Evaluating the Trade–Offs in the Parallelization of Probabilistic Search Algorithms *

R. L. Carceroni[‡], **W. Meira Jr.**[‡], **R. Stets**[†] **and S. Dwarkadas**[†]

[†] Department of Computer Science
University of Rochester
Rochester, NY 14627–0226 USA

[‡] Depto. de Ciência da Computação
UFMG – Caixa Postal 702
B. Horizonte, MG 30123–970 Brazil

Abstract

In this work, we propose a speculative parallelization strategy for Probabilistic Search algorithms. We design a parallel version of one such algorithm for a real–time computer vision problem with many practical applications. The implementation is performed on a cluster of eight DEC AlphaServer 2100 4/233 machines connected via a DEC Memory Channel network. Four types of run–time systems are tested: Hardware–coherent Shared Memory (HSM), Distributed Shared Memory (DSM) with software coherence (Cashmere-2L), Reflective Shared Memory (RSM), and message passing (Digital PVM). A run–time system that is normally not the most efficient among these four (RSM) is found to be the best choice for our combination of algorithm and architecture. This shows that algorithmic parallelization and the selection of an implementation environment mutually interfere with each other and must be performed in a synergistic manner.

Resumo

Neste trabalho propomos uma estratégia de paralelização especulativa para algoritmos de Busca Probabilística. Um algoritmo de Busca Probabilística paralelo é implementado para o Problema da Correspondência, cuja resolução é necessária em várias aplicações de visão computacional com requerimentos de tempo real. Essa implementação é executada em um *cluster* de oito DEC AlphaServer 2100 4/233 conectadas por um DEC Memory Channel. Apresentamos e discutimos resultados com quatro sistemas de tempo de execução: Memória Compartilhada com coerência por *Hardware* (HSM), Memória Compartilhada Distribuída (DSM) com coerência por *software* (Cashmere-2L), Memória Distribuída Refletiva (RSM), e troca de mensagens (Digital PVM). Dentre estes, o sistema de tempo de execução que normalmente é o menos eficiente (RSM) apresenta o melhor desempenho, mostrando que a estratégia de paralelização do algoritmo e a seleção do ambiente de execução se influenciam mutuamente e não podem ser consideradas isoladamente.

*This work is supported in part by CAPES process BEX 0591/95-5; CNPq grant 200.862/93-6; NSF grants CDA–9401142, CCR–9319445, CCR–9702466, CCR–9705594 and CCR–9510173; and an equipment grant from Digital Equipment Corporation's External Research Program.

1 Introduction

Parallelism has played a central role in the area of high–performance computing. Historically, it gained momentum with the development of several high–performance shared–memory machines. However, recent improvements in networking technology have made distributed–memory systems a very attractive low cost alternative to such traditional parallel architectures.

A big challenge motivated by this technological revolution is the development of software systems that allow users to express all the inherent parallelism of their applications in an easy and architecture–independent way, without any major drawbacks in terms of performance. Message passing protocols seem to meet the efficiency requirement quite well, but require users to manage shared data migration and replication explicitly, which increases development time and decreases reliability and portability. Distributed–Shared–Memory (DSM) systems, on the other hand, seem to be a natural answer to these problems, but their relative efficiency in many architectures is still questionable due to problems such as excessive communication caused by false sharing.

Although this trade–off between ease of programming and efficiency is valid for many classes of applications, the selection of an ideal paradigm for the implementation of applications with critical performance requirements still seems to be highly dependent on the particular structure and data–access patterns of the algorithms involved, and on the characteristics of the underlying run–time system and architecture.

In this paper, we study the parallelization of Probabilistic Search algorithms, a generic technique employed for solving NP–hard problems. We use the Correspondence Problem, which is a key component in several relevant real–time computer vision tasks, as a representative of combinatorial optimization problems that can be solved by using Probabilistic Search. We start by describing the problem and proposing a speculative parallelization strategy for it. We then describe the implementation of this strategy in four run–time systems, the performance results in each of them, and discuss the issues involved in choosing the run–time system that best supports our parallelization strategy.

2 Probabilistic Search

Probabilistic Search (PS) is a technique that resembles *Branch and Bound* (B&B) [5] in the sense that it minimizes the total number of alternatives that must be verified in order to perform an exhaustive search (*i.e.,* a search that is guaranteed to find a solution if one exists) in the solution space of a given combinatorial optimization problem. The basic idea is to partition this search space recursively into smaller sub–spaces and check these sub–spaces for feasibility with certain restrictions imposed in the problem definition. This way, sub–spaces that are entirely infeasible can be discarded in an early stage of the partitioning process, reducing the total computational cost of the search.

Both PS and B&B are useful to deal with problems for which no polynomial–time algorithm is known. In some of these problems, although it is not always possible to find a feasible solution in polynomial time, there is an efficient way of determining which sub–spaces are more likely to contain at least one feasible solution. These are the problems that can be successfully tackled with PS. Unlike B&B, PS assigns a feasibility probability to each sub–space that is under consideration as the possible location of at least one feasible solution. Then, whenever PS is faced with a decision between two or more sub–spaces with the same *cardinality*, it uses these probabilities and chooses to work on the sub–space with highest probability first.

