PRP - Parallel Recursive Procedures

Arne Maus,
Dept. of Informatics, University of Oslo
email: arnem@ifi.uio.no

Torfinn Aas,
Norges Bank, P.O.Box 1179, 0107 Oslo
email:torfinna@ifi.uio.no

October 17, 1995

Abstract

It is believed that writing parallel programs is hard. Therefore a paradigm for writing parallel programs that is not much harder than writing a sequential program is in demand. This paper describes a general method for parallelizing programs based on the idea of using recursive procedures as the grain of parallelism (PRP). This paradigm only adds a few new constructs to an ordinary sequential program with recursive procedures and makes both the underlying hardware topology and in most cases also the number of independent processing nodes, transparent to the programmer. The implementation reported in this paper uses workstations and servers connected to an Ethernet. The efficiency achieved by the PRP-concept must be said to be good with a more than 50% processor utilization on the two problems reported in this paper. Other implementations of the PRP-paradigm is also reported.

Index Terms: Parallel programming, recursive procedures, programming paradigms, distributed computing.

1 Introduction

Writing parallel programs adds levels of complexity on top of solving the same problem on a sequential uniprocessor. This stems mainly from :

1. The need for learning and using additional language constructs - i.e learning and using a programming paradigm for writing parallel programs.
2. The problem of identifying and breaking up the original problem into 'independent' subtask to make it executable on more than one processor. Maybe the most difficult problem is the communication and synchronization between these subtasks.
3. Often the parallel solution has to take strongly into account the topology of the underlying hardware, i.e. the number of processors and their interconnect.

4. The parallel machines are few and far between. Hence the operating system support and programming development tools are definitely not as good as those found on more conventional platforms.

As a consequence of these and other problems, development of solutions to problems using parallel machines are more error-prone, cost more and take longer time. Since for an end-user, the only thing that distinguishes these solutions from sequential solutions is their speed; their late and costly development make them often compete badly with less expensive and more reliably developed solutions using newer and faster generations of single-CPU computers. Improvements are needed in all these problem areas. No standard solution to the 'parallel programming problem' has emerged, and new and more developer-friendly paradigms for writing such programs are needed.

This paper presents a new parallel programming paradigm that tries to simplify the programmer's task for the first three areas for some fairly large classes of problems. This is a report from a master thesis [Aas 94] at the University of Oslo and partly from follow up projects.

2 Parallelizing of programs

The efficient development of parallel programs has been attacked along many lines and good overviews can be found in standard textbooks as [Hennesey and Patterson 94] and [Foster 94] and in survey papers like [Andrews 91] and [Carriero 89]. We only note here that the line of attack has been diverse:

- To build special purpose machines, like vector processors, data-flow machines, hypercubes, and more recently symmetric multiprocessors (SMP) and then make program paradigms and tools to suit these machines.
- To start from the process concept with message passing and/or synchronizing primitives like semaphores and monitors from operating system theory and often include possibilities for such parallelism into programming languages like Ada and Occam.
- To make the process almost automatic by making minor changes to existing programs typically in FORTRAN. This is often done by calling special libraries for performing parallelized matrix operations or by changing the user program by parallelizing loops etc. This direction, which definitely has the highest number of users, are now standardizing a new HPF (High Performance FORTRAN) [HPFF 94].
- To make a completely new programming paradigm like LINDA [Carreiro 89b]
- To hide the underlying hardware, we see standard libraries like PVM [Geist et al. 94] and MPI [Pacheco 95] to get a clear split between program and implementation - a hardware abstraction layer. Existing programs can then more easily be ported to new computers and new topologies.

The idea presented in this paper uses elements from the last four approaches.

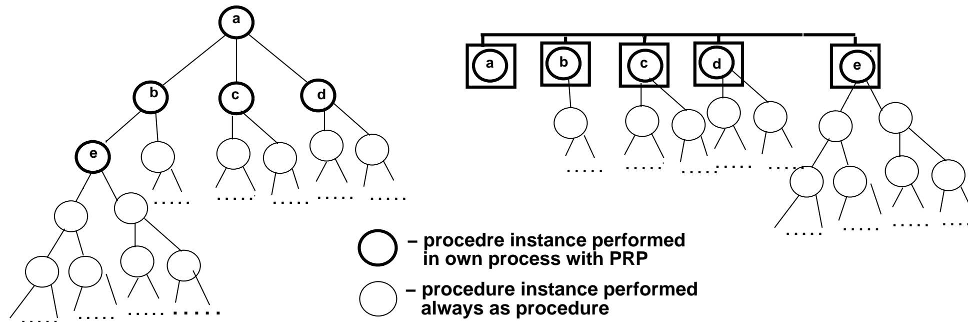

Fig 1 (a) Recursion three from sequential algorithm

Fig 1 (b) Recursion threes when using PRP–consept

Figure 1: *The PRP distribution of recursive procedure instances to the set of available processors*

