Procs. ZEUS'95 Workshop on Par. Programming and Computation, Linköping, Sweden (1995).

Portability versus Efficiency? Parallel Applications on PVM and Parix

Alexander Reinefeld
Center for Parallel Computing
Universität Paderborn
ar@uni-paderborn.de

Volker Schnecke
Mathematik/Informatik
Universität Osnabrück
volker@informatik.uni-osnabrueck.de

Abstract

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 use of a high-level programming environment must be paid for by a reduced system performance.

But how much does portability cost in practice? Is it worth paying that price? What effect has the choice of the programming model on the algorithm architecture?

In this paper, we attempt to answer these questions by comparing two applications from the domain of combinatorial optimization that have been implemented with the Parix and PVM programming models. Performance benchmarks have been run on three different systems: a massively parallel transputer system with relatively slow T805-processors, a moderately parallel Parsytec GC/PowerPlus system with powerful 80 MFLOPS processors, and a UNIX workstation cluster connected by a 10Mbps LAN. While the Parix implementations clearly turned out to be fastest, PVM gives portability at the cost of a small, acceptable loss in performance.

1 Introduction

Contemporary parallel computing systems are strikingly diverse in their architectures and operating systems. A variety of interconnection networks can be found in today's most successful commercial machines: hypercube (nCube), 2D-grid (Intel Paragon, Parsytec GC), ring/crossbar combination (Convex SPP1000), Ω-network (IBM SP2), fat tree (CM5), hierarchical rings (KSR2), and 3D-torus (Cray T3D). All of them have their specific advantages and it is still an open question which topologies will 'survive' in the future.

Unfortunately, the programming interfaces are also very dissimilar, making it difficult to port a given code from one hardware platform to another. As it seems, portability can only be achieved by the use of vendor independent programming environments like PVM [4, 24], MPI [10, 5], PARMACS, P4, Zipcode, Express, and Linda. (For an overview on programming environments see [5]).

Portability and scalability are the keywords of this paper. Ideally, a parallel implementation would meet both properties, but in practice interdependencies make this impossible. We compared the performance of two application programs running on Parix (PARallel extensions to UnIX) and PVM (Parallel Virtual Machine). Three different hardware platforms have been used in our experiments: A 512-node transputer system, a Parsytec GC/PP with 96 PowerPC-601 processors and a UNIX workstation cluster with 29 SUN SparcStations. Our experiments give answers to the following questions: Is there any impact of the choice of the programming model on the architecture of the application program? How fast is PVM as compared to Parix? To what extent are PVM programs portable and scalable?

2 Applications

As an application problem domain, we have chosen the class of *discrete optimization problems*, which can be defined in terms of finding a solution path in a tree or graph from an initial (root) state to a goal node. Many applications in planning and scheduling can be formulated with this model: the cutting stock problem [2], the bin packing problem, vehicle routing, VLSI floorplan optimization [1, 27], satisfiability problems, the traveling salesman problem [6], the $N \times N$-puzzle [7], and partial constraint satisfaction problems [3]. All of them are known to be NP-hard.

For our benchmarks, we have chosen two typical applications with different solution techniques: the VLSI floorplan optimization problem, which is solved with a depth-first branch-and-bound strategy, and the $N \times N$–puzzle, which is solved with an iterative-deepening search. Both are based on *depth-first search (DFS)*, which traverses the decision tree in a top-down manner from left to right. DFS is commonly used in combinatorial optimization problems, because it needs only $O(d \cdot w)$ storage space in trees of depth d and width w.

2.1 VLSI Floorplan Optimization

The *floorplan area optimization* [23, 27] is a stage in the design of VLSI chips. Here the relative placements and areas of the building blocks of a chip are known, but their exact dimensions can still be varied over a wide range. A floorplan is represented by two dual polar graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{H} = (\mathcal{W}, \mathcal{F})$, and a list of potential implementations for each block. As shown in Figure 1, the vertices in $\mathcal{V}$ and $\mathcal{W}$ represent the vertical

