The DYPAC System: A Dynamic Processor Allocation and Communication System for Distributed Memory Architectures

Erich Schikuta

CRPC-TR93359
November 1993

Center for Research on Parallel Computing
Rice University
P.O. Box 1892
Houston, TX 77251-1892

THE DYPAC SYSTEM: A DYNAMIC PROCESSOR ALLOCATION AND COMMUNICATION SYSTEM FOR DISTRIBUTED MEMORY ARCHITECTURES

Erich Schikuta[1]

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston , TX 77251-1892

Abstract

In this report a DYnamic Processor Allocation and Communication system (DYPAC system) is presented, which establishes a programming model for the development of parallel programs independently of the underlying parallel system architecture.

The DYPAC system provides functions for the creation, deletion and administration of processes and the installation of communication lines between them.

Aim of the project was to create a programming tool for the development of parallel software systems with a high degree of portability. This was reached by a high level functional framework, which is independent of the underlying operating system and the physical hardware architecture.

1. Introduction

Message passing is the commonly used method for information sharing of parallel processes on distributed memory machines. In practice it has proven to be effective, easy to understand and to implement. The problem of portability arises with the development of parallel software systems, because of the variety of the available parallel hardware architectures and the accompanying system software. The frameworks of the proprietary message passing packages of the different parallel hardware architecures differ considerably. A number of packages exist (e.g. Express [Parasoft90], PVM [Dongarra91][Sunderam90], Zipcode [Skjellum92] or the MPI initiative [MPIF93]), which try to provide a common platform for different architectures; but generally they lack in availability, conciseness and simplicity. A quite comprehensive survey of parallel programming tools can be found in [Cheng93].

In this paper a dynamic processor allocation and communication system is proposed. It overcomes the mentioned problems and supplies a concise, simple and general process administration and data communication package for the development of highly portable parallel software systems.

2. Characteristics of the DYPAC system

The main difference of the DYPAC system to existing packages is that the physical architecture of the underlying hardware architecture is totally hidden from the program development process.

---

1 Authors permanent address: Erich Schikuta, Institute of Applied Computer Science, Dept. of Data Engineering, University of Vienna, Rathausstr. 19/4, A-1010, Vienna, Austria

The developer is supposed to know nothing about the characteristics (capabilities or deficiencies) of the physical system. This paradigm guarantees a unrestricted portability of the developed software system. For instance, the actual number of available processors is not known during the developing process.

The DYPAC-system provides a standardized process handling interface, which is basically oriented on the widely used interprocess communication package of the UNIX system. It contains functions for

- process handling,
- process communication,
- process information and
- system administration.

(See the Appendix for a listing of the available functions.)

The process model of the DYPAC system is dynamic, which means that during the execution of processes new processes and communication lines are established dynamically. This leads to a hierarchical execution model in the sense that a new process, a "child" process, is always created by a unique "father" process. In every system a "root" process exists, which is the origin of all further created parallel processes. Processes can be created and destroyed. Each process has a unique identification, which is passed to its father process during the creation process. The DYPAC system allocates available physical processors of the underlying system as needed.

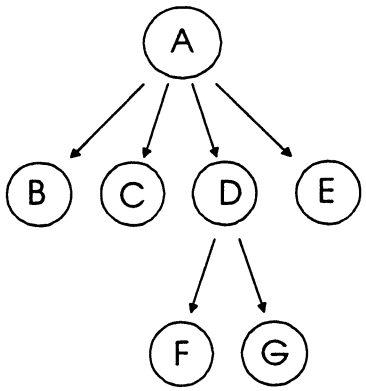

Figure 1: hierarchical execution model

In the above figure A denotes the root process, with its children B, C, D and E. Further D is father to 2 additional processes F and G.

Information is synchronously or asynchronously passed via messages between processes, which are distinguished by the unique process identifier. Synchronous means in the context of the DYPAC system that a process is blocked until the arrival of the message at the addressed process. By this blocking mechanism the synchronization of processes is achieved. Each call to a message handling function returns a unique message identifier. These identifiers can be used to get information about the status of pending communication operations.

Child processes can query the process identifiers of their father process. By this recursively applied operation processes can get information about the process structure of the software system. This information can be used to establish arbitrary communication lines.