Another difference between PS and B&B is that the former is much more difficult to parallelize. The major obstacle to the successful parallelization of PS algorithms is the fact that checking whether a given sub–space is completely infeasible or not usually changes the values of the feasibility probabilities in regions of the search space that are not necessarily in the neighborhood of the current search point. Note that PS is not a randomized technique and thus it must explore the search space in the exact order indicated by the probabilities assigned to its sub–spaces. In the general case, the identity and even the approximate location of the next sub–space to be partitioned depends on the results obtained in the feasibility verification for the current sub–space. This results in data dependencies between successive computation threads of PS algorithms.

In order to cope with this difficulty, we propose a *speculative parallelization strategy* for PS algorithms. In general, most of the sequential computation time of PS is spent in the verification of the feasibility of the search sub–spaces under consideration. Therefore, a successful parallelization strategy must allow multiple such verification steps to be at least partially overlapped in time. According to our proposed strategy, rather than waiting for the result of the current verification to update the probabilities and then choosing the next sub–space to be verified (and possibly partitioned), a parallel PS algorithm immediately speculates an outcome for the current feasibility check, updates the probabilities, and starts working on what would be the next sub–space to be verified, if its speculation were correct. If the parallel PS algorithm can perform several successful speculations in sequence, then it executes multiple verification steps concurrently and the overall execution time is significantly reduced. The disadvantage of this strategy is that whenever the outcome of a feasibility check is not in agreement with what was speculated, the PS algorithm must roll back, restore the appropriate values for the probabilities, and discard the work performed on all the successive sub–spaces whose exploration was incorrectly determined. Thus, some care is needed in the speculation process, so that the benefits of the parallelization can outweight the extra cost of the wasted computation and the roll–backs.

3 The Correspondence Problem

In order to show that the strategy described in the previous section can be successful in practice, we use it to parallelize a known PS algorithm for a combinatorial optimization problem of fundamental importance in the field of computer vision, with applications in visual navigation, automated surveillance, hand–eye coordination and augmented reality

[7, 4]. To understand why this problem is so important let's consider how a typical *active–vision* system works, as illustrated in Fig. 1.

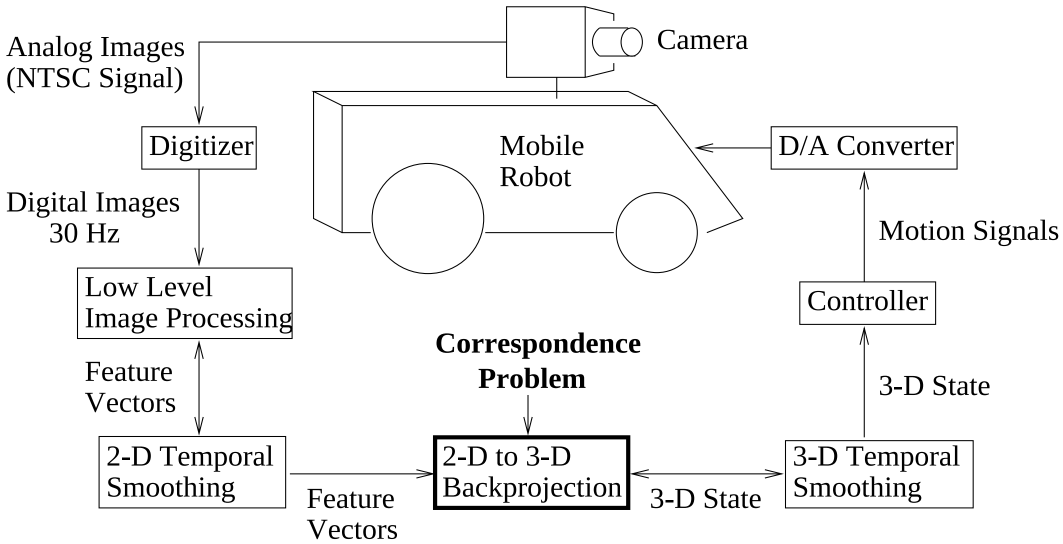

Figure 1: Components of a typical active–vision system.

Analog images collected by one or more cameras placed at a mobile platform are digitized at a frequency of 30 Hz. Initially, this large amount of incoming information must be processed (usually with specialized vectorial hardware), so that relevant features such as brightness edges can be isolated. After that, each incoming scene can be described as a vector of 2-D measurements. But it is necessary to generate 3-D descriptions of these scenes somehow, so that the relative pose (position and orientation) of the mobile platform with respect to targets or landmarks of interest can be determined, and used to calculate adequate motion patterns. This 2-D–to–3-D backprojection step is frequently based on the prior existence of a *model* for the 3-D scene geometry. Its execution depends on the ability to establish pairwise correspondences between the features in the data–driven 2-D scene description and the features in the 3-D geometrical model.