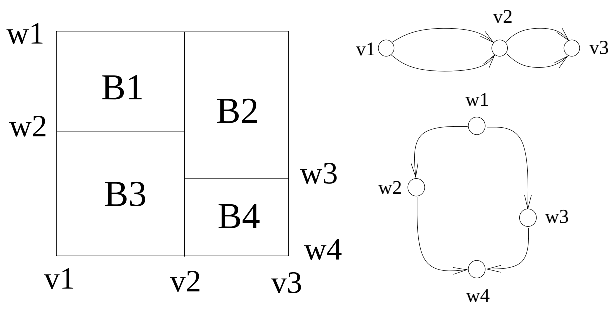

Figure 1: A floorplan and the graphs $\mathcal{G}$ and $\mathcal{H}$

and horizontal line segments of the floorplan. There exists an edge $e = (v_1, v_2)$ in the graph $\mathcal{G}$, if there is a block in the floorplan, whose left and right edges lie on the corresponding vertical line segments. For a specific configuration (i.e. a floorplan with exact block sizes), the edges are weighted with the dimensions of the blocks in this configuration. The solution of the floorplan optimization problem is a configuration with minimum layout area, given by the product of the longest paths in the graphs $\mathcal{G}$ and $\mathcal{H}$.

Our implementation builds a tree where the leaves are complete configurations and the inner nodes at depth d represent partial floorplans consisting of blocks $B_1 \dots B_d$. The *depth-first branch-and-bound (DFBB)* solution algorithm employs a heuristic cost-function to eliminate unnecessary parts of the search space that are known not to contain an optimal solution. When a new (possibly non-optimal) solution has been found, the search continues with the improved cost-bound, now pruning all subtrees with cost-estimates higher than the new cost-bound. Newly established cost-bounds are broadcasted, so that all processors share the best available bound at any stage in the search.

2.2 $N \times N$–Puzzle

Applications with bad initial cost-bounds and many alternatives in the decision trees are solved with heuristic search algorithms like A* [11, 13] and IDA* [7]. One such example is the $N \times N$–puzzle [7, 11, 13]. Here, a given initial configuration of tiles in a squared tray of size $N \times N$ must be re-arranged with the fewest number of moves into a given goal configuration. No effective upper cost-bounds are known for this problem, and hence it cannot be solved with DFBB. Even the relatively small 15-puzzle ($N = 4$) has a search space of $16!/2 \approx 10^{13}$ states. While it would seem easy to obtain any solution, finding an optimal (=shortest) solution is *NP*-complete [16]. In general, it takes some hundred million node expansions to solve a random problem instance of the 15-puzzle, using the *Manhattan distance* (the sum of the minimum displacement of each tile from its goal position) as a heuristic estimate function.

*Iterative-deepening A** (IDA*) [7] simulates a best-first search by a series of depth-first searches with successively increased cost-bounds. Its low space overhead of $O(d)$ makes it feasible in applications where A* [11] cannot be used due to memory limitations. With a non-overestimating heuristic estimate function, IDA* is guaranteed to find an optimal (shortest) solution [7]. In contrast to DFBB, IDA* halts after finding a first solution, because optimality is guaranteed by the iterative approach with the minimal cost-bound increments [7].

3 Parallel Depth-First Search

Depth-first search can be performed in parallel by partitioning the search space into many small, disjunct parts (subtrees) that can be explored concurrently. We have developed a scheme called *search-frontier splitting* [18] that can be applied to breadth-first-, depth-first- and best-first search. It partitions the search space into g subtrees that are taken from a 'search-frontier' containing nodes n with the same cost value $f(n)$. Search-frontier splitting has two phases:

1. Initially, all processors generate (synchronously) the same 'search-frontier' and store the nodes in their local memories. A 'search frontier' is a level in the tree where all nodes have the same cost-value. Each node represents an indivisible piece of work for the next phase. Hence, on a p-processor system, the search frontier should be chosen large enough, so that it contains at least p nodes.[1]

2. In the asynchronous search phase, each processor selects a disjunct set of frontier nodes and expands them in depth-first fashion. When a processor becomes idle, it requests a work packet (=unprocessed frontier node) from another processor. Work requests are forwarded from one processor to the next until one of them has work to share. When there are no work packets left, the search space is exhausted and the search is terminated. When a processor finds a solution, all others are informed by a broadcast.