3 The PRP-concept and related work

The idea of using recursive procedures as the unit for parallelizing was put forward in [Maus 78], but not implemented before 1994 in a master thesis [Aas 94]. The idea is basicly that we have a number of processors connected with some kind of communication channel and a sequential recursive solution that generates dynamically a number of procedure instances. In a PRP-system then each time a new procedure-instance is generated, if there is some idle processor in the system, then the procedure-instance turns into a process that migrates to that idle processor. If all available processors are busy, then the procedure-instance is created as an ordinary procedure-instance on the same processor where the recursive procedure call was made and executed with the same efficiency as an ordinary recursive procedure.

The above picture sets out the general idea. How we administer the program and the pool of idle processors might, as we will later see, slightly modify this picture.

This is illustrated in fig. 1. In fig 1a we see an ordinary sequential recursion tree performed on a single processor. In fig 1b we assume as an illustration 5 interconnected processors, and see how the top 5 procedure instances are executed in its own process on each of the 5 processors, and most importantly how the recursion tree is split and the total work is split. We note that the more time consuming activities (process generation and communication on the net) is only performed for a limited number of procedure instances at the top of the recursion tree. The rest of the recursive calls are made at full speed in each processor.

The idea of using recursive procedures as the unit of parallelism for imperative programming languages, has been found in one other paper [Horowitz and Alessandro 83]. This is a theoretical paper with no implementation. Their concept also differs from the PRP-concept in the respect that *all* the recursive procedure instances are transformed into processes if the subproblem to be solved by this procedure instance is judged *not* to be 'small' (by some user defined test). As described above, in the PRP-concept, there exists only one recursive procedure instance transformed to a process in each processor.

Parallel recursive procedures in functional languages has been described by Hudak [Hudak

86], [Hudak 91]. Both papers present the 'Para-Functional Programming style'. This concept differs from PRP in many respects. First the programmer has to explicitly name the processors that the various functions shall be performed upon, either by absolute number or by relative address (left, right, up down, etc.) each time a procedure migrates as a process. In this 'Para-Functional Programming Style', the parallel execution of function also relies heavily on the fact that no side-effects occur in functional languages. Both the *ParAlfl* and *Haskell* languages presented in these papers have been implemented but no performance figures are presented.

3.1 'Breadth first' versus 'depth first'

Ordinary recursive procedure calls progress from-left-to right and depth-first. That is not always the case in the PRP-concept.

In order to start the parallelism in PRP, all procedure instances called from one procedure that are transformed to new processors as processes, start logically at the same time. That is, part of the recursion tree must be called and started breadth-first. In fig 1 procedure 'a' must call and start 'b', 'c' and 'd' in parallel, breadth first. Then 'b' in turn must start 'e' and then perform the rest of its recursion depth-first. We see a mixture of first 'breadth-first' (the processes) and then afterwards 'depth-first' (the rest of the recursive instances) traversal of the recursion tree in the PRP-concept. The return of these 'breadth-first' procedure calls come in no particular order to their caller process.

Since the PRP-concept can hide the actual number of processors available, the programmer does not necessarily know which procedure is performed 'breadth-first' and which is done 'depth-first'. The consequence for the programmer is that she or he can not use the return value from one recursive call when determining the parameters for the next recursive call on that level - i.e. when 'a' calls 'c' the return value from the call to 'b' with high probability has not been received by 'a' and hence can not used in any way when 'a' calls 'c' (or 'd',...). Only after waiting for the return of these processes are their return values available.

For some (most ?) algorithms this arbitrary shift from 'breadth-first' to 'depth-first' does not matter, for some it does. PRP is only suitable for the first class of problems.

3.2 Fan-out and the administration of processors

The fanout for a recursive procedure is how many times it calls itself on the next recursion level. For some problems that might be fixed (i.e. Quicksort has a fanout of 2). Other problems have a variable fanout that is determined by the problem (i.e. when traversing a graph, a recursive procedure will typically call itself for each edge going out from a node).