This is exactly the problem known as the *Correspondence Problem* (CP): given M 3-D features (scene model), N 2-D features (image) and a projective transformation with some unknown parameters that maps the 3-D space onto the 2-D space (camera model), find a maximum–cardinality set of pairwise correspondences between 2-D and 3-D features, so that all the selected 3-D features are projected onto the corresponding 2-D features with a unique set of values for the unknown parameters. In general, this problem has a single solution and the set of parameters that allows the maximum number of correspondences to be established is exactly the true pose (position and orientation) between the camera and the scene.

The most traditional approach for solving this problem is to perform a brute force search, where every possible set of matches is checked for feasibility individually [1, 4]. Unfortunately, the cost of such methods is unaffordable for real–time applications. David Lowe [7] proposed an alternative PS algorithm that leads to dramatic savings with respect to the previous brute force approaches. In the following subsection, we explain this algorithm, relate it to our previous discussion of PS, and show how to parallelize it.

3.1 A Probabilistic Search Algorithm and Its Parallelization

In order to characterize a PS algorithm for a given combinatorial optimization problem, it is necessary to define: how the search space can be recursively partitioned; how the resulting sub–spaces can be checked for feasibility; and how feasibility probabilities can be assigned to individual sub–spaces. We discuss each of these steps below:

In Lowe's algorithm for the CP, a search sub–space corresponds to a partial solution in which (conjectured) correspondences are imposed only for $m < M$ model features. Partitioning a sub–space amounts to conjecturing the correspondence for an additional model feature (*i.e.,* increasing m by one) and a complete solution is a set of pairwise matches between model and image features with size $m = M$ (the algorithm assumes that $M \leq N$).

Checking a sub–space (*i.e.,* a partial set of matching pairs) for feasibility involves a careful geometrical analysis that is only sketched here. Recall from the previous section that in the CP the relationship between 3-D and 2-D features (*i.e.,* whether they match or not) is determined by a parametric projective transformation whose (unknown) parameters correspond to the pose of the camera with respect to the rest of the scene. Thus, the CP has a dual nature: given a certain camera pose, it is straightforward to simulate the imaging transformation and then to determine all the resulting pairwise matches between 3-D and 2-D features; and, on the other hand, given a certain set of correspondences, it is also possible (though relatively expensive, computationally) to determine a pose that fits all the projections of the 3-D features to their 2-D image matches, in a least–squares sense [6]. The difficulty of the CP lies exactly in the fact that neither a solution in its discrete domain (the correspondences) nor a solution in its continuous domain (the pose parameters) are known. However, if a conjectured discrete solution with *at least four matching pairs* is available, its feasibility can be determined in the process of computing the dual least–squares pose solution, by verifying whether the residual of the least–squares fitting is below a certain predefined threshold, as explained in [7].

Finally, in order to complete a definition of a PS algorithm for the CP, it is necessary to determine how to assign feasibility probabilities to partial solutions in its discrete domain and how to search its discrete solution space according to these probabilities. In Lowe's algorithm, this search is performed in two completely distinct phases: the first one tries to find an interesting partial solution in a breadth–first–like style and the second tries to expand this partial solution in a depth–first way, in order to transform it into a complete one.

Note that, as mentioned in the last paragraph, partial solutions with less than four matching pairs can *not* be checked for feasibility. So, initially, Lowe's algorithm performs a probability–guided search in the space of $\mathcal{O}((M \times N)^4)$ possible partial solutions with size equal to four, until it finds one such partial solution that is ruled as possibly feasible. In order to assign numeric probabilities to all partial solutions of size four, Lowe's algorithm assumes that a reasonable initial estimate for the unknown camera pose can be obtained through temporal smoothing and extrapolation (keep in mind that this technique is used in the core of a real–time visual servoing loop). Then, the imaging process is simulated, generating a predicted *synthetic image*. This synthetic image is compared against the image actually digitized and an error measure with two degrees of freedom is computed for each of the possible $M \times N$ matching pairs composed by one model feature and one image feature. A bivariate normal distribution for the two parameters of this error measure is then used to assign an initial numeric probability for each pair. Finally, the feasibility probability for any given set of matching pairs is defined as the product of the individual probabilities assigned to each of its elements.

All the probabilities used in this first phase are based on an *a priori* estimate for the camera pose and for this reason we call them *a priori* probabilities. Whenever the search algorithm verifies that a certain set of matching pairs is not feasible, it is not possible to compute a better pose estimate. However, the individual feasibility probabilities for all the matching pairs involved must necessarily be decreased, so as to indicate that at least one of them is an outlier. Lowe's algorithm adopts a heuristic penalization scheme: the *a priori* probabilities for all possible matching pairs are stored in a matrix of size $M \times N$; whenever a partial solution is ruled as infeasible, the elements associated with its pairs in this *a priori* matrix are multiplied by a number in the range $(0, 1)$.