Search-frontier splitting is a general scheme. It can be employed in depth-first branch-and-bound (DFBB) and in iterative-deepening depth-first search (IDA*). In DFBB, the search continues until all nodes have been either explored or pruned due to inferiority. Newly established bounds are communicated to all processors to improve the local bounds as soon as possible.

[1]It seems natural to generate the search frontier iteratively, by incrementing the cost-value by the minimum amount until there are at least $const \cdot p$ entries in the frontier nodes array. In our experiments, we have chosen $const = 5$.

In the iterative-deepening variant, *AIDA** [18], subsequent iterations are started on the previously established frontier node arrays. Work packets change ownership when they are sent to other processors, thereby improving the global load-balance over the iterations. Lightly loaded processors, which have asked for work in the last iteration will be better utilized in the next. As a consequence, the communication overhead decreases during the search process [18].

4 Parallel Implementations on Parix and PVM

We implemented the search-frontier splitting scheme on Parix and PVM using the VLSI floorplan optimization problem and the 15-puzzle as application domains.

4.1 Parix

PARIX (PARallel UnIX extensions) [12] is the native operating system on Parsytec GC systems. It provides UNIX functionality at the front-end with library extensions for the needs of the parallel system. The Parix software package comprises components for the program development environment (compilers, tools, etc.), runtime environment (libraries), multi-user administration, and control-net software. Parix is available for a variety of parallel Parsytec computers. At the Paderborn Center for Parallel Computing, we run Parix on

- a 1024-node transputer system *GCel-1024*, where the T805-nodes are interconnected in a 2-dimensional mesh topology,
- a medium size *PowerXplorer* with PowerPC 601 application processors and T805-transputers for the communication,
- a high-performance *GC/PowerPlus*, with two PowerPC-601 application processors and four T805 communication processors interconnected in a 2-dimensional fat mesh with 4 links in each direction.

A *virtual topologies library* [20, 21] provides an optimal mapping of the application process structure onto the underlying hardware interconnection structure. Based on the virtual topologies, common *global communication functions* like broadcast, global sum, min, max and barrier synchronization are available for groups of processes.

Algorithm Architecture on Parix. While in principle any of the Parix virtual topologies could have been chosen for our implementation, it is clear that the torus- and ring-embeddings yield lowest dilation and congestion on our 2D hardware grid.

For the work distribution, we used a *packet forwarding* scheme, where the work requests are forwarded to neighbored processors until either some work is sent back or

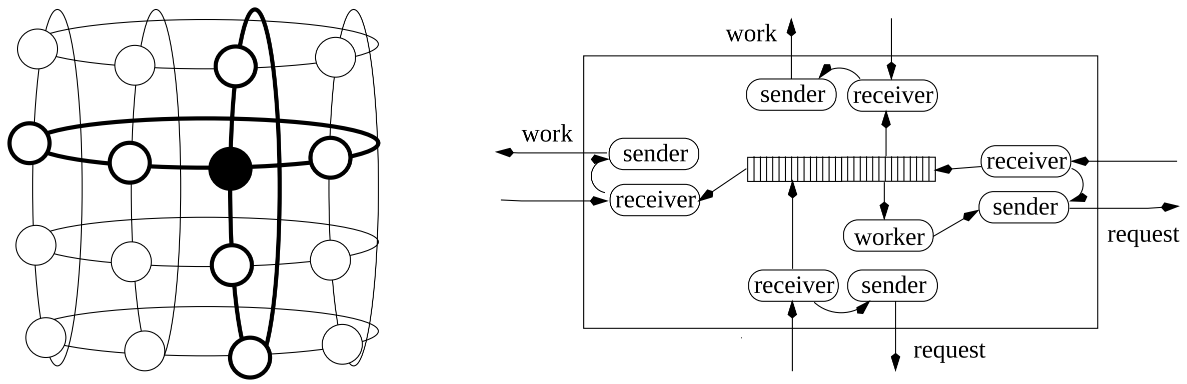

Figure 2: Process architecture on Parix

the message makes a full round through the system, thereby indicating that no work is available. On a torus topology, requests are first forwarded along the horizontal ring. If none of the processors has work to share, the request is then sent along the vertical ring, see Figure 2. This yields a wide-spread work distribution, while each processor communicates only with a subset of $2\sqrt{p}-2$ processors.

