Maple on the Intel Paragon

Report

Author(s):
Bernardin, Laurent

Publication date:
1996

Permanent link:
https://doi.org/10.3929/ethz-a-006651673

Originally published in:
Internal report / Swiss Federal Institute of Technology, Computer Science Departement 251

Maple on the Intel Paragon

Laurent Bernardin
Institut für Wissenschaftliches Rechnen
ETH Zürich, Switzerland
bernardin@inf.ethz.ch

October 15, 1996

Abstract

We ported the computer algebra system Maple V to the Intel Paragon, a massively parallel, distributed memory machine. In order to take advantage of the parallel architecture, we extended the Maple kernel with a set of message passing primitives based on the Paragon's native message passing library. Using these primitives, we implemented a parallel version of Karatsuba multiplication for univariate polynomials over $\mathbb{Z}_p$. Our speedup timings illustrate the practicability of our approach.

On top of the message passing primitives we have implemented a higher level model of parallel processing based on the manager-worker scheme; a managing Maple process on one node of the parallel machine submits processing requests to Maple processes residing on different nodes, then asynchronously collects the results. This model proves to be convenient for interactive usage of a distributed memory machine.

1 Introduction

The Intel Paragon [4] is a massively parallel, distributed memory machine. Each node contains up to 3 processors (Intel i860XP running at 50 Mhz). The machine is scalable and contains up to several thousands of nodes. The Paragon at ETH Zürich operates with 160 nodes dedicated to computations, each equipped with 64MB of memory and 3 processors (one of which is a dedicated message passing processor).

The processors within a node can cooperate using a thread library. Communication between the nodes is handled by a fast proprietary message passing library, NX. Other message passing APIs (PVM, MPI) have been implemented on top of the NX library.

Our goal was to port Maple [3] to this machine, so that it could take advantage of the whole computational power of the distributed memory architecture. This effort is comparable to the Sugarbush system [2] which uses a combination of Maple and a C library and allows distributed applications on a network of workstations.

2 The Message Passing Primitives

We have implemented basic primitives for using the NX library [4] from within the Maple user level. This means that by using the message passing model of parallel programming, code written in Maple's programming language can take full advantage of all the nodes of the Paragon. Consider the example from figure 1, usually referred to as a "ring", where a message is passed on from one node to the next until it again reaches the originating node.

```
me := nxcall(mynode()):
n := nxcall(numnodes()):
m := 100:
if me>0 then
      data := nxcall(crecv());
      nxcall(csend(1,data,me+1 mod n));
else
      data := [seq(i,i=1..m)];
      nxcall(csend(1,data,1));
      back := nxcall(crecv());
      if sent<>back then printf("transmission error!\n") fi;
fi:
```

Figure 1: Ring Code

Note that all the calls to the NX library are wrapped by a new Maple function, `nxcall`. This function then parses the function name given to it as an argument, does some necessary conversions and calls the appropriate function from the NX library. The primitives that we implemented are:

- `mynode()` which returns the node number of the executing node.
- `numnodes()` which returns the total number of nodes accesible to the current process.
- `csend(type,data,node)` which sends `data` to the node with number `node`, tagged with an arbitrary 32-bit integer `type`.
- `crecv(type,node)` which receives data from node `node` tagged with `type`. Both `type` and `node` are optional. If they are either missing or equal to -1, a message from any node and of any type is received.
- `lastnode()` returns the last node from which a message was received.

An arbitrary Maple structure can be sent from one node to another. As a Maple structure is internally a directed acyclic graph (DAG), these structures have to be linearized before they can be sent over a sequential channel and the DAG has to be reconstructed after receiving such an encoded message. For this transmission we use the same format that Maple uses for saving its structures to a file in a compact form (called the dot-m format). Although this encoding is not optimal in terms of space, it is easy to use and reasonably efficient to convert to and from.

In order to get an idea of the overhead of encoding Maple data structures when sending over a fast channel from one node to another we ran the above ring test for varying message sizes (m=0,100,1000,10000) and different number of nodes

(4,8,16,32,64,128). Figure 2 summarizes our timing results. (*) entries represent times that are too small to be measured reliably. The remaining figures are times in milliseconds, averaged over five runs and divided by the number of nodes in use.