Whenever a partial solution of size four is ruled as possibly feasible, Lowe's algorithm enters a second phase that resembles a depth–first search towards a complete feasible solution. Note that the fact that a possibly–feasible partial solution has been found implies that a better pose estimate is available (namely, the least–squares fitting computed in the verification step). So, the algorithm momentarily abandons the *a priori* probabilities and computes a set of *a posteriori* probabilities, that are kept in a separate $M \times N$ matrix. These probabilities are then used to determine which is the next best matching pair to be added to the current partial solution at each step. Then, the expanded set is checked for feasibility again and the *a posteriori* probabilities are recomputed, completing a cycle.

If the partial solution being expanded ever gets infeasible in this phase, our implementation performs an individual check for each of its elements, in order to determine which of them is most likely to be an outlier. Then, we search the probabilistic neighborhood of the current solution for an adequate replacement to this conjectured outlier (*i.e.,* another pair that makes the resulting partial solution possibly feasible again). If one such replacement is found, the search can proceed normally towards a complete solution. Otherwise, a backtrack occurs. In this case, the *a posteriori* probability matrix is discarded, the *a priori* matrix is restored and properly updated, and the search reverts back to the first phase.

Our parallelization of the first phase of the search algorithm described above is based on the speculative strategy presented in Section 2. Preliminary experiments showed that in most of the computationally demanding instances of the problem at hand, a lot of initial sets with size four must be checked and rejected until one of them can be expanded towards a possible complete solution. Every time one such set is generated, our parallel algorithm immediately updates the *a priori* probability matrix, assuming that this set is going to be ruled as infeasible. Then our algorithm uses the resulting probabilities to speculate the next set to have its feasibility verified and immediately starts this verification in parallel with the current one, completing a cycle. In the relatively unlikely (but very important) case in which a set with size four is found to be (potentially) feasible, the incorrectly speculated sets are discarded, appropriate values for the *a priori* probabilities are restored, and the search enters its second phase.

This second phase is mostly inherently sequential at task level. However, good parallelization opportunities appear in the points where the current partial solution being expanded is ruled as infeasible. In this case, both the tests to determine which of the pairs currently selected is most likely to be an outlier and the search for a replacement involve performing multiple feasibility checks that we fully parallelize in a non–speculative way.

In order to simplify the coordination of the whole search process, we adopt a centralized approach in which all the probability matrices are read and updated by a unique master process. This master process is also responsible for choosing which sets of matching pairs are going to be checked for feasibility and for sending work requests to a set of slave processes. The slave processes then perform the computationally intensive feasibility checks and return the results of these checks back to the master. Although the accesses to the probability matrices are inherently sequential, a distributed–control approach in which these matrices migrate between several owner processes would also be possible. However, preliminary tests indicated that the contention for this exclusive–access shared data structure would result in extremely poor performance and this approach was immediately discarded.

The protocol for communication between the master and the slaves starts in a non–blocking mode, in which the master repeatedly sends requests of work to the slaves and then tries to retrieve answers to requests posted so far, even if they arrive out of order. Of course, the out–of–order results based on speculative guesses can not be used until these guesses are confirmed. Whenever one such result is retrieved from the communication structures, it is stored in a private queue at the master, where it may be immediately accessed when needed. If eventually some speculative guess is found to be false, all the results queued after it are discarded from the queue. At certain points in its execution, the master process can not proceed and generate new feasibility–check requests before the answers to one or more pending requests are received from the slaves. In these situations, the master may either use a blocking primitive or keep polling the communication structures, depending on the availability and on the relative efficiency of these options in the different implementation environments chosen.

4 Parallel Processing Environment

In this section we describe our experimental environment. We start by describing the hardware resources available and then we describe the four run–time protocols used in our evaluation.

4.1 Hardware

Our hardware platform consists of eight DEC AlphaServer 2100 4/233 machines which are connected by a DEC Memory Channel network. Each of these bus–based multiprocessors houses four 233 MHz 21064A Alpha processors and 256 MB of memory. The Memory Channel is a reliable, in–order, low–latency, remote–write–access network. One–way message latency is 5.2 μs.

The Memory Channel [2] also provides user–level messaging capabilities through a very unique programming interface. Through special initialization calls, processes attach to address space regions reserved for the Memory Channel. The attach call must specify the message direction, *i.e.,* `TRANSMIT` or `RECEIVE`. If a region is attached as the former, then any stores to the region are automatically propagated to processes that have mapped the region as `RECEIVE`. All `RECEIVE` regions are pinned down in physical memory so that the network interface can quickly transfer incoming data to its destination. `TRANSMIT` regions are simply uncached writes to I/O space. As a result, values in `TRANSMIT` regions can not be read. These regions can only be accessed through basic store instructions. The `RECEIVE` regions, however, are in physical memory and can be accessed through load and store instructions. However, store instructions do not produce messages.