The DYPAC system guarantees a high degree in portability. A parallel software system developed with the DYPAC system can be transferred from one physical system architecture to another without change of the source code and/or (more important) the programming paradigm. The DYPAC-system has to be adapted and implemented on the specific system once. The software developer is provided with a general and simple process administration and communication model and uses always the same programming paradigm. Therefore the development of parallel programs is simplified dramatically and the arising development time and costs are reduced. It is possible to port the developed software system not only to parallel but also to sequential system (like a conventional UNIX system) without a change. The implementation of the DYPAC-system on a one-processor system sequentiates the processes on a single processor using the multitasking capabilities of the underlying operating system.

3. The DYPAC system structure

The DYPAC-system consists of the RES/REI (n.) modules, the request executing server (RES) and the request expressioning interface (REI). The following figure illustrates the situation of a software system running on different hardware architectures:

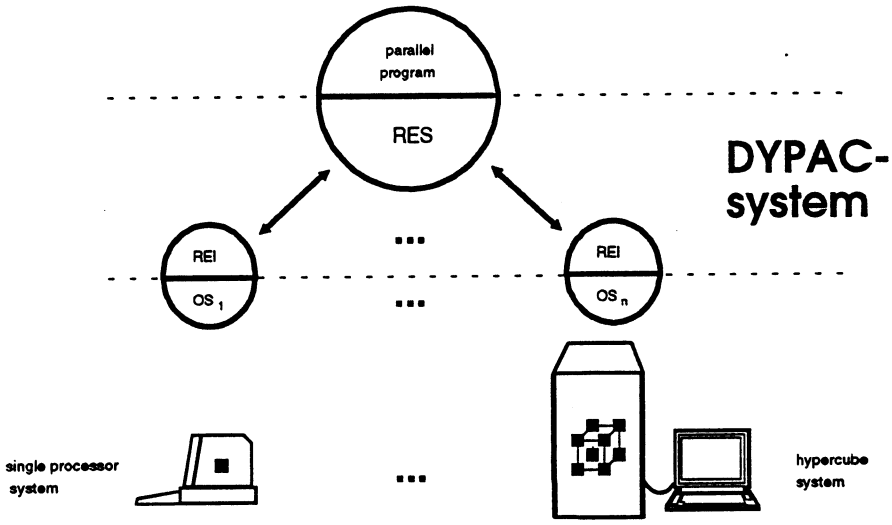

Figure 2: DYPAC structure

The RES builds the interface of the DYPAC system to the underlying physical hardware architecture and the REI provides the procedural interface to the software system. With the implementation of the DYPAC system on different hardware architectures, only the RES has to be adapted. The REI doesn't change and guarantees the unrestricted portability of the software system to different computing platforms.

The RES is a server process running on a single (dedicated, but not fixed) processor of the underlying hardware of the parallel system and handles the requests of the REI. It has to fulfill two different tasks:

- the handling of the processes and the administration of the physical processors of the underlying hardware system
- the installation of communication lines between processes running on different processors

3.1. Administration of the physical processors

The processor allocation server administrates the physical processors of the underlying hardware. The server concentrates the requests of the REI and allocates processors as requested. To fulfill this task it chooses appropriate processors of the hardware regarding to the actual state of the underlying system. This is done by a scheduling algorithm, in that sense that

- the workload is equally spread (all running processes of a processor are accounted and profiled)
- the capabilities of a single processor (like type of the processor, available mathematical coprocessor, connected disk node, etc.) are exploited to increase the performance of the software system. Further the requests of the REI for special processor characteristics are accomplished.
- the processes are distributed arbitrarily, if none of the conditions mentioned above is applicable.

The processes are allocated to processors by using the functionality of the underlying operating system. If the system software of the hardware architecture supports multitasking (e.g. Intel IPSC/2), more than one process can be assigned to a processor. If it doesn't has multitasking capabilities (e.g. Intel IPSC/860), each process is uniquely assigned to a single processor. All implementation specific inherent properties of the RES are totally hidden from the programmer and the developed program. The DYPAC-system tries to guarantees an evenly distributed workload and tries to maximize the throughput of the system.

3.2. Installation of communication lines

The interprocess communication interface of the REI is basically similar to the well know standardized communication protocol of UNIX systems [Rochkind85]. Each call to a message communication function returns a unique message identifier. These message identifiers allow to check the status of pending transfers. It is also possible to emulate signals with this construct. This provides the possibility to synchronize parallel processes.

One important fact is that the DYPAC system uses the available communication and data transportation facilities of the underlying operating system. In the existing implementations the DYPAC system (if possible) doesn't perform any physical data transfer itself. It administrates and uses the capabilities of the process communication package of the underlying system software only.

