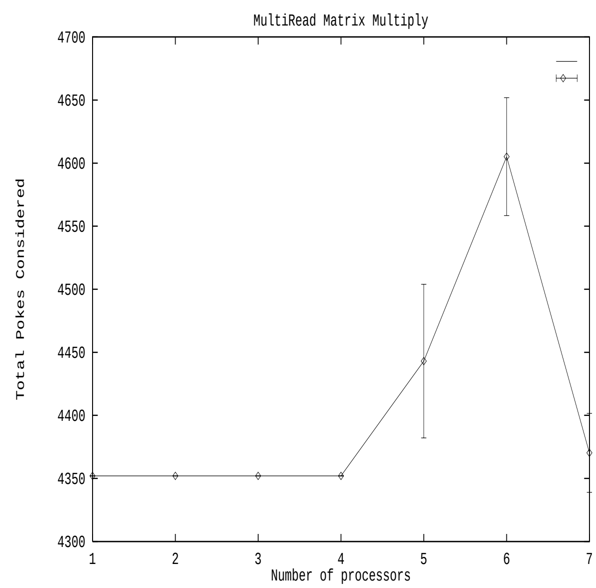


MultiRead Matrix Multiply

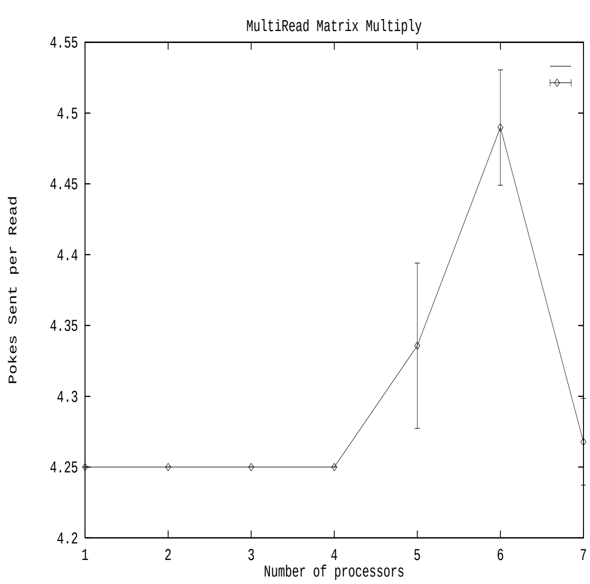
The graphs in this section correspond to the Chandy-Misra version of the Poker simulator running the version of the Matrix Multiply algorithm using *MultiRead* construct. Here, not only do we see the effect of null messages on the overhead, but overhead due to the input waiting rule is also present. Once again, the term “pokes” is used in lieu of the term “null messages” in the graphs.

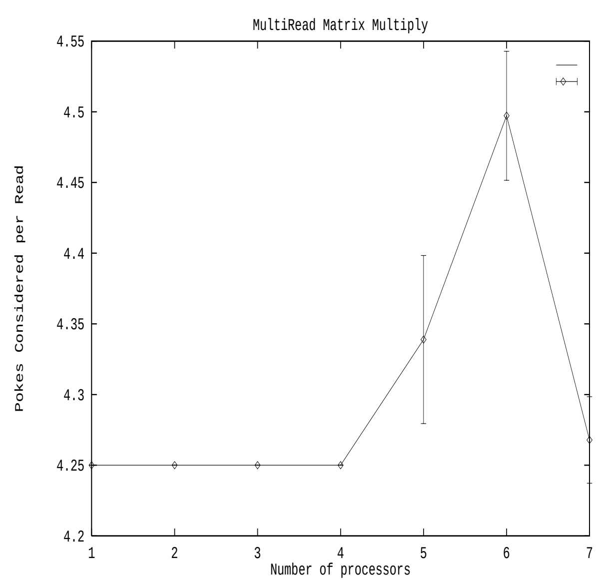
Once again we begin by considering the total number of null messages or “pokes”. On the left we show the total number of null messages which were sent, and see that it is no longer invariant with respect to the number of processors. Likewise, the total number of null messages which were considered also changes with the number of processors. This graph is shown on the right.