4.2 Run–Time Protocols

Hardware–Coherent Shared Memory

The HSM protocol simply runs on the hardware–coherent shared memory available inside one of the AlphaServer machines. Synchronization and communication are implemented entirely through shared memory and hardware coherence. Our hardware limits this protocol to executions of one to four processors. The user sees it as a library with routines for shared–memory spin locks, tree–based barriers, and synchronization via flags.

Cashmere-2L

Cashmere-2L [8] is a state–of–the–art software–based DSM system. The system is designed for operation on clusters of multiprocessors connected by a network with capabilities similar to the Memory Channel. Like HSM, the protocol is implemented as a user–level library which must be linked with the applications. A special compilation script is required to instrument applications and to link the Cashmere library. The instrumentation adds four *polling* instructions to the start of every loop. These polling instructions check for incoming messages and jump to a message handler if necessary.

The library contains synchronization and virtual memory fault handling routines, which provide the main entry points to the protocol. The protocol itself can be logically divided into inter–node and intra–node levels. The inter–node level implements "moderately" lazy release consistency, while allowing multiple concurrent writers. As data accesses are tracked via virtual memory page faults, the coherence unit is naturally a virtual memory page. The protocol maintains a distributed *page directory* that stores page–state information including a page's current sharing set and its associated *home node*. The home node contains the page's master copy, which is kept updated according to the definition of release consistency. When a processor first attempts to modify a page, the protocol will construct a *twin* of the page. Subsequently, at a *release* operation, the page is compared with the twin to uncover all modifications. The modifications are flushed to the appropriate home node(s) and *write notices* are sent to processors in the page's sharing set through the Memory Channel's remote–write–access capability.

As mentioned, Cashmere-2L exploits our platform's available hardware coherence. All processors inside a node share the same page frame for a single page of shared memory. Consistency of the page modifications is then maintained by the hardware. By virtue of this underlying hardware coherence, any software coherence transaction performed by one processor is actually performed on behalf of all processors on the node. In other words, the result of the transaction is reflected on all processors. Thus often coherence transactions from multiple processors on the node can be coalesced or a single transaction from one processor can obviate the need for transactions from other processors.

Reflective Shared Memory

Reflective Shared Memory (RSM) is a very straightforward implementation of DSM. Any access to shared memory is automatically propagated to all other nodes in the system. Our implementation of RSM simply maps the entire shared memory address space into Memory Channel space at initialization time. The shared memory is actually mapped twice: once as a `RECEIVE` and once as a `TRANSMIT` region. The two regions are separated by a fixed offset. During the compilation phase, application executables are instrumented so that all writes to shared memory are duplicated. One write places the data in the `RECEIVE` region and the second write stores data to the `TRANSMIT` region which results in an immediate broadcast of the data. Applications with high read latencies may benefit from RSM, although extensive computation in shared memory may overwhelm the available network bandwidth and limit the application's scalability.

PVM

Digital PVM has been written to exploit the user–level messaging capabilities of the Memory Channel [3]. Outgoing messages are copied directly to a `TRANSMIT` region and thus are dispatched very efficiently. On the receiver side however, the messages can not be handled with the same efficiency. Incoming data is deposited by the Memory Channel into a predefined `RECEIVE` region. The PVM library must then copy the data from this region to the final location. This *double–copy* is necessary because `RECEIVE` regions can not be re–located. Interestingly, the PVM library also uses an adaptive back–off spin loop to poll for incoming messages. The library polls for a certain number of cycles and then yields the processor. The number of polls performed decreases in each subsequent cycle until a timeout threshold is reached.

5 Experimental Evaluation

In order to evaluate the effectiveness of our parallelization scheme, we initially implemented a sequential version of Lowe's algorithm for the CP. All the geometrical analysis needed in the feasibility checks and in the probability assignments was implemented in Matlab, and then translated to C, with the Matlab Compiler 4.2. The search procedure itself was implemented directly in C. The resulting code was compiled with the optimization option -O4. All the results described here are based on a test–case with $M = 12$ and $N = 36$, in which 225 conjectured partial solutions must be checked for feasibility before a true solution is found, in the sequential case.

Our first implementation environment was shared memory, because of its simple programming paradigm. In our shared memory implementation, the communication between the master and each slave is done through two shared buffers: one is used to issue work requests to the slave and the other is used to store the answers to these requests. In the test case selected the size of each of these two per–slave buffers is much smaller than the unit of coherence (8 KB). The master–slave synchronization is managed with a vector of flags, one per slave. Whenever the master wishes to receive help from a specific slave, it writes a request in the request buffer and then sets the flag associated with that slave. As soon as this operation is propagated through the Memory Channel, the slave is awakened from a blocked state, performs the required work, writes the results to the results buffer, and then resets its flag. Finally, whenever the master wants to collect answers, it keeps polling the flags of the active slaves, in a round–robin fashion. When the state–change in one of these flags is detected, the master effectively synchronizes with the slave and copies the results from shared memory to data structures in its private memory.