The right part of Figure 2 illustrates the process architecture of our Parix implementation. Each processor executes nine concurrent threads (=light weighted processes). All threads run in the same context, that is, they share the same global variables defined by the main program. Hence, the worker and receiver threads have direct memory access to the processor's frontier node array for retrieving new work packets. Memory contention is arbitrated by semaphores. The sender and receiver threads serve incoming messages and send work requests on demand.

4.2 PVM

The *Parallel Virtual Machine (PVM)* is one of the most popular message passing models. PVM is public domain and can be obtained free of charge via ftp [4]. PVM makes a heterogeneous network of parallel and serial UNIX computers to appear as a single concurrent computational resource. Controlled interaction of a heterogeneous set of workstations is done via TCP/IP sockets or native MPP communication protocols, with the possibility of dynamically adding more resources to the network as they are needed.

Applications view PVM as a general and flexible parallel computing system that supports the message-passing paradigm. PVM is not restricted to specific architectures. Application programs may have arbitrary control and dependency structures with arbitrary – and even dynamical – relationships among the parallel processes. This allows the most general form of MIMD (resp. SPMD-) programming. Task granular-

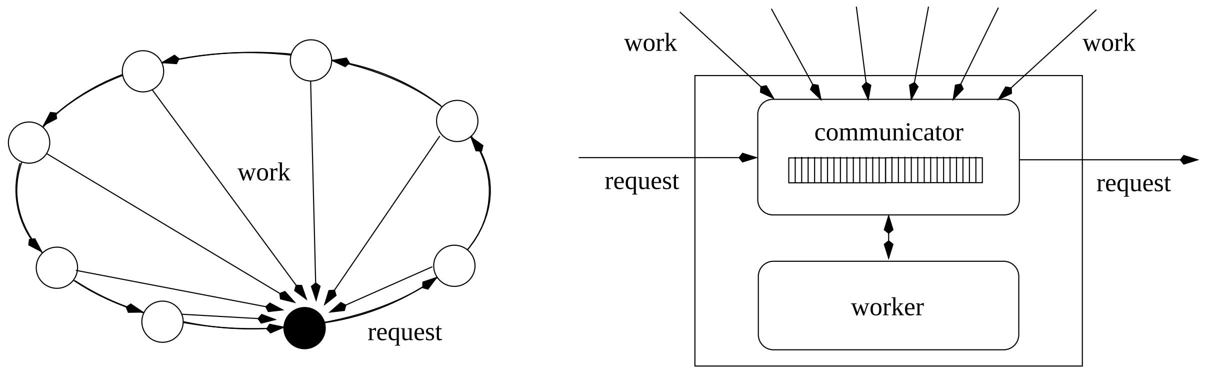

Figure 3: Process architecture on PVM

ity, communication frequency and load balancing are completely at the disposal of the programmer. There exist library calls for blocking and non-blocking send and receive operations. Groups of processes can be defined for broadcasts, barrier synchronization and other group functions. Release 3.3 provides features for the efficient use of architecture-dependent and low-level communication modes on parallel systems.

Algorithm Architecture on PVM. Only few modifications were necessary to adapt our Parix application to the PVM programming model. Both models provide similar constructs for the communication interfaces. Compared to Parix, PVM is lacking a common 'execution context' for sharing objects among tasks running on the same processor. Therefore, we had to implement a mechanism for sharing the frontier nodes by explicit message transfer. In PVM, all synchronization and information sharing is solely done via messages, even when running on the same processor. While this is certainly the most abstract (hardware independent) approach, it might cause unnecessary inefficiencies.

In our implementation (Fig. 3) the frontier nodes array is maintained by the the communicator task. When a worker task runs out of work, it sends a message to its communicator executing on the same node. When there are no work packets left in the frontier nodes array, the communicator task sends – as before – a request to it neighboring processors. However, unlike in the Parix approach, work packets are directly returned to the requester. This does not cause any extra programming effort, because PVM implies a clique network, directly connecting every processor to every other.