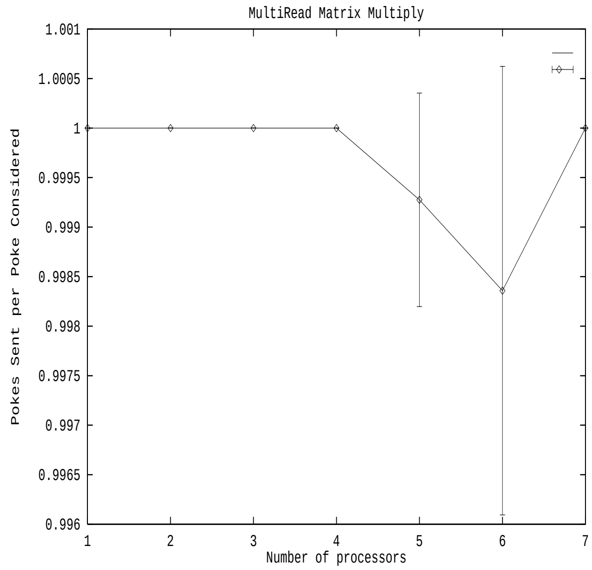


In the next two graphs the data in the above graphs in simply divided by the number of reads, which is an invariant in the simulations.

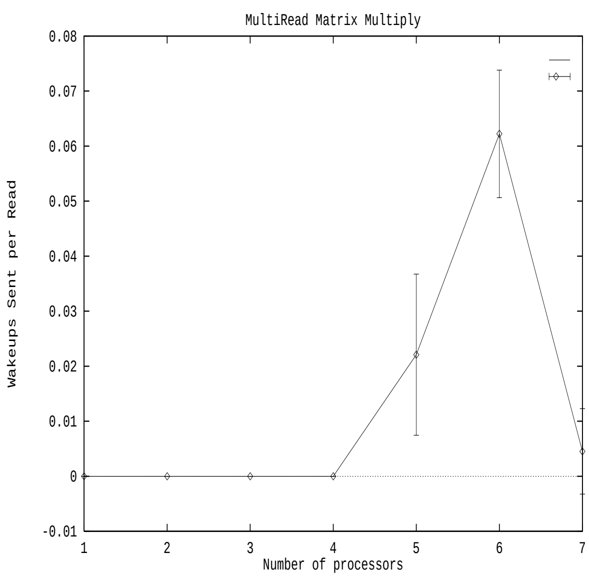


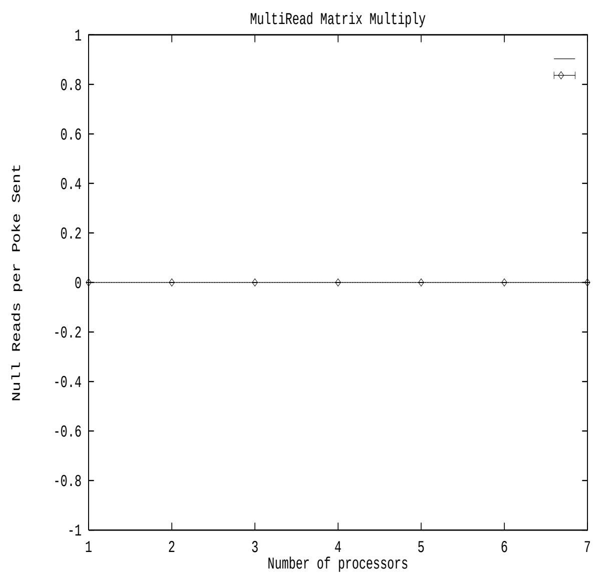


The next graph shows that there were times when null messages were considered, but not sent. This corresponds to the times when the null message would not have increased the timestamp of the processing element's clock, so it was not necessary to send the message.