Initially, we used the HSM run–time library, which is simply hardware–coherent shared memory. The measured speedups, as a function of the number of slaves, are plotted in Fig. 2. In this same figure, we also display the results obtained with other choices of run–time systems, as well as a curve showing the maximum speedups predicted by Amdahl's Law, given the fact that the profiling of Lowe's PS algorithm revealed that the inherently sequential fraction of its computation–time is 11%. Hardware coherence obtains results very close to optimal speedups for this particular algorithm, but the number of total processors is limited by our platform to four (three slaves).

To leverage the full number of available processors, we used Cashmere-2L. As we mentioned in the previous section, Cashmere-2L is a natural choice for the architecture at hand, since it allows one to explore the availability of hardware coherence in all the interactions between processors of the same node, and still provides additional support for inter–node interactions. The results displayed in Fig. 2 show that when the execution is limited to a unique node, the performance of Cashmere-2L closely matches that of pure hardware coherence, as expected. However, as soon as inter–node interactions are needed, the gap between the speedups obtained with Cashmere-2L and those expected according to Amdahl's Law quickly increases. The performance achieves a maximum level with twelve slaves and tails down after that.

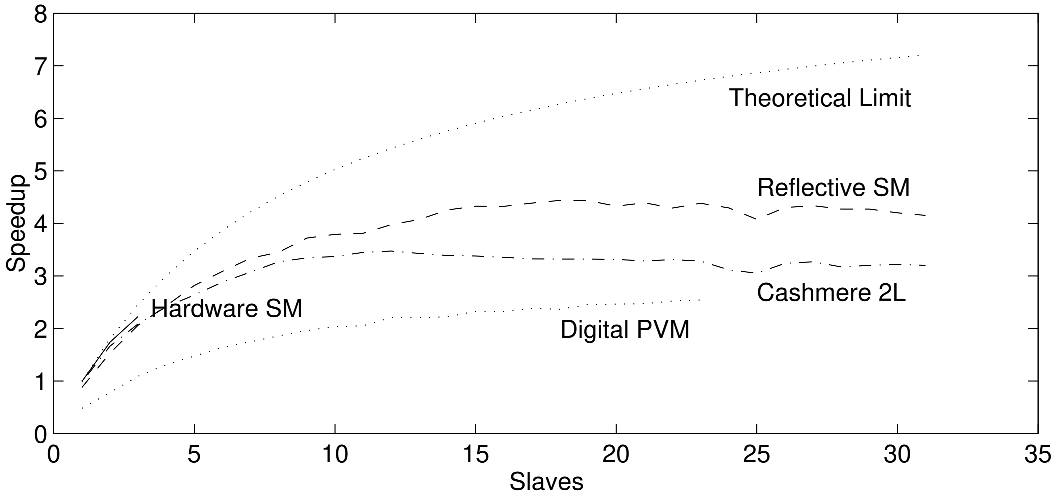

Figure 2: Speedups: elapsed sequential time divided by the elapsed time of the parallel execution, as a function of the number of slaves.

At a first glance, we attributed the modest gains obtained with the use of additional nodes to the fact that the size of the coherence unit in Cashmere-2L (8 KB) is much bigger than the actual amount of data communicated in a typical master–slave interaction of our parallel PS algorithm (about 400 bytes, for the test case selected). The natural way of avoiding this conjectured extra communication overhead seemed to be to migrate our implementation to a message passing environment in which the granularity of the communication is smaller, such as Digital PVM.

In our message passing implementation of the parallel PS algorithm, each slave communicates with the master through a private *mailbox*. Whenever the master is in a state in which there is more work to be done regardless of whether additional results are received from the slaves, it uses a non–blocking receive primitive to collect incoming results. When it finally reaches a state where the computation can not proceed until some additional result is received, it uses blocking receives, which were found to be slightly more efficient than their non–blocking counterparts. The slaves always use blocking receives.

The timings obtained with this implementation (Fig. 2) show that its performance scales relatively well as the number of slaves is increased. However, the method used to poll for incoming messages and the double–copy operation generate an unacceptably high overhead, even if a few processors are used. Instrumentation later inserted in the user–level code revealed that regardless of the number of processors used, the computation times for the threads corresponding to the feasibility verification step roughly doubled with respect to the sequential implementation, both in the master and in the slaves. A likely cause for this increase is the use of the adaptive back-off spin loop approach in the polling for incoming messages, which may be forcing the sequential threads to be de–scheduled and re–scheduled repeatedly. However, due to the proprietary nature of Digital PVM, we could not use protocol–level instrumentation to confirm this. We also tried to use a traditional TCP–based PVM [9] but the preliminary results obtained with it were so poor that it was immediately abandoned.

