First European PVM Users' Group Meeting Rome, Italy, October 9-11 1994

Performance of PVM on a Highly Parallel Transputer System

Alexander Reinefeld
Center for Parallel Computing
Universität Paderborn
ar@uni-paderborn.de
http://www.uni-paderborn.de/pcpc/pcpc.html

Volker Schnecke
FB Mathematik/Informatik
Universität Osnabrück
volker@informatik.uni-osnabrueck.de
http://brahms.informatik.uni-osnabrueck.de

Abstract

Although PVM was developed to use a network of heterogeneous UNIX computers as a single large parallel computer, it has become an interface for portable programming even on MPP's. We present PVM performance results for a massively parallel transputer system with up to 512 processors. In comparison to an implementation of the same application in the native transputer operating system Parix, we realized about 30% performance loss for the PVM application. This is mainly caused by restrictions in the process model and the less efficient communication of PVM in comparison to Parix.

1 Introduction

Analogous to the shift from assembler language programming to the third-generation languages in the early years of computer science, we are currently witnessing a paradigm change towards the use of portable programming models in parallel high-performance computing. Like before, the high-level programming environment is paid for by a lower system performance.

But how much does portability cost in practice? Is it worth paying that price? And what effect has the choice of the programming environment on the algorithm architecture of the application program?

In this paper, we investigate the performance of an application program running on three different programming interfaces: (1) the native Parix operating system, (2) a homogeneous PVM environment and (3) a heterogeneous PVM programming environment. For direct comparison, we used the same underlying hardware system for all three cases.

2 The Application

Our application is an algorithm from the domain of Artificial Intelligence, the iterative-deepening search algorithm *IDA** [1]. This is a heuristic DFS algorithm that simulates a best-first search by a series of depth-first searches with successively increased cost-bounds. Iterative-deepening search is used in many applications that cannot be solved with direct best-first searches (like A*) because of memory restrictions. The search tree is highly irregular, so that efficient work-partitioning and dynamic work-load balancing schemes are required in a parallel implementation to achieve a good overall performance. Our Asynchronous IDA* (AIDA*) algorithm [3] works in two phases:

1. In an initial task partitioning phase, the nodes of a search-frontier level in the tree are generated in parallel and stored on all processors. Each of the nodes is a root of a subtree and represents an indivisible piece of work for the next phase.

2. In an asynchronous search phase, each processor expands its 'own' frontier nodes in depth-first fashion. When a processor becomes idle, it sends a work request to obtain a work packet (i.e., an unprocessed frontier node) from another processor. Work requests are forwarded on a ring topology until one processor has work to share or the request is returned to the sender. If no work is left for this iteration, the processor proceeds with the next iteration, that is, it again starts the search on its frontier-nodes, but now with an increased cost-bound. When a processor finds a solution, all others are informed by a broadcast.

To avoid speedup-anomalies, we search for all solutions in our implementation. This ensures that the node count is always the same for a given problem instance.

3 Parix Implementation

Parix is a the native operating system for the Parsytec GC transputer systems. It provides UNIX functionality at the frontend with library extensions for the needs of the parallel system. Parix is available for the massively parallel transputer based GCel series, the medium-sized PowerXplorer (with PowerPC 601), and the new high-performance GC/PowerPlus series with two PowerPC 601 and four transputers per node.

On the GCel-1024, the processing elements are organized in a 2D-grid. A virtual topologies library provides a set of commonly used topologies for application programs, that are optimally mapped onto the underlying hardware. We used the *ring* and *torus* topologies for our implementation.