Sometimes, for special communication methods, like synchronous data transfer, a certain communication overhead is not avoidable. This is based on the fact that the synchronous message transfer (in the definition of the DYPAC system) has to guarantee the arrival of the message at the addressed process and not only the successful initiation of the sending process. If this functionality is not directly supported by the underlying system facilities (like e.g. in the Intel hypercube systems), it is established via a logical handshake protocol to emulate the requested functionality.

This approach is a warrant that the performance of the software system is only affected minimally by the usage of the DYPAC system.

The following communication facilities are supported,

- synchronous message passing,
- asynchronous message passing,
- point to point communication and
- any point communication (broadcasting of messages to a group of processes).

4. Practical examples

The following section gives 2 practical examples, which fitted well to the hierarchical process structure underlying the DYPAC system. The first example describes the implementation of the communication structure in a multiuser database system and the second the inherently parallel evaluation process of a database query.

4.1. The GFDBS system

The DYPAC-system was used for an early port of the Grid File Database System, the GFDBS [Schikuta91], to an Intel IPSC/2 hypercube system. The GFDBS is a specialized data base system [Date86], which supports Gridfiles [Nievergelt84] as internal data structure. The original GFDBS was implemented on a one-processor, UNIX based, workstation. The GFDBS consists of different interacting processes, like a data base server (the GFDBS server), different interface server (SQL/QBE interface), a query optimizer (ICO, intermediate code optimizer) and a query plan executor (ICI, intermediate code interpreter). All these processes run inherently independent and information is transferred via message queues. The following picture shows the system structure of the GFDBS system:

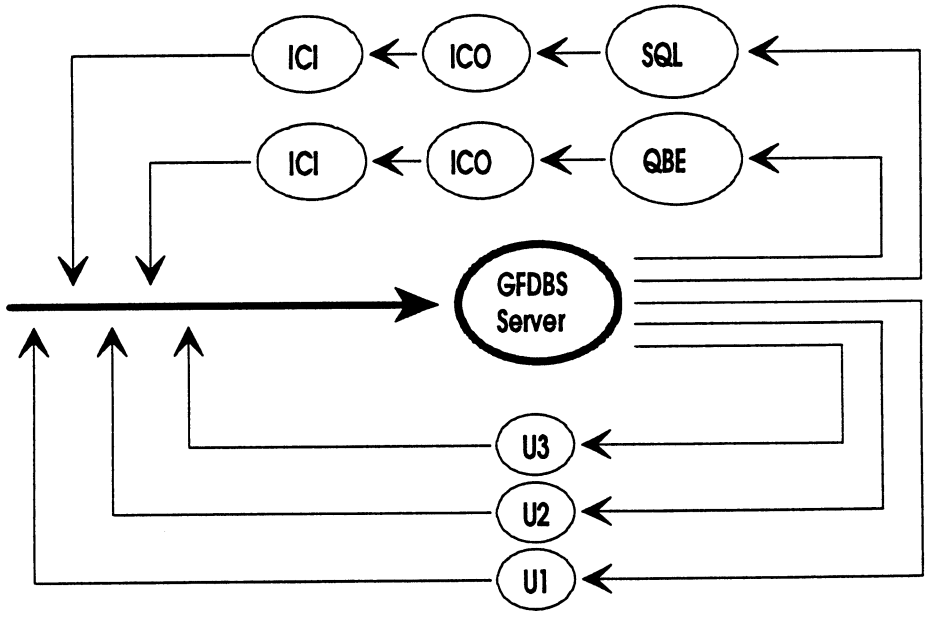

Figure 3: GFDBS structure

The port to the hypercube system was straightforward without difficulties. The original source code of the GFDBS was only changed in respect to the interprocess communication calls, which didn't affect the logical structure of the system in any way. The inherent parallelism of the GFDBS exploited the parallel hardware in a data-driven way and an immediate performance boost was recognizable.

Until now two implementation of the DYPAC exist, one on a conventional UNIX system, where the parallel process structure is sequentiated on a single processor and the standardized UNIX interprocess communication facilities are used, and another one on an Intel IPSC/2 hypercube system with 8 processors, where the available parallel processors and the proprietary communication facilities are exploited. It is planed to do further ports to other parallel system architectures in the near future.