5 Performance Comparison

We run two application programs (15-puzzle and floorplan optimization) on three machines (GCel, GC/PP, UNIX workstation cluster) and two programming models (Parix

	Parix			PVM		
p	*time*	*sp*	*eff*	*time*	*sp*	*eff*
64	416	62	97%	390	53	83%
128	211	124	97%	202	102	80%
256	107	223	87%	109	184	72%
512	51	404	79%	79	251	49%

Figure 4: GCel transputer system

	Parix			PVM		
p	*time*	*sp*	*eff*	*time*	*sp*	*eff*
12	81	12	97%	83	11	92%
16	61	15	96%	63	15	91%
24	35	23	96%	45	20	84%
32	31	30	94%	36	25	77%
48	21	45	93%	26	35	72%
64	16	58	90%	21	42	66%
96	12	78	81%	16	55	57%

Figure 5: GC/PowerPlus

and PVM). Not counting the workstation experiments, for which only PVM was available, we run a total of 10 benchmarks. From the bulk of statistical data, we selected the most significant results to be presented in the following sections.

5.1 Results for the 15-Puzzle

Tables 4 and 5 show the results obtained with a subset of Korf's standard random problem instances of the 15-puzzle [7] on the GCel and GC/PP. For systems with p processors, the total elapsed *time*, the speedup *sp* and the corresponding efficiencies *eff* are given. All times are wall-clock times, measured in seconds. Speedups and efficiencies are normalized to the fastest sequential Parix implementation.

With the Parix process model discussed above (Fig. 2), we achieved on both Parsytec machines an almost perfect scalability ranging from 97% on the 'small' systems to 80% efficiency on the larger networks. This is a remarkable result, considering that the largest GCel configuration comprises as many as 512 processors, while the GC/PP has only 96 PEs. The latter, however, is more powerful, resulting in a much faster execution speed of only 12 seconds on the largest system. Clearly, for such short execution runs, the presented speedups can hardly be improved.

With its 80 MFLOPS peak performance, a single GC/PowerPlus node is expected to be approximately 20 times faster than a GCel-transputer node with its 4 MFLOPS. In this special application, the speedup is even larger: We measured a performance ratio of 26. On the other hand, the communication bandwidth of the GC/PP is only 4 times higher than that of the GCel. Due to the worse communication/computation performance ratio, it is clear, that the 15-puzzle application does not scale as well on the larger GC/PP systems.

Being implemented 'on top of Parix', PVM requires some additional CPU-time for the bookkeeping, buffer-copying and function calls. For the larger GCel system, this

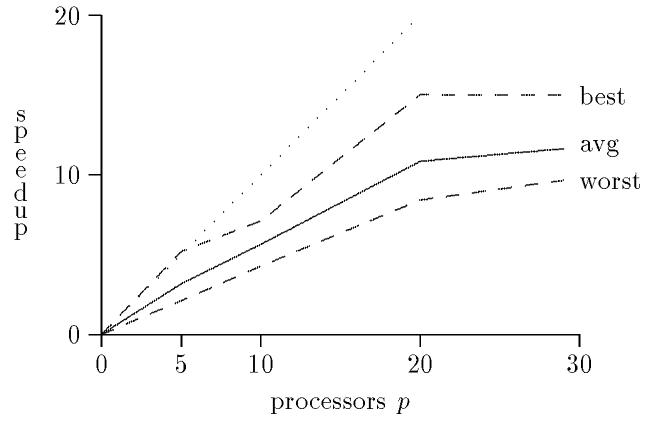

Figure 6: PVM speedup on SUN SS10-cluster, Ethernet

results in

1. increased communication latency due to the larger number of hops,
2. increased message traffic due to the increased number of fine-grained work packets,
3. increased computing overhead, since the T805 processors also perform routing.

Even so, the PVM performance losses seem to be acceptable for the medium sized systems.

PVM's availability and portability allowed us to develop our implementations on a workstation cluster, while the production runs were performed on all kinds of parallel systems. We also benchmarked a *heterogeneous PVM environment*, that allows a 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 UNIX front-end of the parallel system. Since all communication is first checked whether its destination lies 'within' or 'outside' the MPP system, the heterogeneous PVM cannot be as efficient as the homogeneous PVM discussed above. Performance results for the heterogeneous PVM may be found in [19, 22].

Figure 6 illustrates the performance achieved on a cluster of SUN SparcStation-10. While all workstations are of the same model and speed, the measurements were taken on busy systems with different load factors and a busy Ethernet with several bridges. This results in widely varying execution speeds. Figure 6 shows the average, the slowest and the fastest of 10 runs on the same sample problem set. As can be seen, the initially good speedup seems to level off when more than 20 workstations are employed. The exact data indicates that this is due to the longer start-up times and increased communication latencies.