At this point we performed a careful instrumentation of our DSM implementation, both at user and protocol levels. The execution–time in the master processor was broken down into five major categories, as shown in Fig. 3: (Verif) the time spent in "residual" feasibility verification operations that take place in inherently sequential parts of the search and thus can not be distributed to the slaves; (Probab) the time spent in the computation of the feasibility probabilities, also inherently sequential at task level; (Comm) the communication overhead, *i.e.,* the time spent transferring data to or from the buffers in shared memory; (Synch) the overhead associated with synchronization primitives; and (Wait) the time spent in the busy–waiting loop that polls the flags.

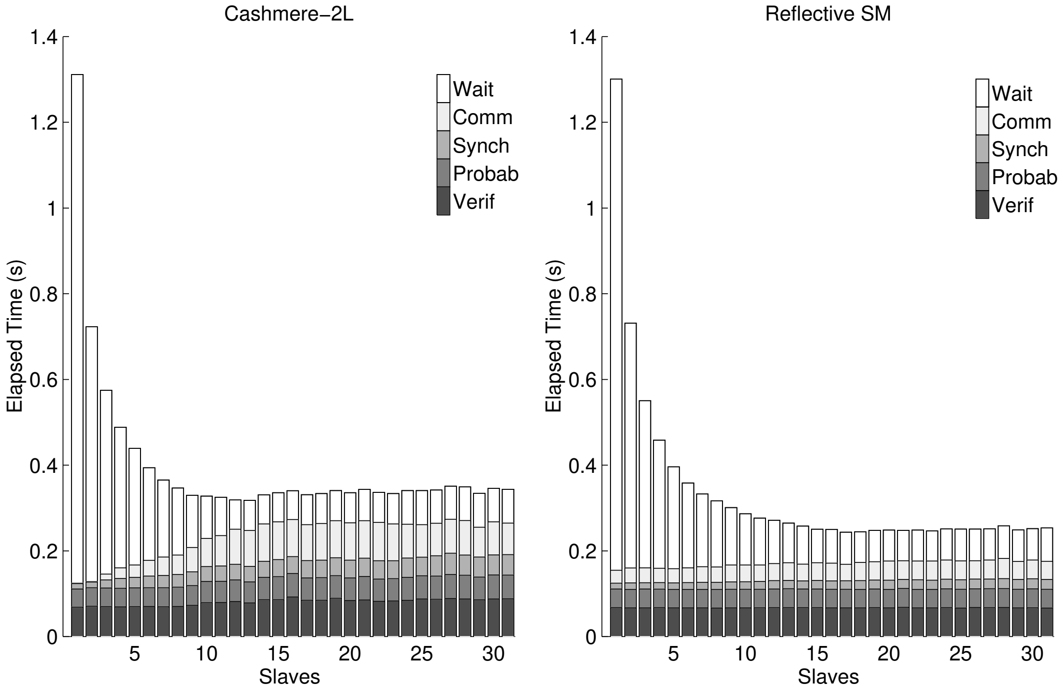

Figure 3: Composition of the elapsed times *per image* from the point of view of the master process. Actual times were obtained in real–time loops iterated for at least 50 images.

It can be observed that with a few slaves only, most of the time in the master is in the Wait category. This is due to the fact that at this point the slaves can not keep up with the rate at which the master generates new work requests. When the number of slaves is increased this time decays, until a saturation point is reached with twelve slaves. The residual waiting time is due to the fact that in the second phase of the search algorithm (expansion towards a complete solution) there is an algorithmic limit on the degree of parallelism that can be achieved. Note that twelve is exactly the size of a complete solution to the test case used. Whenever it is necessary to determine an outlier in a solution ruled as infeasible, at most twelve feasibility checks are needed and nothing else can be done while these are not finished.

Another interesting phenomenon is the fact that the communication overhead increases when the number of slaves is gradually raised from one to fifteen, and remains constant afterwards. The explanation for this is quite complex. Through a careful analysis of the code, we were able to determine that the node that contains the master process was always chosen as the home node for all the shared pages used in the communication with the slaves. Thus, whenever the master writes to a shared page, the modifications are of course immediately reflected on its node, but are not propagated to the remote nodes. When the master increments a flag, its processor uses the remote write capabilities of the Memory Channel in order to update the protocol directory in the slave's node. These updates signal that the old copy of the modified page in the slave's cache is now invalid. When the slave is finally awakened, it incurs a page fault and sends a request for the missing page to the master's node. At this point a major overhead in the master is incurred, because as it detects that a request has arrived (via polling), it must send the requested page through the Memory Channel, an operation that takes time in the order of 300 μs.