4.2. Query Evaluation

Parallelism can not only be utilized at the high system level but also on a lower database operation level. One example is the evaluation of a database query, where the inherent operator parallelism can be easily exploited to increase the system performance. The following figure shows a database query and its respective evaluation tree:

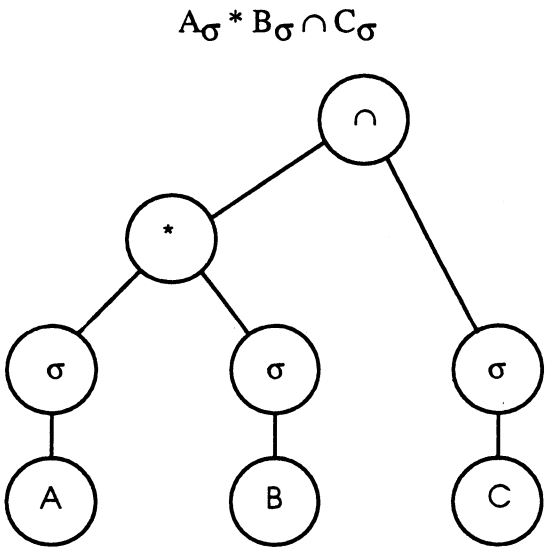

Figure 4: Evaluation tree

The intermediate code interpreter performs a transaction by building up an evaluation graph, where the nodes are the basic operators and the edges are the communication lines, normally pipes. The evaluation of this graph starts at the leaves representing relation identifier. The calculated result tuples of an operation are transferred sequentially along the connection lines to the next operator nodes. The whole evaluation process is done pipelined starting from the leaf nodes in data-driven way (similar to the DB^{++}-system [Agnew86]). The final result is therefore the output of the root node.

5. Results and limitations

The usage of the DYPAC system for the development of parallel software systems guarantees a high degree of portability between different physical hardware platforms.

It can also be used to port existing software with inherent parallel process structure without large effort to parallel hardware architectures. An immediate performance boost is easily reachable with server-client program architectures, as it was described in the GFDBS example. In many cases the existing programs have not to be changed at all. It is sufficient to link them with the DYPAC libraries. Because of the inherent parallelism of the process structure the programs can exploit automatically the underlying parallel hardware.

The author emphasizes that the DYPAC-system is not an automatic parallelizer. It is only a standardized paradigm for parallel program development. If the program developer intends to gain finer parallelism in his program, he has to adapt the program to exploit the possible parallelisms of the used algorithms in his program. The DYPAC-system delivers him a portable and efficient tool to succeed in his efforts.

6. Acknowledgment

I want to thank Eric Wagner for the implementation of the DYPAC-system on the Intel hypercube system and Wolfgang Ristl for the UNIX implementation and the adaptation of the GFDBS. My special thanks goes to the CRPC for providing a stimulating and supportive atmosphere that contributed to this work.

This research was in part supported by the grant J0742-PHY of the Austrian FWF.

7. References

[Agnew86] Agnew M., Ward R., *The DB++ relational database management system*, Proc. of the EUUG Spring Conf., Italy, 1986

[Cheng93] Cheng D.Y., A survey of parallel programming languages and tools, Report RND-93-005, NASA, Ames Research Center, Moffet Field CA, March 1993

[Date86] Date C., *An introduction to database systems*, Vol. 1, Addison Wesley, 1986

[Dongarra91] Dongarra J.J., Geist G.A., Manchek R., Sunderam V.S., A user's guide to PVM, Techn. Rep. No. ORNL/TM-11826, Oak Ridge National Laboratory, July 1991

[MPIF93] Message Passing Interface Forum, Document for a standard Message-Passing Interface (Draft), to be presented at SCP 93, 1993

[Nievergelt84] Nievergelt J., Hinterberger H., Sevcik K.C., *The Grid File: an adaptable, symmetric multikey file structure*, ACM Transactions on Database Systems 9, 38-71, 1984

[Parasoft90] Parasoft Corp., Express C user's guide, Version 3.0, 1990

[Rochkind85] Rochkind M., *Advanced UNIX Programming*, Prentice-Hall 1985

[Schikuta91] Schikuta E., *A Grid File Based Highly Parallel Relational Data Base System*, Proc. 4th ISMM Int. Conf. Parallel and Distributed Computing and System, Washington, D.C., 1991

[Skjellum92] Skejellum A., Smith S.G., Still C.H., The Zipcode systen user's guide - version 1.00, Techn. Rep., Lawrence Livermore National Laboratory, Oct. 1992

[Sunderam90] Sunderam V.S., PVM: a framework for parallel distributed computing, Concurrency: Practice and Experience, 2, 4, 315-339, Dec. 1990

8. Appendix

Reference Guide, Version 1.0[2]

A host program (see the example *init.c*) allocates a group of processors (e.g. a cube at the Intel Hypercube) and runs the communication server and a root process (stated via commandline parameter).The root process can start one or many child process(es), which can also start further child processes recursively.After termination of the root process, the host program releases the cube and the communication server is shutdown.