Other problems again can be suited to a predetermined fanout. One example is a set of 13 machines where we want to use PRP to find the prime numbers of the first say the 24 first million integers. The parameters to the recursive calls to the 'findprimenumber' - procedure can then be set such that the first machine searches for prime numbers among the first 2 million numbers, the second machine between 2 and 4 million numbers ,...etc. We have then partitioned our problem into the same number of subproblems (minus one - the root) as we have machines available. We do a full fanout on the problem from the root and on the second recursion level solve the resulting subproblems; maybe more efficiently

with iteration.

We sometimes can speed up the calculations if we signal to our PRP runtime-system the fanout before making the recursive calls. That has to do with an effective load balancing - see next paragraph.

Even though we often want to hide the number of processors available and their topology, we must have a policy for the administration of these machines. The policy chosen in this implementation, is that the top procedure instance owns all the machines. When the PRP runtime system is told the fanout before the recursive calls are made (by setting the system variable `prp_fanout`), all the free machines owned by the calling machine are divided as equally as possible among the called machines. If a called procedure in another machine then receives the ownership of more than one machine (itself), this is repeated on the next recursion level (as with 'b' calling 'e' and itself in fig. 1) until all free machines are allocated to a procedure instance transformed to a process.

We said in the general description of the PRP concept that this procedure instance migrates to an idle processor. What actually happens is that the program is duplicated before startup in all the participating machines. When a procedure instance migrates to an idle processor, then a process in the idle processor that waits for a message with the parameters to the procedure instance is called. This process in turn calls the recursive procedure with these parameters.

3.3 Return values

We noted in s subsection 3.1 that the procedures does not necessarily return in the same order as they are called. The return values upon return are stored in a system array `prp_ret[]`. When a procedure calls itself recursively, then when the i'th such call returns, the returnvalues are placed in `prp_ret[i]` (The implementation reported in this paper only supports the return of a single integer value; later implementations have lifted this restriction).

There must be introduced a special primitive for waiting for the return of the 'breadth-first' procedure calls, `prp_wait`. There are three conditions that an algorithm can specify as parameter to `prp_wait: WAIT-ALL, WAIT-FIRST or` 'Condition'.'Condition' is a boolean expression that might involve the value of the results from the returning procedure (typically waiting for some return value to exceed a certain value). The other wait-parameters are straight forward:`WAIT-ALL` blocks until all procedures that has been called from this procedure has returned and `WAIT_FIRST` blocks the calling procedure until the first called procedure returns. All this waiting is a dummy operation when the procedure calls are performed as ordinary recursive procedures 'depth-first'.

3.4 Additional concepts and reserved words

When programming using the PRP concept, the programmer starts with an ordinary recursive algorithm. In the main part of the program she or he initiates the PRP-system by calling `prp_init()` which takes the name of the available machines as parameters. He then calls the procedure in question with the prefix `prp_call`, ie. result = `prp_call` hamiltonian(...);

Finally he shuts down the system by calling `prp_terminate()`. This disconnects and frees resources used in the communication with the machines indicated in the `prp_init()` call.

The recursive procedure also has to undergo some changes before it can be used. First the procedure must be prefixed with `prp_proc`. (There can be only one `prp_proc` procedure in the current implementation.) Then the procedure must decide how many calls it's going to make in parallel. Often this will be determined by the algorithm or the data at hand. If fanout can be set more or less as we want, it is often best to make as many parallel calls as possible, that is one call for each available machine (minus one - the root machine). The system-call `free_servers()` can be used to find how many available machines there are at this point. The system-variable `prp_fanout` is then set to the desired fanout. It is important to set this variable, because the run-time system will use this value when it divides the ownership of the free machines among the calls - otherwise only one processor is handed out for each call.

Finally, after the calls have been made, the program has to wait for the calls to return. As mentioned earlier, the program can wait for the first call to return, it can wait for all the calls to return or it can wait for a call which makes a condition true.

In case of the first and last, the system-variable `prp_ret` can be used to find which calls returned. The return values can be found in the system-variable `prp_return`. The algorithm can then use these values in whatever calculation and finally return a value to the caller. In case of several processors returning before a condition is satisfied, the table `prp_returned` can be checked to find out who has reurned.

4 An example program - the Hamiltonian Path in a Graph

The problem solved in this example is to determine whether or not there exists a Hamiltonian Path in a graph G - that is a path from node to node along the edges visiting each node once and only once. This is a NP-complete problem. The graph is represented with an integer array G, where G [i][j] = 1 if node i has an edge to node j - 0 otherwise. As we can see in the program, an extra iteration is used to count the number of not investigated ('unseen') neighbor nodes in lines 3 to 5 in the recursive procedure 'hamiltonian'. This is the fanout for this procedure instance and the system variable `prp_fanout` is set. Then each 'unseen' neighboring node is called recursively and the highest return value, i.e. the longest possible distance from this current node through its 'unseen' neigbours is picked up and a distance one longer than that is returned.

