delim $$

Task Parallel Implementation of the JPDA Algorithm

Robert A. Wagner

Duke University
Department of Computer Science
Durham, NC 27708-0129

1. Introduction

JPDA problem

2. Previous Work

A sequential implementation of this algorithm is presented in [], and a version translated into PROTEUS in []. The work described in this paper began with these versions of the algorithm.

Efficiency of the running version was the primary driver behind much of the design work reported here. For concreteness, the IBM SP1 and SP2 machines were chosen as examples of massively-parallel machine architectures typical in today's marketplace. These machines consist of between 64 and 2048 RS/6000-based processor nodes. Each node of the SP1 consists of one such RS6000, running at 62.5 MHz; the SP2 nodes contains two such processors, one reserved for inter-node communication. The IBM RS/6000 can execute one fixed point, and one Floating Multiply Add [FMA] operation per cycle. Floating point loads are considered fixed-point operations, and can simultaneously update an index register. Each node contains its own memory, and each processor has its own on-chip cache, Inter-processor communication is done through a generic network, whose topology is typically not revealed to user-level programmers. A supplied library implements message-passing facilities whose performance is based on that of a simple point-point send/receive operation, Such an operation requires 30 usec latency per message, and transmits data at 8MBps. These facts are important in the algorithm design, for they limit the efficiency with which the JPDA algorithm can be implemented in parallel.

3. Sequential version

To get an estimate of the effectiveness of a parallel implementation, it is necessary to use as a base an efficient sequential implementation. Such an implementation may itself be fast enough to make a parallel implementation unnecessary for improving response time or throughput, although there may be other reasons for desiring such an implementation. One measure of the potential efficiency of a sequential implementation comes from estimating the arithmetic complexity of the algorithm. Here, it can be shown that the JPDA algorithm requires

FMA operations, for \$m\$ measurements and \$n\$ tracks.

One of the early steps in this effort was to verify that, for reasonable values of \$n=19\$, and \$m=20\$, the compilers and source code can be designed to achieve at least 50% of this peak efficiency. This effort examined the usefulness of causing certain loops to be unrolled, and partially executed prior to compilation, to eliminate the majority of the slow tests specified in the original algorithm. A tool was

developed which allows a program to designate certain variables and expressions as "known at preprocessing time". The tool creates a C program which executes the program, by printing statements which cannot be completely executed at preprocessing time, and executing those statements which are known at that time. The result is a new C program which can be compiled, and in which the statements and expressions whose values were known at preprocessing time have been eliminated. In particular, this allows the preprocessor to specify which elements of which arrays are to be loaded into scalar variables; the XLF compiler then assigns these scalars to registers.

A second tool was developed, to evaluate the operation of the algorithm against the machine's cache. This tool edits the C source code, replacing references to the important array (here called f[][][]) with calls to a routine, ref(), which simulates the cache operation. The current version of ref() simulates a 2-way set associative cache, of a size which is easily modified to fit a new machine. The editing process, in this case, also eliminates redundant references to the same cache line, which it detects by assuming that the algorithm's unrolled inner loop causes one cache line to be completely loaded into registers. Since a cache line holds between 8 and 16 double-words, and the machine provides 32 registers, this is appropriate on the RS/6000. It could be eliminated, if more accurate measurements are required.

These tools were used to assist in an accurate performance prediction. This prediction shows exactly how many loads and stores occur, by observing the total number of references reported by the cache simulator. It also shows the number of cache misses that occur. The two numbers together give a fairly accurate measure of the total time the fixed-point execution unit requires. For the JPDA algorithm this requirement, in number of cycles, in somewhat larger than the number of FMAs needed, and is estimated (from the few small cases actually simulated) to be 1.4 cycles per FMA. This suggests that, if the compiler in fact schedules instructions well, the sequential JPDA algorithm, version 3 (see below), should execute at 71% of the peak rate predicted above.