	4	8	16	32	64	128	
0	(*)	70	66	82	109	118	
100	(*)	70	78	101	116	124	total
1000	88	116	137	150	175	174	
10000	827	760	776	794	822	835	
0	(*)	(*)	0.0	0.0	0.0	0.0	
100	(*)	(*)	6	6	6	6	
1000	(*)	(*)	55	53	55	55	pack/unpack
10000	(*)	(*)	644	650	658	659	
0	(*)	(*)	0	0	0	0	
100	(*)	(*)	63	94	109	117	
1000	(*)	(*)	1	3	7	15	transmission
10000	(*)	(*)	2	5	11	23	

Figure 2: Ring Timings

We see that the time needed for a message roundtrip increases linearly with the number of nodes as expected. We also see that the overhead of packing and unpacking a Maple data structure grows linearly with its size. For large messages this overhead dominates the transmission times. However, the ratio between communication and encoding overhead is still reasonable and justifies the use of a massively parallel machine over using a network of workstations where the transmission cost would be an order of magnitude larger.

For zero-length messages we get an overhead that is a lot larger than the message passing latency of the Paragon itself which is around $3\mu s$. This is due to the overhead of a procedure call in the Maple language interpreter.

3 An Application: Multiplication of Polynomials

In this section we will present an application of the message passing model of distributed programming. Our goal is to multiply two univariate polynomials modulo a large prime. The Maple code in figure 3 specifies the problem of multiplying two random dense polynomials of degree n modulo an n-bit prime. These kinds of computations arise for example in univariate factorization [6].

```
n := 500; # or:  n := 1000
p := nextprime(trunc(evalf(2^n*Pi,n)));
a := modp1(Randpoly(n),p);
b := modp1(Randpoly(n),p);
r := modp1(Multiply(a,b),p);
```

Figure 3: Multiply two polynomials modulo a prime

We use Karatsuba multiplication [5] for polynomials down to degree 25. Given enough nodes we distribute two of the three multiplications needed after every subdivision to different processing nodes. Figure 4 shows our timing results in wallclock seconds for $n = 500$ and $n = 1000$ and for 1, 3, 9, 27 and 81 nodes. The speedup is computed as $\frac{\text{Time on one node}}{\text{Time on k nodes}}$ and the efficiency is $\frac{\text{Speedup}}{\text{Number of nodes}}100$. We also give the time taken on a SparcStation 20/51 as a reference value.

	n=500			n=1000		
# of Nodes	Time (s)	Speedup	Efficiency	Time(s)	Speedup	Efficiency
1	91	1.0	100 %	954	1.0	100 %
3	33	2.8	92 %	329	2.9	97 %
9	13	7.0	78 %	124	7.7	85 %
27	7	13.0	48 %	52	18.3	68 %
81	8	11.4	14 %	34	28.1	35 %
Sparc	53			481		

Figure 4: Karatsuba Timings

We can see that for this particular application, the Paragon outperforms a state-of-the-art workstation already using 3 nodes. Note however the poor performance of the $n = 500$ problem when using 81 nodes; The original polynomials of degree 500 are subdivided into pieces of degree 31. For degrees as small as this the overhead of the data transmission becomes too large. This is to be expected especially because at degree 25 the sequential Karatsuba algorithm also becomes ineffective for 500-bit coefficients.

4 An Interactive Parallel Server Model

Given the message passing primitives described in section 2, we could implement a manager-worker based model of distributed computation using only Maple's user-level language. This implementation basically provides two commands:

`h := submit("...");` asynchronously submits a string containing arbitrary Maple instructions to any node and returns a handle, `h`, for futur reference to this job.

`r := result(h);` retrieves the result of the computation referenced by the handle . This function blocks until the result is available. If `h` is omitted, the result of the first computation that becomes available is returned.

This pair of routines provides a nice way of interactively using a massively parallel machine from within a computer algebra system. When a job is submitted, any idle node is selected and sent the request. If no node is available, the request is queued. Whenever the result of a computation is successfully retrieved from a node, the first entry in this queue is submitted to that node.

This model can also be used in parallel programs. However its usefulness is reduced to a restricted class of problems because the computation can only be subdivided once at the toplevel as the worker nodes can not act as managing nodes themselves. For most parallel programs it is therefore more efficient to use the message passing primitives directly.