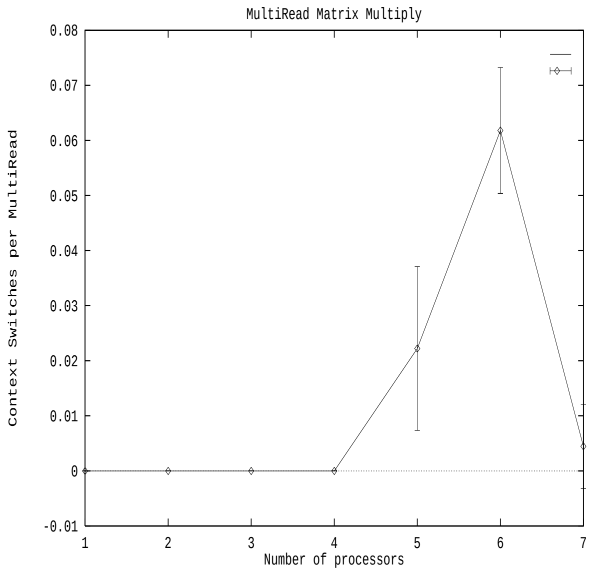


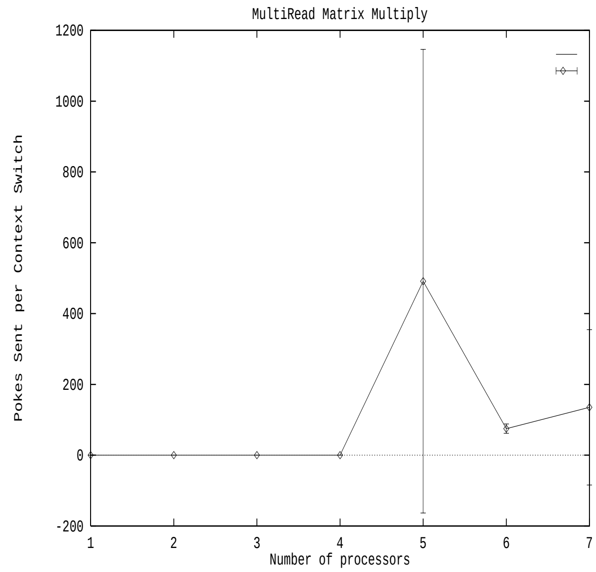
The following two graphs involve the effect of null messages on the context switching of Poker processing elements. The left graph shows the frequency that reads by adjacent processors cause processing elements to be moved onto the ready queue when they are blocked waiting for I/O. The graph on the right shows the frequency that null messages cause extraneous context switches, i.e. that they cause processing elements to be moved to the ready queue and when activated, the processing elements are immediately swapped back out because no I/O messages had arrived.





The final two graphs pertain to the overhead of the input waiting rule. The graph on the left shows the number of context switches per *MultiRead*, an indication of the amount of time which should be charged to the input waiting rule. The graph on the right shows the number of pokes sent per context switch, which can be used to asses the relative magnitude of the input waiting rule and null message overheads.