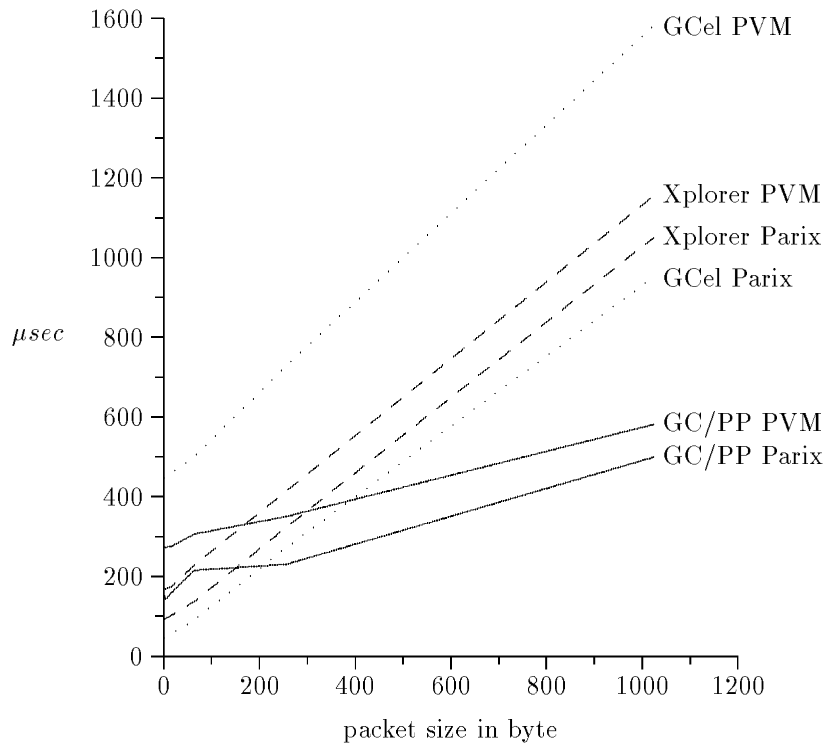

Figure 7: Communication speed between neighbored nodes

5.2 Communication Speed of Parix and PVM

In a separate experiment, we benchmarked latency times and communication throughput of Parix and PVM on the GCel, GC/PP and PowerXplorer. Figure 7 shows the communication performance between neighbored processors for each of the systems. The PVM times include complete packing and unpacking on both sides, sender and the receiver.

The most apparent fact is, that in contrast to the other performance graphs, the GC/PP curves are not straight. This is attributed to the fixed initial overhead required for multiplexing the message among the four links of the communication subsystem. The multiplexing is done by four dedicated T805 communication processors. In practice, the computing performance would greatly benefit by latency hiding techniques, which is not reflected in these performance graphs. Hence, these graphs represent the worst case.

For the systems with PowerPC 601 processors (GC/PP and PowerXplorer), the communication throughput of PVM and Parix are very similar. Only for the GCel, there is a larger discrepancy between Parix and PVM communication performance, illustrated by the dotted line. This is caused by the GCel transputer nodes, which are by a factor of 20 slower than the PowerPC nodes. Hence, it takes much longer on the GCel to initiate a communication (function calls, data copying, etc.). Once the initial setup is done, the same throughput is achieved on the GCel as on a PowerXplorer. Due

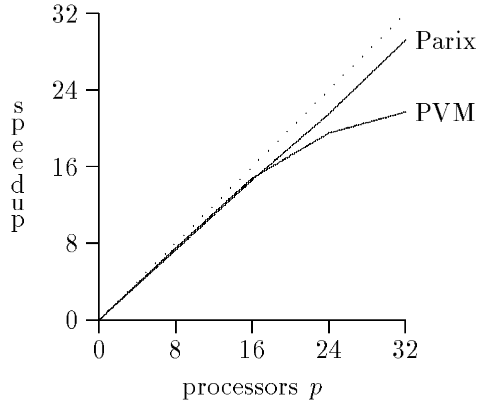

Figure 8: VLSI floorplan optimization on GC/PowerPlus

to the fat communication mesh, consisting of 4 links in each direction, the GC/PP communication is faster.

