Technology

Pak J Sci Ind Res 2002 45 (5) 355-362

SOEPP-Subroutine Oriented Environment in Parallel Programming

Mohammed Yaqub*, S
Shakil Ahmad and Qameruddin Shaikh

PCSIR Laboratories Complex, Karachi-75280, Pakistan
(Received 1 October 1997; accepted 20 August 2001)

Generally, the software for real time parallel processing comprises of system software and the application software. The system software, in turn, comprises a run time executive and a parallel compiler for high-level language interface. This paper presents a novel approach to the design of run time support software and a parallel compiler to facilitate the user with a Subroutine Oriented Environment in Parallel Programming. Using this environment, the user is able to upgrade the parallel processor system for more power and bandwidth by adding more processor boards in the system architecture without modifying or recompiling the system or the applications software.

Key words: Shared memory, Parallel programming, Multi processor system.

Introduction

The development of the microprocessor design technology has made the multi-processor systems (MPS) very famous. The major design requirements of an MPS are: the implementation of the interconnection topologies between the processor and memory modules and between the processor and I/O devices; the development of system software; the inter processor communication and the development of applications software (Hwang and Briggs 1984; Jones 1980; and Tabak 1990). The main jobs of the software of the MPS are: the partitioning of the tasks into several sub_tasks; allocation of tasks to the available processors; providing communication between the processors; shared data protection and the provision of control and management for system activities (Gaski *et al* 1985; Kirrman *et al* 1981).

The existing parallel processing systems do not provide an upgrade path by which the processor boards can be added to achieve more computer power and bandwidth without modifying the software (Riganati *et al* 1984; Stankovic *et al* 1998). There is, therefore, a demand for the system where a user can achieve more computer bandwidth and speed by simply adding processor boards without modifying or recompiling the system software or the application software.

In the proposed strategy, both the run time software and the compiler, cooperate with each other to provide the user with parallel programming environment. The compiler reads the user program with explicit parallelism and generates code for the associated parallel blocks. At run□time, this code produces a table of information and a Ready To Run (RTR) queue, in the shared memory. The run-time executive also uses this information to manage the control for subroutine synchronization; scheduling and shared data protection. An idle processor searches the information table, dispatches the process descriptor (pointer to the subroutine) from the RTR queue and consequently executes it till end (Yaqub 1990).

Proposed software model. Two types of operating system for the MPS are described in the literature (Jayant *et al* 1984; Stankovic *et al* 1998), the master/slave and the separate supervisor in each processing unit. The proposed SOEPP is novel in the sense that the software design uses the combination of the both master/slave and separate supervisor and a subroutine is a building block of execution control. As a master/slave organization, one of the processors executes both sequential and parallel subroutines of the user's application program, while the other processors if connected to the system execute the parallel subroutines only. In addition, the control of execution, in the proposed software model, is centralized for sequential subroutines, while it is distributed for parallel blocks i.e., the executive routine for each processor operates asynchronously during parallel execution. This is due to the fact that every processor connected to the system is provided with run-time executive software.

In the proposed model, the idea of multi-threading is implemented (Midkiff and Padua 1990; Gaski 1985), in which, a subroutine in terms of a thread is scheduled to a processor for its lifetime. In other words, subroutine does not migrate between processors. In contrast to other operating systems (Lister 1981, Dijsktra 1986 and Schwederek *et al* 1986), where the processors are scheduled to the processes and the processes migrate from one processor to the other for execution.

Generally, there are three ways to differentiate between sequential and parallel processes (Schwederek 1986; Midkiff *et al* 1990) Coroutines; Fork & Join; and Cobegin & Coend. For the work described in this paper, the high level language, 'C' (Kermighan and Ritchie 1989; Deital 1985), is used both for

*Author for correspondence

the compiler and as a target language for application software. The function of the 'C' programming language (subroutine) is the building block of control in the proposed design. The user must place the parallel executable functions of 'C' between cobegin and coend. The compiler recognizes the keyword cobegin as the parallel block while the keyword coend is used as the block terminator. In other words, the functions within the parallel block are termed as threads that are scheduled to the processors connected to the system. These threads are executed sequentially by a processor until completion without being interrupted (Yaqub 1991).

Each processor, when idle, competes to access the address of the subroutine from Ready To Run (RTR) queue of subroutines. A processor first performs indivisible bus protocol (Dijstra 1986) which include the locking and unlocking of the shared data. It then inspects the task descriptor in RTR queue and fetches the pointer to the subroutine pointed by the current task pointer. Therefore, a processor cannot decide which subroutine is it going to execute from the RTR queue. It depends upon the order of execution of the subroutines in the parallel block. Only one processor at a time is allowed to access shared data. The other processors if required access to the shared data perform busy-wait. The routines to lock and unlock the shared data are implemented in MC 68020 assembly language using the processor indivisible instruction, TAS (Test & Set) (Beim 1984). The time one processor spends inside the indivisible critical region is very small, the shared data is locked during this time only. On exit, the processor unlocks the bus in favor of the other processors. Furthermore, this lock is not very frequent because only one lock is required for each subroutine in parallel block.