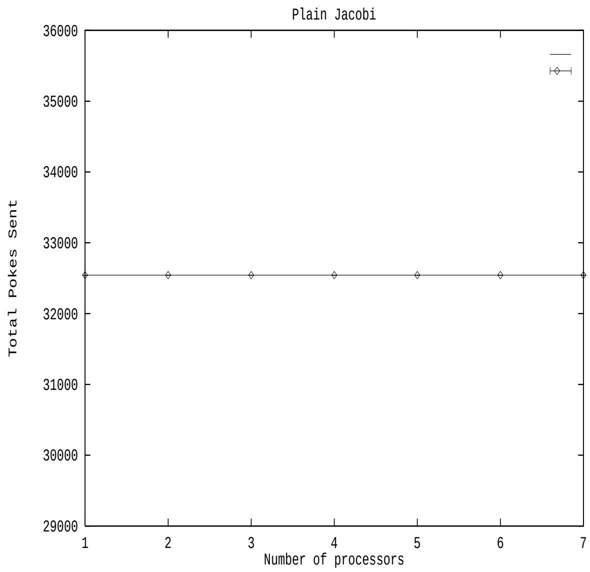


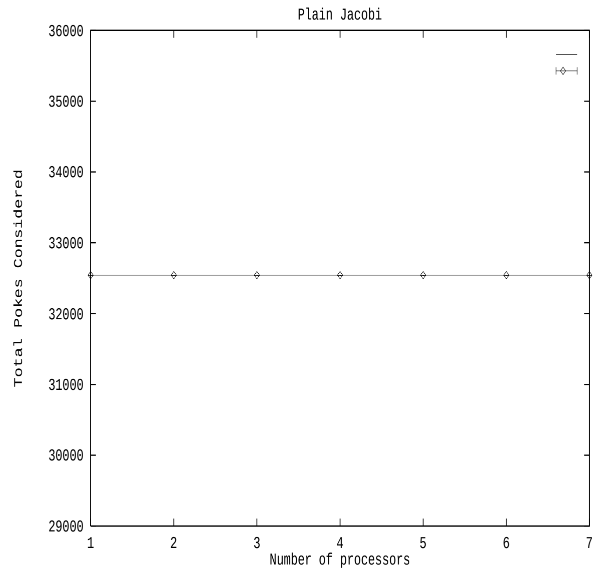


Plain Jacobi Iteration

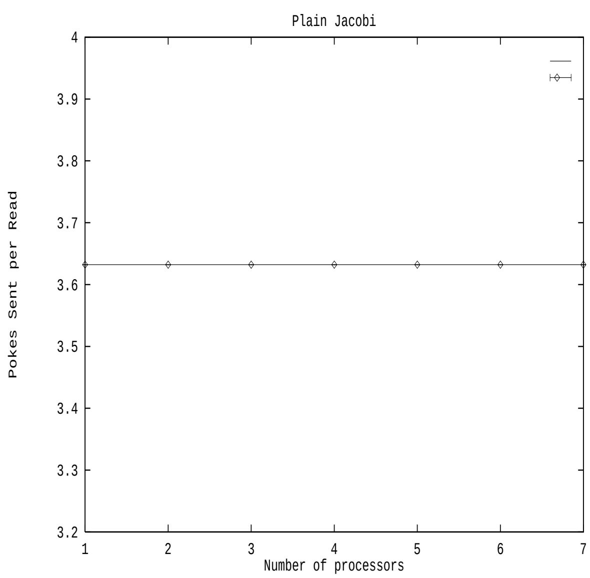
The graphs in this section correspond to the Chandy-Misra version of the Poker simulator run on the Jacobi Iteration code which does *not* use *MultiRead.* We only show the results of the compute phase of the algorithm, since we believe that the data for the aggregate phase may not be representative since the time spent in this phase is so short. Once again, the graphs in this section measure the effects of adding null messages, or “pokes”, to the original Poker simulator; no overheads are present from the input waiting rule.

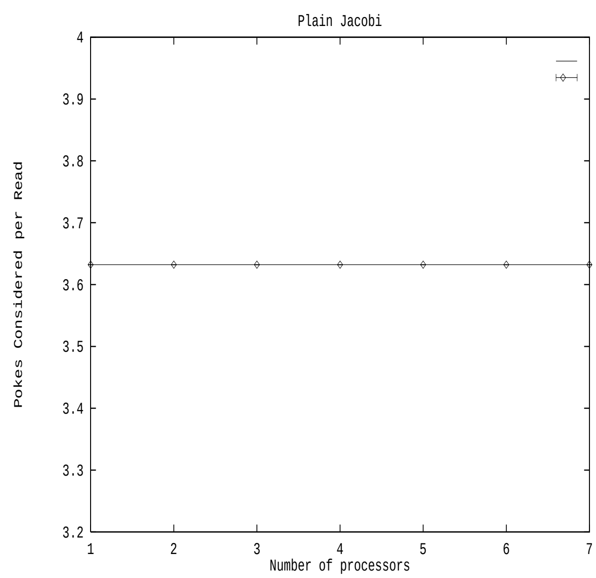
We begin by considering the total number of null messages, or “pokes”. On the left we show the total number of null messages that were sent and see that the number is invariant with respect to the number of processors. It is also interesting to see that the number is invariant over all sampled runs. On the right we show the total number of null messages considered, which equals the number of null messages sent in this algorithm



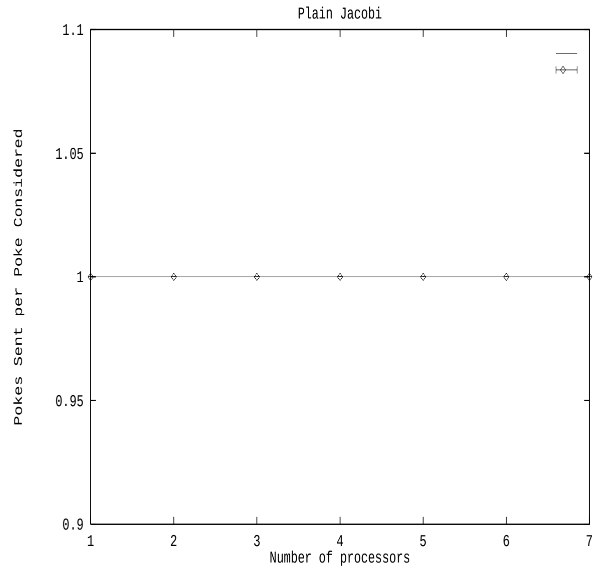


In the next two graphs, we simply divide the data in the above graphs by the number of reads, which is also invariant in the simulations, so we once again obtain constant data.