4. First parallel version

A task parallel implementation, on a message passing architecture, must be designed to perform a significant amount of computation between communication events. To specify the various possibilities, we observe that the elementary operation in the JPDA algorithm is the FMA operation:

```
f[el+1][j][a] += p[j][t] * f[el][j][a less t];
```

Several opportunities for parallelizing the JPDA algorithm exist:

1. The tasks indexed by distinct values of j are nearly independent; they could each run on a separate processor, with little, perhaps no, interprocessor communication. This is algorithm P0.
2. Suppose processor $p sub i$ holds all f[el][j][a] values that satisfy: bits x..x+np-1 of a are equal to i. Then processor $p sub i$ has immediate access to all f[]-values needed to update all f[]-values it holds, so long as t is not one of the bits in the range x..x+np-1. When t falls in the range above, some other processor holds the required f[]-values, and communication is needed to retrieve them. However, for fixed el and j, processor $p sub i$ can perform some $(n-np) * 2 sup n-np-1$ FMA operations, using data it holds. (Here, it is assumed that all processors can maintain their own copies of the p[j][t] matrix, which does not change during the algorithm.) When x is chosen to cause the selected set of bits to be highest order, we obtain algorithm P1.

   Performance of a parallel algorithm is estimated, by assuming that the algorithm proceeds in sequential phases, in each of which many processors execute in parallel. At the beginning of each phase, processors perform the communication the algorithm specifies; each processor computes using its own local data, and that obtained by communication until all processors complete. The processors implicitly synchronize (using the next phase's communication operations) at the end of each phase, so that no processor advances to phase y+1 until all have finished phase y.

   When this model of performance is applied to the current algorithm and machine, an estimate of performance for both algorithms above can be derived.

1. For algorithm P0, up to m processors can be used. Each distinct el step then requires time of

Since there are only el j-values in progress at step el, some of the processors will be idle, if $np>1$. These idle processors can compute f[el][0][a], if there are enough of them. Otherwise, communication is needed at the end of each el step. That communication would require one processor to receive f[el][0][a], for fixed el, and all a, from the processor dedicated to task $j=0$. This communication operation requires time

As an example, $TP0(19,20,20) ˜=˜ 2.25$ sec; $CP0(19,20,np) ˜=˜ 5.25 $ sec, although for $np=m$, sufficient idle processors exist that no communication is in fact needed. Note however, that communication time is far greater than computation time. This is a theme in all attempts to parallelize the JPDA algorithm for this class of parallel computer architecture.

To analyze algorithm P1, observe that processor $p sub i$ sends as many messages as there are 0's in the np-bit binary representation of i; processor $p sub i$ receives as many messages as there are 1-bits in i. Hence each processor processes np messages. Each message consists of the entire array another processor holds, or $2 sup n-np$ elements. We have:

When $np=6$, so there are 64 processors, $TP1(19,20,6) ˜=˜ 0.438 $ sec, and $CP1(19,20,6) ˜=˜ 9.37 $ sec. Computation time is small, compared to communication time.

Several attempts to improve the efficiency of solution P1 were made. One profitable approach uses data compression to reduce the number of bytes in the messages sent. A general compression scheme could be used, and would have to be assessed, based on the compression it achieves for lists of floating-point numbers, compared to the number of cycles per word needed for compression and expansion.

One simple compression scheme simply stores all floating-point quantities as single-precision 4-byte words, rather than the double-precision format which is native to the IBM RS/6000. This essentially halves the communication time, in all algorithms, neglecting the latency time. In this scheme, $CP1'(19,20,6) ˜=˜ 4.70$ sec, while leaving computation time unchanged.

Another possible approach tries to combine tasks, so that one processor performs several of the fixed high-k-bits tasks defined in algorithm P1. This approach, done manually, can reduce the amount of communication, by allocating tasks that would otherwise have to communicate to the same processor. Communication can also be reduced, by taking advantage of the fact that the same data is sent to several different other tasks. If some of the receiving tasks are allocated to the same processor, one message serves to satisfy the needs of all such tasks for incoming data.

This approach is labor-intensive. No general scheme for this allocation was found. It would be possible to provide some kind of graphics tool, which would allow a human to picture the possible clustering of tasks that might reduce communication. However, for large numbers of tasks, significant effort on a human's part would be needed, to achieve perhaps a factor of 2 improvement.

5. Second Parallel Scheme

A parallel scheme with improved performance was in fact designed, using the notion of assigning one processor the task of computing several f[el][j][a] values, from other f[]-values that processor holds. However, the idea behind this scheme causes one processor to first compute the updating sums for all a-values which match in the high-order np bits; the sum is then sent to another processor which has responsibility for all f[el][j][a] values such that a matches in its next np bits. This scheme requires a permutation operation within a given <el,j> step.