The following program example was run on a graph containing 12 nodes resulting in a recursion tree with 191 989 procedure instances and a graph containing 14 nodes with more than 21 million procedure instances.

The program text demonstrates in **bold-face** the additions made to an ordinary recursive solution transforming it to a PRP-solution.

```
/* hp.prp
   PRP-program to determine if  Hamiltonian Path exist in graph
  */
```

```
int G [NODES][NODES]; /* G[i][j] == 1 if edge from node i to node j in G
                                 == 0 otherwise */

int  prp_proc hamiltonian(int node, int seen[])
{
  int i, call_ = 0, max_num, fo = 0;

  seen[node] = TRUE;

  for(i = 0; i < NODES; i++)
    if(G[node][i] == 1 && seen[i] == FALSE)
      fo++;

  prp_fanout = fo;

  for(i = 0; i < NODES; i++)
    if(G[node][i] == 1 && seen[i] == FALSE) {
      prp_call hamiltonian(i, seen);
      call_nr++;
    }

  prp_wait(WAIT_ALL);

  for(i = call_nr - 1; i >= 0; i–)
    if( prp_return[i] > max_num)
      max_num = prp_return[i];

  seen[node] = FALSE;

  return ( max_num + 1);
}

int  main() {
  int number;
  int seen[NODES];

  /* Init. table */
  for(n = 0; n < NODES; n++)
    seen[n] = FALSE;

  prp_init(4, "machineX", "machineY", "machineZ", "machineW");

  number = prp_call hamiltonian(0, seen);

  prp_terminate();

  if(number == NODES)
    printf("Graph G has a Hamiltonian Path");
}
```

5 Test results

Figure 2 and figure 3 report the results of the test runs. The timing could not be made accurate because they were performed on machines with a varying load due to other users logged on to these machines. We see a better than 50 % CPU utilization, a speedup factor proportional to the number of processors, and roughly loosing less than 40% of the theoretical peak performance. The slight drop in efficiency observed from 9 to 11 machines in these figures can be explained by adding 2 less powerful machines (Sparc10) to the pool of available processors (Sparc 20).

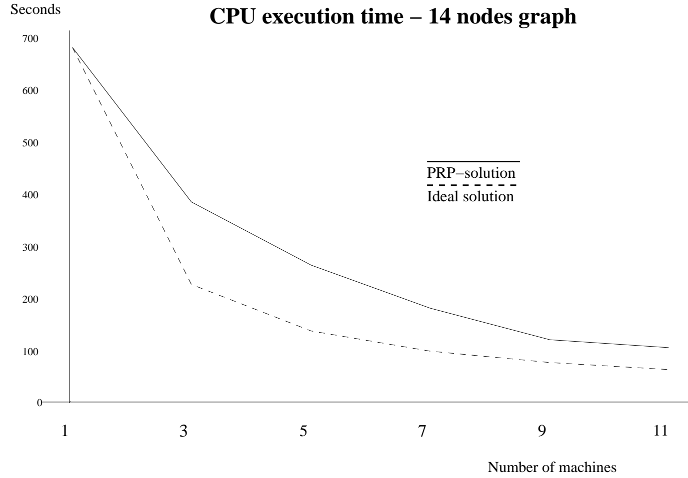

Figure 2: *Execution times as a function of the number of machines used running the Hamiltonian Path program*

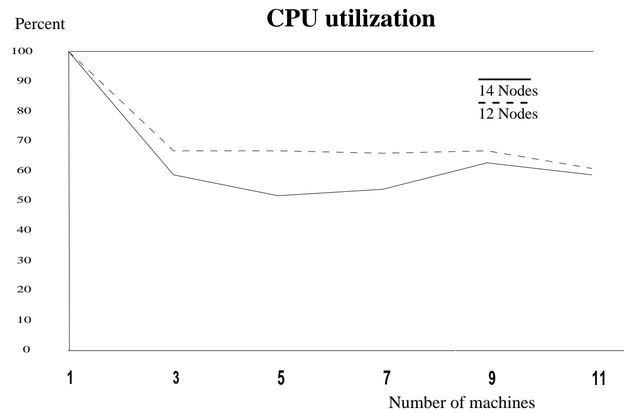

Figure 3: *Mean CPU utilization as a function of the number of machines used when solving the 12 node and the 14 node Hamiltonian Path problem*

These results must be judged very satisfactory since Hennesey and Patterson [Hennesey and Patterson 94] report from 67% down to 1% of the theoretical peak performance (with a mean of approximately 20%) on the 7 reported test runs on five different parallel machines.