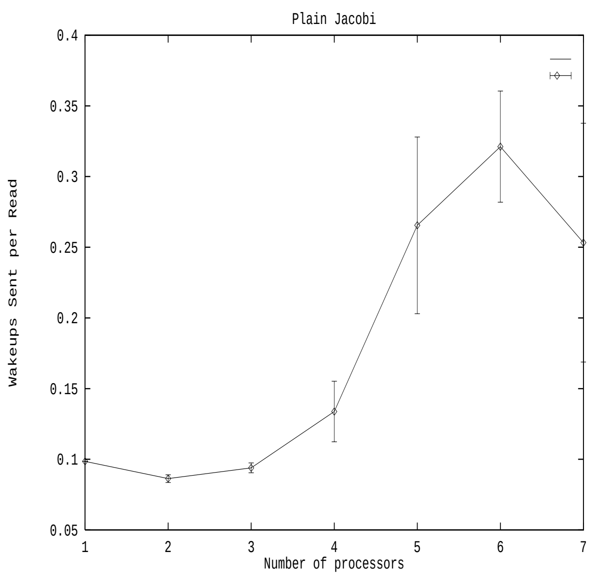


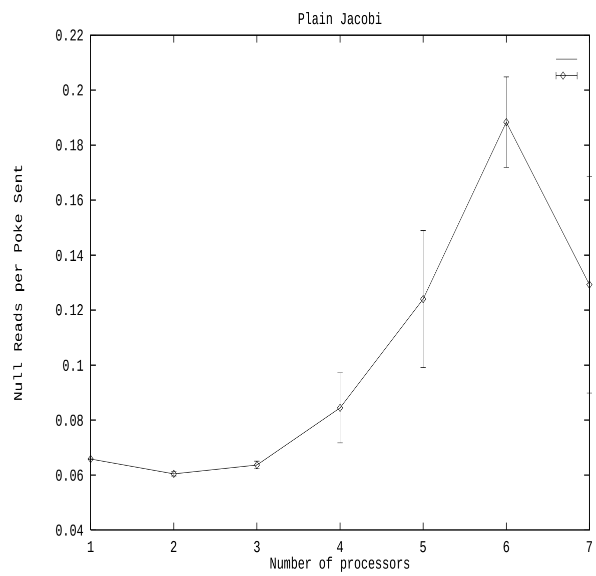


The next graph verifies that all null messages which were considered were actually sent.



The final two graphs for these simulation runs involve the effect of the null messages on the Poker threads package. The graph on the left shows the frequency that reads cause adjacent Poker processing elements to be moved onto the ready queue when they were blocked waiting for reads. The graph on the right shows the frequency that null messages cause extraneous context switches. In these cases the null messages caused the processing elements to be placed on the ready queue, and when they were activated, they were immediately swapped back out because no I/O messages had arrived.We see much more variation in this algorithm than in the Matrix Multiply algorithm.