There are two reasons why this overhead increases with the number of slaves. An obvious one is the fact that the number of master–slave transactions actually increases with the number of slaves, since a higher degree of parallelism implies that longer sequences of speculations will have been made whenever a roll–back is needed in the first phase of the search algorithm. A more subtle reason is the fact that with fewer slaves the master is able to issue work requests faster than they can be serviced. So the general execution patterns tend to be similar to this: the master issues a large number of work requests, the slaves all incur page faults at almost the same time, and the resulting protocol–issued page requests arrive at the master during a short time interval, being serviced in a batch by a unique polling operation. With more slaves, the master needs to do more work in between successive requests to slaves, because the more frequently incoming answers may involve relatively expensive operations. This "more continuous" flow of incoming answers requires more polling operations and thus the associated overhead of entering the protocol increases. Furthermore, the chance that an incoming page–fetch protocol message interrupts the master while it is performing useful computation is significantly increased. These two phenomena help to explain the slight increase in the elapsed times for the inherently sequential operations as well.

After this analysis, it became clear that our parallel algorithm would probably benefit from the utilization of a RSM protocol, in which the modifications performed in the shared memory are immediately broadcast to all the nodes. Since in Cashmere-2L the master is the home node for all the shared pages, the broadcasts do not increase the incoming traffic in the master's node. Furthermore, except in the cases where the number of slaves is very small, the computation–time in the slaves is not as critical as the master's. So a little extra traffic on their side can't hurt performance significantly either (keep in mind that the amount of data communicated is relatively small in this algorithm). On the other hand, with RSM, all the data needed by the slaves is already in their caches whenever they are awakened. This avoids the overhead of cache misses at the slave side and, more important, avoids the interruption of the master when it is performing memory–intensive sequential computation.

In fact, the results obtained with the RSM protocol, displayed in Figs. 2 and 3, show the Comm and Synch times increase very slightly and the Verif and Probab times remain virtually constant, as the number of slaves is increased (contrast these results with those obtained for Cashmere-2L). With only a few slaves, RSM is inferior to Cashmere-2L because data that is communicated just inside a single node in the latter must be broadcast in the former, generating a small but not negligible overhead. However, in these cases, hardware coherence is the best solution anyway. With four or more slaves, RSM becomes the most effective run–time system available for our parallel algorithm, achieving a maximal speedup of 4.44, with 18 processors. We must stress that at this configuration, the maximum theoretical speedup possible given the serial fraction of the program is only 6.27 and most of the difference between this and the speedup actually attained can be explained by algorithmic limitations on the degree of parallelism available in the second phase of the search.

6 Conclusion

It is well known that the greater level of abstraction of shared memory (if compared to message passing) reduces the complexity of implementing, debugging and maintaining parallel applications. However, it is also believed that, in general, more abstract programming models entail bigger overheads in architectures that do not provide native support for them. In this work we demonstrate that the trade–offs involved in the selection of an ideal implementation environment for certain irregular applications can be quite more complex than those captured by "rules–of–thumb". The superiority of all three shared memory protocols tested with respect to Digital PVM shows that the intimate relationship between algorithmic parallelization aspects such as the computation–communication ratio and the intrinsic properties of the underlying architecture such as network topology, bandwidth and latency can lead to quite unexpected results if proper care is not taken.

Even among the shared memory systems, the surprising superiority of Reflective Shared Memory over Cashmere-2L is explained entirely by the fact that our parallel implementation is centralized in a unique master processor. Hence, Cashmere-2L does not benefit from any kind of clustering in the resulting communication patterns, and the extra traffic generated by the broadcasts performed with Reflective Shared Memory does not interfere too much with the critical path of the computation.

We would like to stress the fact that both parallel architectures and run–time systems are still in a process of quick evolution and even subtle changes in their implementation can have a major impact on the performance of applications that use irregular algorithms such as Probabilistic Search. Thus, careful empirical analysis such as the one presented here still seems to be a strong requirement for obtaining satisfactory (*e.g.* real–time) performance in these cases.

References

[1] R. A. Brooks. Symbolic reasoning among 3-D models and 2-D images. *Artificial Intelligence*, 17:285–348, 1981.

[2] R. B. Gillett. Memory channel network for PCI. *IEEE Micro*, pages 12–18, 1996.

[3] R. B. Gillett and R. Kaufmann. Experience using the first–generation memory channel for PCI network. Submitted for publication.

[4] E. Grimson and T. Lozano-Pérez. Localizing overlapping parts by searching the interpretation tree. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 9:469–482, 1987.

[5] E. L. Lawler and D. E. Wood. Branch–and–bound methods: A survey. *Operations Research*, 14:699–719, 1966.

[6] D. G. Lowe. Fitting parameterized three–dimensional models to images. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 13(5):441–450, 1991.

[7] D. G. Lowe. Robust model–based motion tracking through the integration of search and estimation. *International Journal of Computer Vision*, 8(2):113–122, 1992.

[8] R. Stets, S. Dwarkadas, N. Hardavellas, L. Kontothanassis G. Hunt, S. Parthasarathy, and M.Scott. CASHMERE-2L: Software coherent shared memory on a clustered remote–write network. In *Proc. 16th Symposium on Operating Systems Principles*, 1997.

[9] V. Sunderam. PVM: A framework for parallel and distributed computing. *Concurrency: Practice and Experience*, 4(2):315–339, 1990.