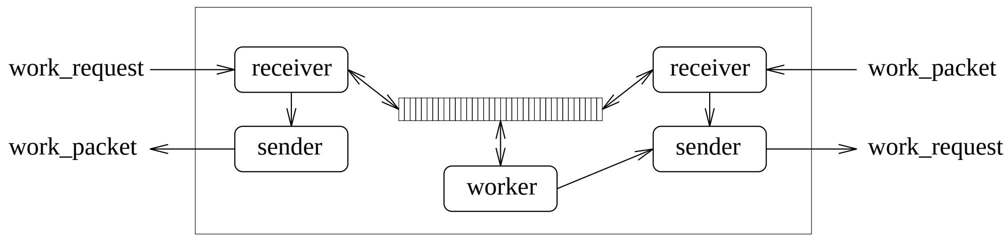

Figure 1: Process model on Parix (ring topology)

Figure 1 depicts the process model of our AIDA* implementation on Parix, with five threads running on a single node. In Parix, all threads created by a program are executed in the same context. They share the same global variables defined by the program. The worker and receiver threads in Figure 1 access the common frontier node array for retrieving new work packets. The communication threads (sender and receiver) serve incoming messages and send work requests when asked by the worker. On a torus topology, nine threads are used instead of the shown five.

Performance results [3, 4] obtained on a 1024-node system indicate that our scheme works well for problems taking more than, say, a minute parallel computing time. Only few communication is necessary at the end of an iteration when some of the processors get idle. On a $32 \times 32 = 1024$-node torus, we achieved an average efficiency of 93 % for some random instances of the 4×5-puzzle [4]. For the smaller 4×4-puzzle we achieved 79% efficiency on Korf's [1] 25 largest random problem instances [4]. The detailed analysis [3, 4] exhibits that the Parix implementation scales well on large MIMD systems and has only low communication overheads.

4 PVM Implementation

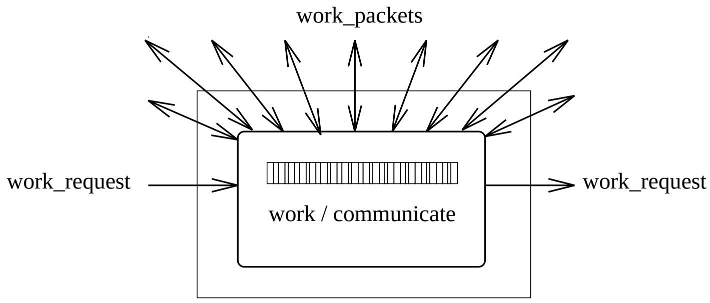

Figure 2: Process model on PVM

For the implementation on PVM, the AIDA* process model had to be changed. Our transputer executes only one task per processor, so that tree search and load-balancing must be done in one single task. Following Rao and Kumar [2], we interrupt the node expansion process at certain time intervals to allow processing of incoming messages. The frequency of the subroutine calls for this communication management depends on the system size. Communication service interrupts occur every 100 to 1000 node expansions. With an average working rate of 35000 nodes per second one processor can manage 40 to 300 messages in one second (64 and 512 tasks resp.). As shown in Figure 2, the implementation also makes use of a ring topology (like the Parix version), but work packets are returned directly to the requester.

5 Results

Table 1 shows the performance results of AIDA* for three instances of the $4{\times}4$-Puzzle out of Korf's problem set [1]. All efficiency data is normalized to the performance of an optimal sequential Parix implementation.

We used two different PVM implementations for our experiments: PVM_{homo} is an α-release of a homogeneous PVM version[1] where all tasks are spawned on the transputer nodes of the parallel system. The heterogeneous version, PVM_{hetero}, runs on any available hardware, permitting a heterogeneous collection of serial and parallel computers to appear as a single virtual machine. The task management is done by a PVM daemon running on the front-end machine of the parallel system. Both PVM environments are based on Parix.

As can be seen from the table, all three versions scale reasonably well for moderate system sizes. Of course, the implementation on Parix is the fastest. The homogeneous PVM is less

[1]Thanks to Parsytec, who provided us with an α-release of the homogeneous PVM.

prob		$t(64)$	$e(64)$	$t(128)$	$e(128)$	$t(256)$	$e(256)$	$t(512)$	$e(512)$
64	Parix	129	97%	66	95%	45	70%	22	71%
	PVM_{homo}	156	80%	82	76%	50	63%	41	38%
	PVM_{hetero}	–	–	86	73%	57	55%	54	29%
75	Parix	282	96%	139	98%	71	96%	40	85%
	PVM_{homo}	327	83%	169	81%	92	74%	67	51%
	PVM_{hetero}	–	–	–	–	99	69%	80	43%
84	Parix	588	98%	294	98%	152	96%	90	81%
	PVM_{homo}	686	85%	355	82%	186	78%	128	57%
	PVM_{hetero}	–	–	–	–	196	74%	149	49%

Table 1: The results on the Parsytec GCel

efficient, but it performs much better than the heterogeneous version. From the very nature of the heterogeneous PVM, this programming environment cannot be as efficient as the homogeneous implementation, because all messages are checked whether the destination is 'within' or 'outside' the MPP system.