6 Strengths and weaknesses of PRP

Because the responsibility of communicating and scheduling the workload is distributed, there is no reason to get a drop in processor efficiency as additional processors are added. While other implementations we have studied often have a clear decreasing processor-efficiency when adding new processors, this could not be observed in the PRP-implementations (at least not within the limited number of machines we had available).

6.1 Global variables and pruning of the search tree

PRP does not support global variables. This of course is a disadvantage, because some of the algorithms (i.e. the Traveling Salesman problem) use global variables for pruning the search tree for search of an optimal solution. The reason for not having global variables is the loosely coupled, distributed architecture PRP can be implemented on with each processor having a copy of the program with all its variables. Alternatively, using RPC-calls to read and set global variables is thought be too inefficient and a common resource responsible for the globals would soon become a bottleneck.

A problem with the implementation of PRP today is that it can lead the programmer to believe that she or he can have global variables - i.e. the the array 'G' in the program example. These variables, however, will become separate instances for each processing node (i.e. one for each subtree in fig. 1), thus an assignment of the 'global' variable in one processor would not influence the 'same' variable in some other processor! Each processing node would have its own local 'global' variable. The effect of many copies of global variables if they are used for optimization (but not for the correctness of the algorithm !) is an open research question.

Our guess is that in most cases the effect of each subtree in each processor having its own local optimum in a local 'global' variable , and then later combining these optima upon return to a true global optimum in the root, would be almost as effective as having one shared global variable throughout the whole computation. This reduces message passing traffic to a minimum. In an Ethernet implementation of PRP, this would in real time be a much faster solution. But again, this is an open research question we are now starting to look into.

6.2 Implementation, the pre-processor and the PRP runtime system

The process of transforming the sequential, recursive solution to a PRP solution is illustrated in fig. 4. As seen from this figure and the program example, the target and implementation language is 'C'. The user starts with an singe-processor, recursive solution ('appl.c'). The user then manually makes the changes necessary to include prp-functionality (`prp_call, prp_wait,...`). The resulting file ('appl.prp') is then sent to the PRP pre-processor which, without any user interference, does:

1. The reserved `prp_` words and the `prp_`recursive procedure are recognized. Two programs are constructed, an 'admin.c' for running with user I/O on the machine that starts the execution (the root), and 'server.c' for the other machines. Also the

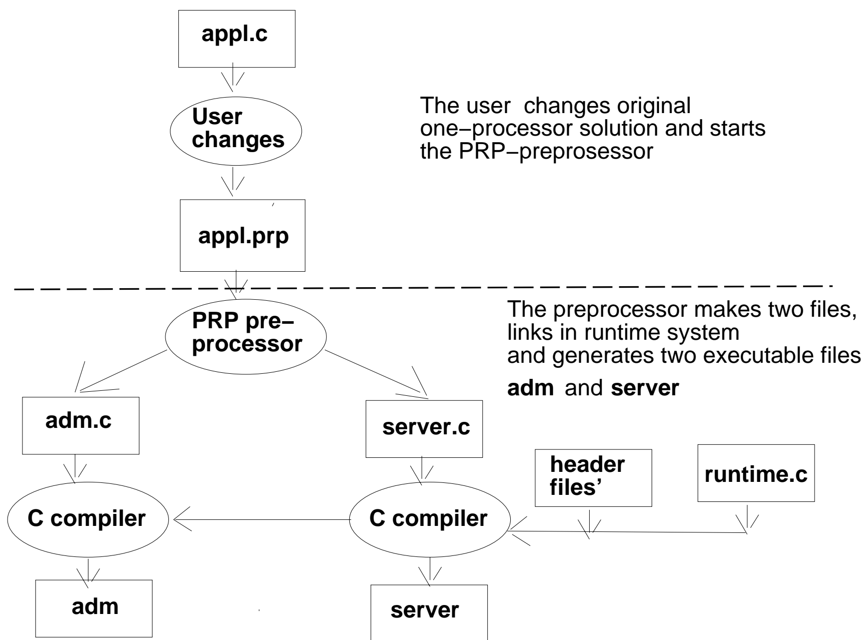

Figure 4: *From ordinary sequential recursive solution to PRP-solution*

prp_procedure is made in two versions in these in these two programs, one for acting as a process and one for acting as an ordinary recursive procedure stripped of all its prp_ functionality. The parameters to the recursive procedure is also recognized, and a parameter-struct is made for packing/unpacking and sending the packet of parameters by RPC-calls on the Ethernet.