On first examination, the permutation operation seemed to require each processor to perform some $ 2 log sub 2 (P)$ communication operations, each transmitting $2 sup n-np$ array elements. If true, this would cause the new scheme to use more communication time than does P1. The analysis that suggests this timing requirement is based on using a Boolean hypercube as the virtual topology for the interconnection network; at stage q, processor $p sub i$ collects all data it holds destined for any processor $p sub k$ such that bit q of k does not equal bit q of i. This data is then sent to processor $ p sub {i ~~~ 2 sup q}$.

In the generic network supposedly implemented by the IBM SP1 and SP2, any processor can communicate directly with any other; furthermore, no network congestion occurs -- messages are supposedly not delayed due to other messages present at the same time. This may be an accurate model for this network. Apparently, the network delivers a packet of data of 255 bytes in 0.5 musec. However, there is a bottleneck at each processor, which reduces the rate at which bytes can enter the network to 8 bytes per musec. This means that a packet of data enters the network from any one processor only every 32 musec. The packet traverses $log sub 2 (P)$ network nodes in 0.5 musec, and so the probability of finding a packet produced by one processor at any given internal network node is 1/(64 lg P). When $lg P ~=~ 10$, this is about 0.0016. This suggests that the probability of packets from different processor arriving at the same internal network node is small. However, this should be tested experimentally.

If the non-congested network model can be believed, there is a potentially faster method of performing the required permutation operation. Processor $p sub i$ simply sends the data it holds destined for processor $p sub k$ directly to processor $p sub k$. The analysis, for one permutation when each processor holds a total of W words is:

The factor of 2 in the equation for $CP(W,P)$ arises because each processor must both send messages, and receive them.

Thus, the latency grows with increasing number of processors P, but the transmission time remains essentially constant. Since for our problem, $W ~=~ 2 sup n / P$, if latency can be controlled, communication time can scale downward with P.

5.1. Controlling latency

In the JPDA algorithm, up to m j-values may be simultaneously in progress. All of these tasks must communicate different data, but to the same processors. These communication tasks can be combined into the same messages. Thus, when a processor collects data destined for processor $p sub k$, the data should be collected from all el j-tasks currently in progress. This reduces latency per j-task by a factor of el.

5.2. The permutation algorithm

The algorithm presented in this section, P3, is an initial version of a promising design whose asymptotic complexity of both computation and communication is O($m^2 n 2^{n-1} / P$), where there are $P=2^{np}$ processors, n tracks, and m measurements. This algorithm's runtime scales linearly with the number of processors, if $n > 3 np$.

```
transpose(f);
for (j=0; j<el; j++) {
        for (c=(1<<np)-1; c >= 0; c-- ) {
                for (b=0; b<1<<(n-2*np); b++) {
                        f1[c][j][b] = 0;
                        }
                for (i=(n-np),k=1; k<=c; i++,k<<=1) {
                        if (k&c) {
                                for (b=0; b<(1<<(n-2*np); b++) {
                                        f1[c][j][b] += p[j][i]*f1[c-k][j][b];
                                        }
                                }
                        }
                }
        }
transpose(f1);

/* Assert f1{q}[c][j][b] = +/{ p[j][i]*f{q less i}[c][j][b] : i in q }  */

for (j=0; j<el; j++) {
        for (c=(1<<np)-1; c >= 0; c-- ) {
                for (i=(n-2*np),k=1; k<=c; i++,k<<=1) {
                        if (k&c) {
                                for (b=(1<<(n-2*np)-1; b>=0; b--) {
                                        f[c][j][b] += p[j][i]*f[c-k][j][b];
                                        }
                                }
                        }
                for (b=(1<<(n-2*np)-1; b>=0; b--) {
                        for (i=0,k=1; k<=b; i++,k<<=1) {
                                if (k&b) {
                                        f[c][j][b] += p[j][i]*f[c][j][b-k];
                                        }
                                }
                        f[c][j][b] += f1[c][j][b];
                        }
                }
        }
```