The worker nodes run the small piece of Maple code from figure 5.

```
do
      got := nxcall(crecv());
      if got="quit" then break fi;
      result := parse(got);
      nxcall(csend(1,result,0));
od
```

Figure 5: Worker Code

The manager node maintains a list with the status of all its worker nodes. The `submit` command queries that list to find an idle node, sends the string containing the Maple commands to execute to the remote node and stores the reference number for this job in the node list before returning it. If no node is avaliable the command string is appended to the list of pending jobs. The code for the `submit` command is detailed in figure 6.

```
submit := proc(s::string) global _nodes, _jnr, _pending; local i;
      for i from 1 to nops(_nodes) do
            if _nodes[i]=0 then
                  if not nxcall(csend(1,s,i)) then
                        ERROR("Could not submit job");
                  else
                        _jnr := _jnr+1;
                        _nodes[i] := _jnr;
                        RETURN(_jnr);
                  fi;
            fi;
      od;
      _jnr := _jnr+1;
      _pending := [op(_pending), [s,_jnr]];
      RETURN(_jnr);
end:
```

Figure 6: Submit Code

The `result` function can take a job reference as an argument. In this case, the manager node tries to retrieve the result of the computation corresponding to this reference. If this computation has not yet been started and is still in the list of queued jobs, an error is issued. If, on the other hand, the computation is in progress on some worker node, the manager node is blocked until the computation is completed. If the worker node is done, the result of the computation is returned.

The reference argument to the `result` function can also be omitted. In this case the result of any node that has already completed its job is returned. If no such node exists, the manager blocks until the first worker node completes its computation.

A simplified version of the `result` function (missing some of the error handling) is given in figure 7.

```
result := proc() global _nodes, _pending; local res,n;
      if nargs=0 then
            res := nxcall(crecv(-1));
            _nodes[nxcall(lastnode())] := 0;
            RETURN(res);
      else
            n := args[1];
      fi;
      for i from i to nops(_nodes) do
            if _nodes[i]=n then
                  res := nxcall(crecv(-1,i));
                  if nops(_pending)=0 then
                        _nodes[i] := 0;
                  else
                        j := _pending[1];
                        nxcall(csend(1,j[1],nxcall(lastnode())));
                        _pending := _pending[2..-1];
                        _nodes[nxcall(lastnode())] := j[2];
                  fi;
                  RETURN(res);
            fi;
      od;
      ERROR("illegal handle");
end:
```

Figure 7: Result Code

5 Porting Problems

The Maple kernel is started simultaneously on all the participating nodes of the Paragon. For this step we used the support functions of the NX library. Because of this we had to identify one node which would handle interactive user input to avoid having the nodes compete over lines from stdin. Our choice was to have only the node number zero output the Maple logo and handle interactive user input. Commands meant to be executed by all the Maple processes on all the nodes have to be put into a file whose name is given as a command line option when Maple is started. Once the commands from this file are exhausted the Maple process on node zero waits for user input while the processes on the other nodes are terminated.

6 Conclusions and Future Work

We have presented our results from porting the computer algebra system Maple to the Intel Paragon. We have seen that we could extend the Maple kernel with a small number of basic message passing primitives and achieve reasonable performance. On top of these primitives, parallel programs can now be written entirely using Maple's user level programming language. We have proven the practicability of this approach by parallelizing a standard operation, polynomial multiplication using these primitives. For polynomials of degree 1000 we saw that our approach scales to at least 81 nodes of computation.

We have also provided Maple code for implementing a simple interactive server for driving distributed computations. This server can also be used by Maple programs, for applications that favour a manager-worker approach to parallelism. A subject of further work is to enable the worker nodes to act themselves as managing nodes. This extension will make our server useful for a wider class of applications.

Note that our Maple port does not take advantage of the second CPU available on each node. Changing this would mean converting the Maple kernel into a multithreaded application. This seems to be a non-trivial task an will be the subject of further research.

Another area of improvement is the encoding that is being used for transmitting Maple data structures over a sequential channel. The OpenMath project [1] might produce an encoding that is more compact and that allows faster parsing.

We also plan to use MPI instead of NX in the future. This will allow us to be independent of the Paragon and use Maple on, for example, workstation networks.

References

[1] Abbot, J., van Leeuwen, A., and Strotmann, A. Objectives of OpenMath. Technical Report 12, RIACA, June 1996.

[2] Char, B. W. Progress report on a system for general-purpose parallel symbolic algebraic computation. In *ISSAC '90: Proceedings of the international symposium on symbolic and algebraic computation* (1990), S. Watanabe and M. Nagata, Eds.

[3] Char, B. W., Geddes, K. O., Gonnet, G. H., Leong, B. L., Monagan, M. B., and Watt, S. M. *Maple V Language Reference Manual.* Springer-Verlag, 1991.

[4] Intel Corporation. *Paragon System User's Guide*, Apr. 1996.

[5] Knuth, D. E. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming.* Addison Wesley, 1981.

[6] von zur Gathen, J. A polynomial factorization challenge. *SIGSAM Bulletin 26*, 2 (1992), 22–24.