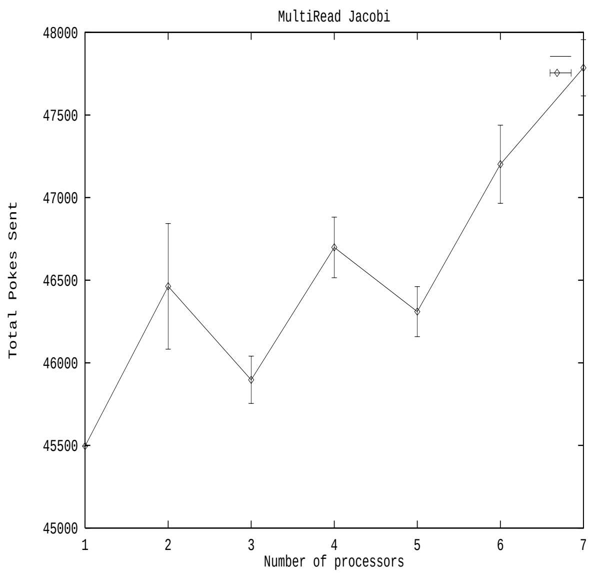


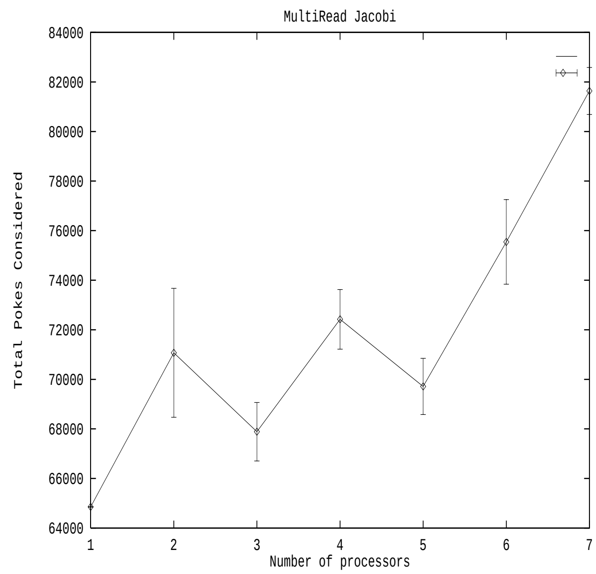


MultiRead Jacobi Iteration

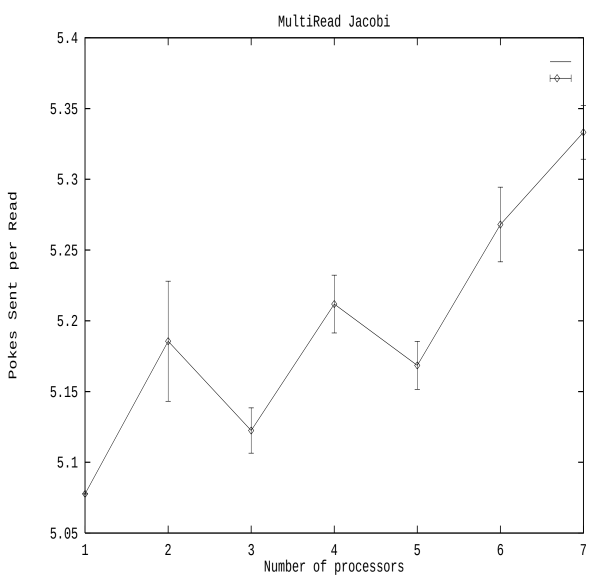
The graphs in this section correspond to the Chandy-Misra version of the Poker simulator running the version of the Jacobi Iteration algorithm using *MultiRead* construct. Once again, we only show the data for the compute phase, since we believe that the timings on the aggregate phase are too small for the data to be meaningful. Here, not only do we see the effect of null messages on the overhead, but overhead due to the input waiting rule is also present. Once again, the term “pokes” is used in lieu of the term “null messages” in the graphs.

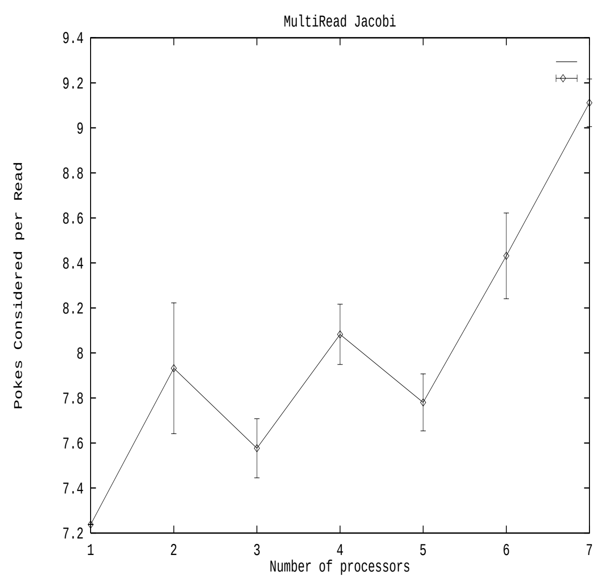
Once again we begin by considering the total number of null messages or “pokes”. On the left we show the total number of null messages which were sent, and see that it is no longer invariant with respect to the number of processors. Likewise, the total number of null messages which were considered also changes with the number of processors. This graph is shown on the right