The transpose subroutine is:

```
transpose(float f[1<<np][M][1<<(n-2*np)]) {
/*  This takes advantage of the storage allocation rule of C that the
a sequence of storage location holds array elements whose right-most subscripts
vary most rapidly.  Thus, the messages each contain all f-elements from the
first el j-tasks, for any given value of c.  */
        for (c=0; c<(1<<np); c++ ) {
                send( &f[c]          /* 1st location */,
                        el<<(n-2*np)        /* record length */,
                        c                 /* destination processor */);
                }
        for (c=0; c<(1<<np); c++) {
                recv(  &buf, el<<(n-2*np), &from );
                copy( &buf, el<<(n-2*np), &f1[from] );
                }
        }
```

The "copy" subroutine is not shown. For efficiency, this subroutine should use double-precision floating-point loads and stores, on the IBM RS/6000. Possibly, the library "strcpy()" subroutine does so.

The arguments for the calls on the "send" and "recv" subroutines may not have the form required by those subroutines, as implemented on the SP1 or SP2. These calls should be regarded as "generic", specifying the intent to send/receive certain messages. Each message is assumed to be accompanied by the number of the sending processor.

In addition, it is possible that the stated design causes deadlock, because the internal buffers provided by the hardware and software cannot hold all the messages destined for one processor. If this occurs, the sending processor might be forced to wait (forever, in this case), until the receiving processor removes a message from its incoming buffer. Such a situation can be resolved, by causing each processor to use "non-blocking" send/receive calls, or by having processors send a few messages, then receive a few, and repeating. If this latter scheme is adopted, one should take care that the "f1" array is not disturbed during the second transpose call, before its contents have been transmitted. ,NH 3 Performance analysis of P3

The computation time, $TP3(n,m,np)$, and communication time, $CP3(n,m,np)$, of algorithm P3 are analyzed separately. It is possible that, on the SP2, computation and communication time can overlap.

During algorithm P3, processor $p sub q$ holds f{q}[c][j][b] initially, at stage el. This f-value corresponds, in the Mitre version, to value f[el][j][(q<<n-np) + (c<<(n-2*np)) + b]. That is, q, the processor number, equals the high-order np bits of "a" of the Mitre algorithm; "c" equals the next lower-order np bits of "a", and "b" corresponds to the lowest order $n-2*np$ bits of "a". After the initial transpose(), f1{q}[c][j][b] holds f{c}[q][j][b]. Algorithm P3 then performs $el * np * 2 sup np-1 * 2 sup n-2*np ~=~ np * 2 sup n-np-1$ FMAs. The second transpose() operation then permutes the f1 entries, which now hold partial sums, so that the Assertion of the algorithm is satisfied. Algorithm P3 then performs $el * (n-np) * 2 sup (n-np-1)$ additional FMAs.

The final equation for TP3() is derived by summing the component terms for $el~=~ 1 .. m$. This equation also assumes that the loop unwinding, and holding of values being updated in registers is possible in this version, and that the figure of 71% of peak arithmetic speed is still obtainable.

There are $2 m$ permutation operations needed during the entire algorithm. Each requires that $el * 2 sup n-np$ words be transmitted and received, so that:

Then $TP3(19,20,6) ~=~ 0.333$ sec, and $CP3(19,20,6) ~=~ 3.267 $ sec. The run-time scales inversely with P, the number of processors, so long as The value of 3.267 is somewhat better than $CP1'(19,20,6) ~=~ 4.70 $ calculated previously.

Using $np=7$, or 128 processors, produces $TP3(19,20,7) ~=~ 0.164$ sec, and $CP3(19,20,7) ~=~ 1.863 $ sec. Using $np=8$, or 256 processors: $TP3(19,20,8) ~=~ 0.083$ sec, and $CP3(19,20,8) ~=~ 1.393 $ sec.

5.3. Improvements to the transpose algorithm

The principle behind algorithm P3 indicates that f-values which match in np bits of the a-index are first collected, and fully combined This step is followed by another step, in which f-values which match in a different set of bits of their a-indices are brought together. This general principle can be extended, by reducing the time needed for the sorting operations which bring f-values together.