5.3 Results on the VLSI Floorplan Optimization Problem

Figure 8 shows the results obtained with the VLSI floorplan optimization algorithm. We used a standard VLSI benchmark problem from [26] with 25 building blocks and six different implementations per block. This gives a search space of $6^{25} \approx 10^{19}$ nodes.

Compared to the 15-puzzle application, more communication is necessary in the floorplan optimization, because the solution speed depends much on the availability of efficient bound values. Each time a new bound is found by any of the worker tasks, it is broadcasted to all others.

The Parix implementation uses a torus topology for submitting work requests and for transferring work packets. Our PVM implementation, in contrast, makes use of the implicit clique topology. A master process hands out work packets to the slaves which search their assigned subtrees and broadcast newly found bound values.

As can be seen in Figure 8, this simple farming approach yields sufficient performance results on small systems with ≤ 24 processors. Beyond, a hierarchy of master processes should be implemented to eliminate potential communication bottlenecks.

6 Conclusions

We evaluated two programming models, PVM and Parix, on three hardware platforms with two different application programs. What are the lessons learned? Recalling the title of our paper 'Portability versus Efficiency?', we draw the following conclusions:

Portability. PVM provides a portable programming model for implementing parallel applications on a wide variety of message-based MIMD systems. Many important scientific and industrial applications have been ported to PVM with the expectation to gain more independence from specific hardware platforms and vendors. First results from the Esprit project EUROPORT indicate, that it is possible to execute large PVM applications on various systems, ranging from moderately parallel symmetric multiprocessor systems to massively parallel systems with many hundred nodes.

Hardware independency, however, may tempt the programmer to use program features that might later be found inefficient on certain systems. Time-critical applications may need post-optimization to exploit specific MPP system features. As an example, PVM's clique-structured communication pattern is tailored for bus-connected workstation clusters (Ethernet, FDDI, ATM). As seen above, clique communication poses problems on massively parallel 2D systems, where the communication latency depends on the number of hops a message needs to reach the recipient. Here, nearest neighbor communication should be used instead.

Efficiency. Executing on top of the native operating system, the PVM programmming interface clearly induces additional overheads. The slower the application processor, the more time is spent in setting up the communication between processors. Compared to Parix, PVM needs more bookkeeping for the management of communication buffers, for any necessary data conversions and for additional system calls.

The performance losses are especially pronounced in the results obtained on the large transputer system (Table 4). Here, the PVM implementation took 79 seconds on a 512 node-system, while the Parix implementation computed the same solution in only 51 seconds.

Due to unpredictable execution times, time-critical interactive applications should better not be run on workstation clusters. Different processor capabilities may induce load imbalances and the system load might change dynamically during the program runtime. While the different processor speeds did not affect the performance of our PVM implementations (which includes a dynamical load balancing strategy), we still observed widely varying execution times from one execution run to the next.

Process Architecture. The choice the 'best' process architecture for a given application may be influenced by the hardware architecture and the programming model.

Compared to the Parix implementation, PVM needs more effort to implement a mechanism for sharing work-packets among the 'communicator-' and 'worker tasks' executing on the same node. In Parix, all threads run in the same context and share the same global variables defined by the main program. Hence, all threads have direct memory access to the same 'work packet array' maintained on a node. Memory contention is arbitrated by semaphores. In PVM, in contrast, all communication is performed via explicit message transfer – regardless whether it is a long distance communication or a communication within tasks running on the same processor.

Typically, the PVM process architecture is designed without knowing the topology of the target hardware system. Since PVM does not provide information about the location of the tasks in the (hardware-) network, fast communication (e.g., nearest neighbor communication) is hard to implement. The virtual topologies provided by PARMACS and MPI allow more efficient mappings of tasks to processors. While such mapping functions can also be implemented by matching the Parix process descriptors to the PVM task identifiers, such programming tricks clearly violate our ultimate goal of portability.

Acknowledgements

Thanks to our local PVM expert Axel Keller, who helped a lot in getting our experiments done within a tight time-frame. Also thanks to Jing Li for implementing parts of the PVM floorplan optimizer.