	1 byte	64 byte	256 byte	1024 byte	4096 byte
Parix	$45\,\mu s$	$90\,\mu s$	$272\,\mu s$	$951\,\mu s$	$3716\,\mu s$
PVM_{homo}	$468\,\mu s$	$527\,\mu s$	$769\,\mu s$	$1681\,\mu s$	$4794\,\mu s$
PVM_{hetero}	$3069\,\mu s$	$3109\,\mu s$	$3346\,\mu s$	$4269\,\mu s$	$7986\,\mu s$

Table 2: Communication time between neighbored processors

In a separate experiment, we benchmarked latency times and communication speed of the three environments. Table 2 shows the communication time between neighbored processors in the GCel system. The messages are not encoded to achieve maximal performance. As can be seen, the setup times of the PVM communication are very high, especially in the heterogeneous version. In our application, only small messages (≈ 110 byte) are sent, resulting in higher PVM communication overheads for the larger system sizes (Table 1).

processors	64	128	256	512
without search	$43\,ms$	$87\,ms$	$174\,ms$	$349\,ms$
with search	$348\,ms$	$533\,ms$	$648\,ms$	$1209\,ms$

Table 3: Times to send a message (128 bytes) through all processors

In another experiment we tried to determine the performance loss due to the restriction in the process model. We can't do concurrent work and loadbalancing, because we only have one task on each transputer. So, communication can only take place at certain time intervals, when the search process is interrupted. Table 3 shows the time for sending a message on a ring through all processors of the system with and without a search routine. The time measured without the search routine is the time one would expect, if the communication can be done by a special task concurrent to the node expansion. The delay caused by the search routine increases the average time for a message between 'neighbored'[2] processors on the ring from $681\,\mu s$ to $2.4\,ms$ on 512 processors when using the same communication frequency like in the real application.

6 Conclusions

The empirical investigations revealed that our parallel implementation scales sufficiently well on Parix and PVM for moderate systems sizes (<256 processors). This is because few communication occurs in this specific application, and many requests are answered in the direct neighborship of the requester. For the larger system sizes, Parix clearly outperforms PVM. We expect the gap between Parix and PVM to become more pronounced in applications with a higher communication demand.

The performance loss for larger systems is mainly caused by the implementation of PVM for the transputer system. Especially the restriction that only one task on each processor is possible yields to slower communication because each message could not be processed immediately after receivement. Further it is not possible to do efficient work-load balancing if PVM does not provide any information about the MPP, i. e. the position of a task in the network or information about the tasks in the direct neighborship.

References

[1] R.E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search.* Art. Intell. 27 (1985), 97–109.

[2] V. Kumar and V. N. Rao. *Scalable parallel formulations of depth-first search.* In: Kumar, Gopalakrishnan, Kanal (eds.), Parallel Algorithms for Machine Intelligence and Vision, Springer-Verlag (1990), 1–41.

[3] A. Reinefeld and V. Schnecke. *AIDA* – Asynchronous Parallel IDA*.* Procs. 10^{th} Canadian Conf. on Art. Intell. AI'94, (May 1994), Banff, Canada, Morgan Kaufman, 295–302.

[4] A. Reinefeld and V. Schnecke. *Work-load balancing in highly parallel depth-first search.* Procs. Scalable High Perf. Comp. Conf. SHPCC'94, Knoxville, Te, 773–780.

[2]Because PVM does not provide any information on the position of a task in the MPP, the neighbored processors on the ring are not always neighbored in the system, so the mapping of the ring onto the 2D-grid of the GCel is randomly.