Two approaches to reducing this sort time have been investigated. The first involves splitting the available $2 sup np$ processors into two disjoint sets of processors, the "A" processors, and the "B" processors. "A" processors send data to appropriate "B" processors, and vice versa. This avoids having the same processors both send and receive data during one communication step. Analysis shows that this reduces the message latency term in the performance equation by a factor of 4, but leaves the transmission-time term unchanged. The second approach uses a two-digit radix sorting approach for the sort operation, accompanied by splitting the set of processors in two. This radix sort scheme improves performance when $np~>=~ n/2$, although processor efficiency is reduced, compared to that of algorithm P3.

5.3.1. Performance analysis

When processors are divided into two equal sets, and information is sent only from one set to the other, logically little of algorithm P3 changes. However, for any given number of processors $2 sup np$, the number of words each processor holds is clearly twice what it would be in algorithm P3. To offset this, each processor either sends or receives during one communication step, thus requiring only 2 such steps, rather than the 4 of algorithm P3. Also, each processor sends data to only $2 sup np-1$ processors on each step, halving the number of messages sent per step, and yielding a factor of 4 reduction in latency, when the reduction in number of communication steps is also considered. Calling this version algorithm P4, we have:

6.

This gives $CP4(19,20,6) ~=~ 3.15 $ secs, $CP4(19,20,7) ~=~ 1.633$ secs, $CP4(19,20,8) ~=~ 0.932$ secs. Thus, the communication time continues to scale down almost linearly with the number of processors, for a larger number of processors than occurs in algorithm P3.

Algorithm P4 ceases to yield reduced communication time when

For $n=19,~ m=20$, this minimum time occurs when $np=9$, or $np=10$: $CP4(19,20,9)~=~0.696$, $CP4(19,20,10)~=~0.809$. Algorithm P5 was developed to allow more than $2 sup n/2$ processors to be used in reducing run-time.

6.1. Radix sort transposition: algorithm P5

Algorithm P5 requires that the f_i arrays be allocated so that the original algorithm's a-index is partitioned, so that $a ~=~ q1 ~cat~ q2 ~cat~ c1 ~cat~ c2 ~cat~ b$. Initially, f0{A}{q1}{q2}[c1][j][c2][b] holds f[j][a]. Note that index j appears after the third "digit" of a. This allows the elements f0[c1][0..el][*][*] to occupy contigus memory locations, for any given c1 value, and thus be passed in one call to send(). After one radix-sort step, requiring time $30 ~+~ el * 2 sup n-np ~mu$-sec, f1{B}{c1}{q2}[q1][j][c2][b] = f[j][a]. Processor {c1}{q2} then combines the data it holds into f1, so that

```
f1{B}{c1}{q2}[q1][j][c2][b] =
        +/{ p[j][i+n-np/2] * f1{c1}{q2}[q1 less i][j][c2][b]
        : i in [0..np/2] }
```

A second radix-sort step moves f1{B}{c1}{q2}[q1] to f1{A}{q1}{q2}[c1], and a third radix-sort step moves f{A}{q1}{q2}[c1] to f2{B}{q1}{c1}[q2]. An update step, performed by {B} processors on index q2 of f2, followed by the inverse of the third radix-sort step leaves:

```
f1{A}{q1}{q2}[c1][j][c2][b] = +/{ p[j][i] * f[j][a less i]
                                : i in [n-np/2..n] }
f2{A}{q1}{q2}[c1][j][c2][b] = +/{ p[j][i] * f[j][a less i]
                                : i in [n-np..n-np/2-1] }
```

The {A} processors then combine f1, f2, and the f-values selected by bits [0..n-np-1] (indices c1, c2, and b) to complete the update of f. It should be noted that both messages sent from {A} to {B} processors can be sent, before any computation is done; the return messages can be sent after the first two combining steps are complete, and the {A} processors can perform most of their required computations during the same time the {B} processors perform their computations. Some message-combining may also be possible, reducing the latency time by a factor of 2. This refinement of algorithm P5 has not been incorporated in the analysis below.

6.1.1. Performance of algorithm P5

The communication time of algorithm P5 dominates the computation time. $CP5(n,m,np)$ can be compared with $CP4(n,m,np)$, by noting that P5 performs twice as many communication steps, as does P4. However, P5 sends only $2 sup (np-1)/2$ messages per step (if np is odd).

$CP5(19,20,8) ~=~1.585$ secs, $CP5(19,20,9) ~=~0.817$ secs, $CP5(19,20,10) ~=~0.447$ secs, $CP5(19,20,11) ~=~0.271$ secs. Thus, run-time continues to decrease, even when 2048 processors are used.