References

[1] S. Arvindam, V. Kumar and V. Rao. *Efficient parallel algorithms for searching problems: Applications in VLSI CAD.* 3rd Symp. Frontiers Mass. Par. Comp., Maryland (1990), 166–169.

[2] N. Christofides and C. Whitlock. *An algorithm for two-dimensional cutting problems.* Operations Research 25, 1 (1977), 30–44.

[3] E.C. Freuder and R.J. Wallace *Partial constraint satisfaction.* Artificial Intelligence 58(1992), 21–70.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Knoxville, TN, Techn. Rep. ORNL/TM-12187, May 1994. ftp: cs.utk.edu.

[5] R. Hempel, A.J.G. Hey, O. McBryan and D.W. Walker (eds.). Special Issue on *Message Passing Interfaces.* Parallel Computing 20,4(1994).

[6] M. Held and R.M. Karp. *The traveling salesman problem and minimum spanning trees.* Operations Research 18 (1970), 1138–1162.

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

[8] V. Kumar and V. Rao. *Scalable parallel formulations of depth-first search.* Kumar, Gopalakrishnan, Kanal (eds.), Par. Alg. for Mach. Intell. and Vision, Springer (1990), 1–41.

[9] V. Kumar, A. Grama, A. Gupta and G. Karypis. *Introduction to Parallel Computing. Design and Analysis of Algorithms.* Benjamin/Cummings Publ., Redwood City, CA (1994).

[10] Message Passing Interface Forum. *MPI: A message-passing interface standard.* Comp Sc. Dept., Univ. Tennessee, Knoxville, TN, CS-94-230, April 1994.

[11] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publ., Palo Alto, CA, 1980.

[12] Parsytec. *Parix V1.3 PowerPC Software Documentation* (Dec. 1994).

[13] J. Pearl. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, Reading, MA, (1984).

[14] V.N. Rao, V. Kumar and K. Ramesh. *A parallel implementation of iterative-deepening A*.* AAAI-87, 878–882.

[15] V.N. Rao and V. Kumar. *On the efficiency of parallel backtracking.* IEEE Trans. Par. Distr. Systems 4,4(1993), 427–437.

[16] D. Ratner and M. Warmuth. *Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable.* AAAI-86, 168–172.

[17] A. Reinefeld and T.A. Marsland. *Enhanced iterative-deepening search.* IEEE Trans. Pattern Analysis Mach. Intell., IEEE-PAMI, July 1994.

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

[19] A. Reinefeld and V. Schnecke. *Performance of PVM on a highly parallel transputer system.* First European PVM Users' Group Meeting, Rome, Italy, Oct. 1994.

[20] T. Römke, M. Röttger, U. Schroeder and J. Simon. *An Efficient Mapping Library for Parix.* Procs. ZEUS'95 Workshop on Par. Programming and Computation, Linköping, Sweden (1995).

[21] M. Röttger, U.P. Schroeder and J. Simon. *Virtual Topologies Library for PARIX.* University of Paderborn, Tech. Rep. tr-ri-94-148, 1994 (available via ftp or www).

[22] P.M.A. Sloot, A. Hoekstra and L.O. Hertzberger. *A comparison of the IServer-OCCAM, Parix, Express and PVM programming environments on a Parsytec GCel.* W. Gentzsch, U. Harms (eds), HPCN-94, Munich (1994), Springer Lecture Notes 797, 253–259.

[23] L. Stockmeyer. *Optimal orientations of cells in silicon floorplan designs.* Inform. and Control 57 (1983), 97–101.

[24] V.S. Sunderam, G.A. Geist, J. Dongarra and R. Manchek. *The PVM concurrent computing system: Evolution, experiences, and trends.* Parallel Computing 20, 4(1994), 531–546.

[25] K.V. Viswanathan and A. Bagchi. *Best-first search methods for constrained two-dimensional cutting stock problems.* Operations Research 41, 1993, 768–776.

[26] T.-C. Wang and D. F. Wong. *An Optimal Algorithm for Floorplan Area Optimization.* Proc. 27th ACM/IEEE Design Automation Conf. 180–186, 1990.

[27] S. Wimer, I. Koren and I. Cederbaum. *Optimal aspect ratios of building blocks in VLSI.* 25th ACM/IEEE Design Automation Conference, (1988), 66–72.