2. The two programs 'admin.c' and 'server.c' are sent through the C-compiler and linked to the prp_ runtime system for making the two executable files 'server' and 'admin'.

Finally, the user starts the parallel computation by running 'admin'. As an initiation task, 'admin' copies 'server' to all the other machines participating in this computation. The 'server'-programs then start as processes on these machines waiting for a RPC-call. 'Admin' finally starts the recursive computation as specified by the user.

7 Further work

The PRP-concept has been implemented on the Intel Paragon machine and on a four-CPU SMP machine (SGI Challenge) with good performance results. Preliminary test runs show a better than 50 % CPU utilization on the traveling salesman problem where only local copies in each processing node, as described above, of the shortest-distance-so-far has been used for pruning the computation..

Two new master of science thesis on refinements to the original solution, one porting to a heterogeneous net of computers and the other on porting PRP on top of the PVM library, has been started.

8 Conclusions

A paradigm, PRP, for writing parallel programs using recursive procedures in an imperative programming language has been introduced and an implementation and some test results has been reported.

Since PRP is an addition to a well known programming paradigm, we will claim that it is relatively easy to use mainly due to the lack of need for difficult synchronization. We also claim that it scales well with the number of processing nodes available, that it hides the underlying topology of the processor interconnect from the user and that it is easy to achieve a satisfying speedup compared with other parallel programming paradigms.

PRP is however not the solution to every parallel programming problem. We first demand that the problem has an effective recursive solution with a fanout greater than one, and also that such a solution does not rely on a strict depth-first search of the sequential recursive procedures. The use of global variables are also not supported directly, but as indicated in this paper, copies of the global variables in each machine might in some cases be almost as good as true global variables.

To summarize, we believe that it is easy to formulate effective parallel solutions in the PRP-paradigm to a wide class of problems such as search problems, Graph problems and among these also the class of NP problems.

9 Acknowledgment

We would like to thank Stein Jørgen Ryan, Yan Xu and Arne Høstmark for their work on the PRP concept an to their valuable comments to earlier drafts of this paper.

References

[Carreiro 89] Carreiro N. and Gelernter D. *How to write Parallel Programs: A Guide to the Perplexed.* ACM Computing Surveys, Vol 21, No. 3, Sept. 1989 page 323-357.

[Carreiro 89b] Carreiro N. & Gelernter D. *Linda in Context.* Communication of the ACM, Vol 32, April 1989.

[Foster 94] Foster Ian T. *Designing and Building Parallel Programs .* Addison-Wesley, Reading, Mass. 1994.

[Geist et al. 94] Geist A., Beguelin A., Dongarra J, Jiang W., Manchek R. and Sunderam V. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press 1994.

[Hennesey and Patterson 94] Hennesey J.L. and Patterson D.A. *Computer Organization and Design* p.639-640, Morgan Kaufman Publishers, San Mateo, Calif. 1994

[HPFF 94] High Performance Fortran Forum *High Performance Fortran Language Specification, ver. 1.1.* Rice University, Huston , Texas. November 10, 1994.

[Horowitz and Alessandro 83] Horowitz E & Alessandro Z. *Divide-and-Conquer for Parallel Processing.* IEEE Transactions of Computers. Vol. C-32, Nr 6 June 1983. p 582-585.

[Hudak 86] Hudak P. *Para-Functional Programming.* IEEE Computer. August 1986. p 60-70.

[Hudak 91] Hudak P. *Para-Functional Programming in Haskell.* in Szymanski B. K.(ed.)*Parallel Functional Languages and Compilers.* ACM Press, New York 1991

[Maus 78] Maus A. *Språkprimitiver og en logisk trestrukturert multiprosessor- Idenotat nr. 1 og 2* (in Norwegian) Norsk Regnesentral 1978

[Pacheco 95] Pacheco P. S *Programming Parallel Processors Using MPI.* San Francisco, CA, Morgan Kaufman (to appear) 1995.

[Aas 94] Aas T. *Parallelle Rekursive Prosedyrer.* Master Thesis (in Norwegian) Dept. of Informatics, Univ. of Oslo, 1994.