7. Algorithm development methodology

The various algorithm variations explored, which ultimately lead to algorithm P3, were all obtained by performing various transformations on the original algorithm, which could in principle be done using an automated system, which would check each transformation used to see that it preserved the original meaning of the algorithm. The performance of each algorithm variant produced was evaluated, using the algorithm's statement to obtain estimates of the arithmetic computation, and communication cost (size and number of messages sent and received) at each parallel step, for the slowest processor.

The performance evaluations resulted in a tentative conclusion that the communication bandwidth limited the parallel efficiency of each algorithm quite seriously. This conclusion suggested searching for algorithm variants which would reduce the total number of words transmitted by all processors during the computation. Communication latency proved fairly easy to manage, even when the number of messages increased linearly with the number of processors.

During the performance evaluation phases, techniques for improving the computation time component of total run-time were also explored. These techniques, which involve "blocking" inner loops, to remove unneeded tests, and to hold values which are being updated in registers until all updating operations are complete, were useful in assessing the speed of a purely sequential version of the JPDA algorithm, and would have some value in a parallel version of the algorithm. Assuming these techniques are used, a modest number of processors can divide the necessary work so that each processor's computation time is quite small (0.333 sec, for 64 processors on the running example). However, communication time is still dominant (3.267 sec for the same example).

It appears that a flexible approach to algorithm refinement should be adopted, one which allows the results of early estimates of performance or other algorithm characteristics to guide later choices of possible algorithm variant construction. It should be noted that a great deal of effort can be saved by performing all such explorations of algorithm variants at a very undetailed, possibly unexecutable level. Much effort is expended whenever an algorithm sketch is refined to a level of detail where it can actually execute, and produce useful results. If simulation of partial algorithms, and theoretical calculations of algorithm characteristics like performance and correctness can be used, a small amount of programming effort can produce large increases in algorithm quality.

In the study herein reported, simulation of some parts of the memory hierarchy were used to predict cache performance. At the same time, the simulator was able to demonstrate that certain indexing transformations adopted to avoid cache conflict misses indeed preserved the intent of the original algorithm, for the cases tested. All of the performance equations reported were derived theoretically, not measured. Furthermore, those equations which represent communication performance should be experimentally verified, to ensure that the hardware interconnection network conforms to the assumption that the time for a message to traverse the network is independent of other messages present in the network at the same time. This prediction could be verified by implementing and executing the current algorithm, or could be tested, using a careful experimental protocol. The assumption is critical to the choice of algorithm for the parallelization of the JPDA algorithm, and probably for others as well.

8. Conclusions

Several task-parallel versions of Mitre's [] JPDA algorithm have been derived which are suitable for use on massively parallel message-passing architectures, such as the IBM SP1 and SP2. These machines have many fast processing nodes, interconnected by a switch which exhibits no network contention, in theory, but has message-passing performance characterized by a several μ-sec latency per message, and a time to receive one floating-point number which is some 30 times the time needed for one floating-point operation. These hardware characteristics produce parallel versions of the JPDA algorithm which are limited by the communication bandwidth, if the number of processing nodes to be used is much greater than m, the number of measurements.

The best algorithm among those investigated here depends on the problem size, and the number of processors used. When the number of processors $NP = 2^{np}$ is less than m, a simple

parallel algorithm, in which each j-task is performed by a separate processor, and idle processors compute task j=0 redundantly, seems to be best. It yields run-time for the running example ($n=19,~ m=20$) of 2.13 secs, using 20 processors. For larger number of processors, this run-time is not improved until 128 processors are used, and algorithm P4 is employed. $CP4(19,20,7) ~=~ 1.633$ secs, $CP4(19,20,8) ~=~ 0.932$ secs, $CP4(19,20,9)~=~0.696$ secs, so that algorithm P4 is best, when $np~<=~n/2$. For larger values of np, algorithm P5 improves performance most: $CP5(19,20,10) ~=~0.447$ secs, $CP5(19,20,11) ~=~0.271$ secs. Algorithm P5 has only been outlined, so it is possible that its performance is not in fact as good as predicted here. Also, computation time for algorithms P4 and P5 has not been included in the comparison.