The programmer can use the following functions to control the message passing and the creation/termination of processes.

chkmsg

checks for asynchronous incoming or outgoing message completion

Synopsis

int chkmsg(mid)

MID mid;

Description

This function checks for the completion of a preceeding asynchronous call to *putmsg* or *getmsg*. The parameter *mid* contains the message identifier of the preceeding operations.

Return values

Returns DOK on completion of the message operation, DERR on a failure (the DCPS variable *derr* is set appropriate).

startcp

Starts a child process

---

[2] History:

Version 0.?	Eric Wagner (9008301), Feb. 1992, Vienna
Version 1.0, ...	Erich Schikuta, April 1993, Houston

Synopsis

PID startcp(name, ptype)

char *name;

PTYPE ptype;

Description

This function starts an executable program *name* on a processor of type *ptype*. The processortype describes the necessary capabilities of the processor. The type ANY defines that no special capabilities is necessary

Return values

The function returns the process identifier of the created child process. On failure DERR is returned and the variable *derr* is set appropriate.

stopcp

stops (terminates) a process

Synopsis

PID stopcp(pid)

PID pid;

Description

Terminates the Process stated by its process identifier pid. The process is removed from its processor.

Return values

Returns the process identifier of the terminated process. On failure DERR is returned and the variable *derr* is set appropriate.

initdypac

initiates and startes the DPCS server

Synopsis

void initdpcs()

Description

Initiates and startes the DPCS server. Has to be the first function call of a DPCS session. Only one server is allowed per session. Which DPCS server is actually started is

determined by the SERVER value in the file *config.h* in the source directory of the DPCS directory. Normally it depicts an executable file *server* in the DPCS installation directory.

Return values

None

killdypac

terminates the DPCS server

Synopsis

void killdpcs()

Description

Terminates the running DPCS server. Has to be the last function call of a DPCS session. The status of the running processes is indeteremined.

Return values

None

putmsg

put a message

Synopsis

MID putmsg(pid, mtype, buffer, buflen, mode)

PID pid;

long type, buflen;

char * buffer;

int mode;

Description

This function sends the value of the variable *buffer* of length *buflen* as a message of type *mtype* to process *pid*. The mode of operation is determinable by the mode parameter, SYNC specifies a synchronous put (the function blocks until the message arrives at process *pid*), ASYNC a asynchronous get (the function does not block).

If the process identifier *pid* is set to ALL (CHILD) the message is sent to all (child) processes (broadcast).

Return value

In synchronous mode the function returns DOK on completion. In asynchronous mode the message identifier of the pending request is returned, which can be used with chkmsg for completion. On a failure the return value is DERR and the DPCS variable *derr* is set appropriate.

getmsg

get a message

Synopsis

MID getmsg(pid, mtype, buffer, buflen, mode)

PID pid;

long type, buflen;

char * buffer;

int mode;

Description

This function gets a message of type *mtype* from process *pid* of length *buflen* into the variable *buffer*. The mode of operation is determinable by the mode parameter, SYNC specifies a synchronous get (the function blocks until the message arrives in the buffer), ASYNC a asynchronous get (the function does not block).

If the message type *mtype* is set to ALL the function receives all messages of any type without restriction. If the process identifier set to ALL (CHILD) the process receives the messages of all (child) processes.

Return value

In synchronous mode the function returns DOK on completion. In asynchronous mode the message identifier of the pending request is returned, which can be used with chkmsg for completion. On a failure the return value is DERR and the DPCS variable *derr* is set appropriate.

getmyid

returns the process identifier of the calling process

Synopsis

PID getpid()

Description

This function returns the process identifier of the calling process.

Return values

Returns the process identifier of the calling process.

getparid

returns the process identifier of the parent process of a process

Synopsis

PID getpid(pid)

PID pid;

Description

This function returns the process identifier of the parent process of the (child) process pid.

Return values

Returns the process identifier of the parent process of the process *pid.* If no parent exist for process *pid* the function returns NOPROC. On failure DERR is returned and the variable *derr* is set appropriate.