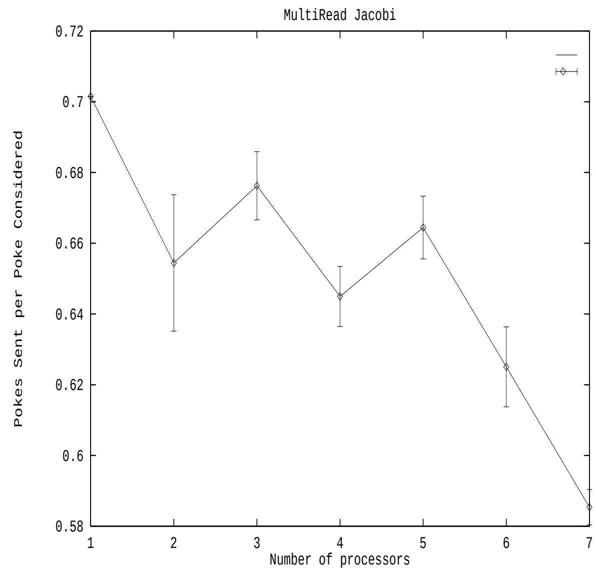


In the next two graphs the above data is simply divided by the number of reads, which is an invariant.

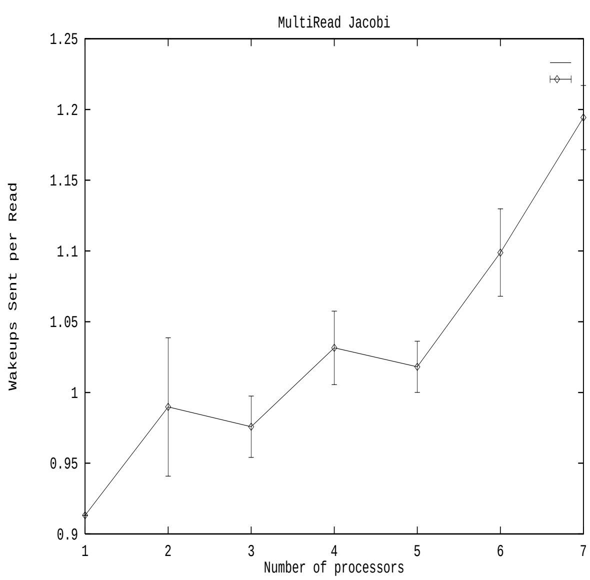


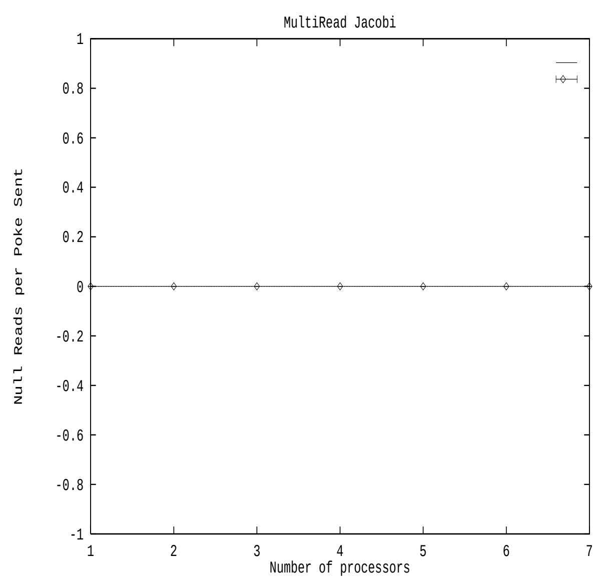


The next graph shows how often null messages were sent when they were considered. In this algorithm, there were many more times when null messages were considered but deemed unnecessary than in the Matrix Multiply algorithm.



The following two graphs involve the effect of null messages on the context switching of Poker processing elements. The left graph shows the frequency that reads by adjacent processors cause processing elements to be moved onto the ready queue when they are blocked waiting for I/O. The graph on the right shows the frequency that null messages cause extraneous context switches, i.e. that they cause processing elements to be moved to the ready queue and when activated, the processing elements are immediately swapped back out because no I/O messages had arrived. Note that there were no instances when extraneous context switches occurred in this algorithm, in contrast to the Matrix Multiply algorithm.





The final two graphs pertain to the overhead of the input waiting rule. The graph on the left shows the number of context switches per *MultiRead*, an indication of the amount of time which should be charged to the input waiting rule. The graph on the right shows the number of pokes sent per context switch, which can be used to asses the relative magnitude of the input waiting rule and null message overheads.