Parallel compiler. The proposed compiler generates code for parallel blocks along with a call for the run time executive and the code for initialization. The compiler code also includes a subroutine call, processor sync. If all the processors involved in the parallel execution have finished their jobs, only then the sequential statements after the "coend", in the application program (if present) are executed. In addition to that, the compiler generates assignment statements for each subroutine inside each parallel block in the user program. These statements, at run time, produce the table of information in shared memory containing all the pertinent data about the subroutine. In the case of more than one parallel block which could appear anywhere in the user program, the same memory segment in the shared memory is reused at run-time, by over writing the new information for the next parallel block.

Run-time support executive software. The run time execution software is designed to support the MPS with any number of single board computers (SBC) based on Motorola MC 68020 and a MC 68881 Floating Point Unit (FPU); designed earlier for parallel programming applications (Yaqub 1991). It supports an application program with explicit parallelism written in the high level language "C" using the UNIX V development system. It also supports the system hardware comprising of a VMEbus shared memory multiprocessor system (SMMPS) with a shared memory board (SMB) and a bus arbiter board including a hardware binary semaphore. The binary semaphore implements the 'test and set' (TAS) flag for system data protection and process synchronization. This flag can be set or software re-setable and accessed on a FIFO (First In First Out) basis. In addition the run-time executive provides communication between the processors by means of shared variable and/or by passing the value parameters.

The system software consists of the run time supervisor software and the associated processes. The run time support software includes run-time library routines; Run time executive routine; and Execution control software. The system software model can be expressed logically as multiple layers from processes to the ready to run queue (Fig 1). The topmost layer in the diagram is the process, which performs the logical tasks required to define the application. This implies that, the subroutines in the user's application program, either sequential or parallel. The next layers below the process comprise the run time executive routine. The run time executive further divides into: the kernel; the data structure and the process manager implemented in MC 68020-structured assembly language for efficient and optimized code. The lowest layer in the logical diagram is the ready to run queue, which is shared by all the processors, involved in parallel processing.

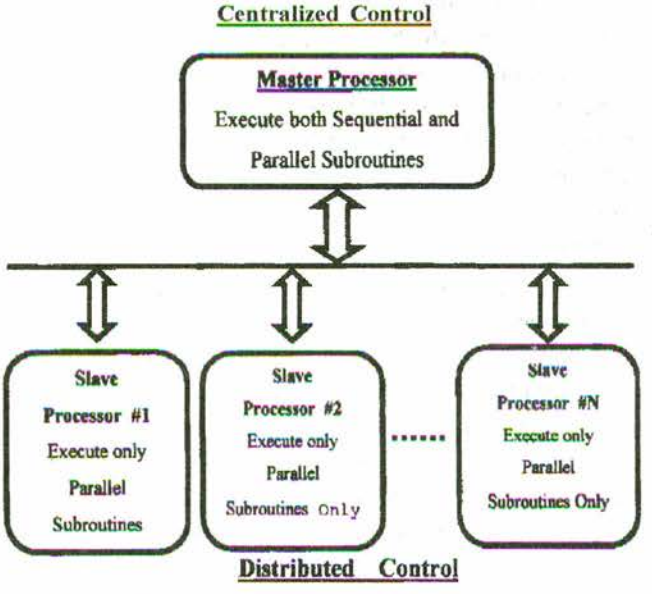

Fig 1. Block diagram of the proposed software model

Kernel, in general, is responsible for supporting concurrency and task allocation. Efficient routines are therefore required for: testing software flags; entering into and obtaining process information from the ready to run queue; and searching and updating the table entries (Burns 2000; ChiShiin *et al* 1999; Manimaran 1998). The kernel of the proposed software model, for each target processor in the system, logically contains two processes: the data structure and the process manager. The data structure holds the information needed to pass data from the process to the kernel, and other information about the subroutines while the process manager comprise of: the link between the process and the run time execution routines, and the kernel body.

Process link to Run-Time Executive. In the user program, the compiler literally declares the call for kernel as an external assembler subroutine. This subroutine, contains the following kernel entry protocols:

- Disable interrupts
- Save the return address, i.e., Save the stack pointer of the calling process
- Save the processor register state
- Load the kernel stack pointer
- The processor register state as well as the return address is required to be saved before actually executing the subroutine, because, the kernel process is executed on the calling processor. In addition, it is necessary to store the process state in the local stack of the calling process, so that it is accessible to the processor executing the kernel.

The kernel release protocol includes:

- Release the common bus; includes a call to the arbiter and the hardware semaphore, using the TAS flag. This implies a Reset of the semaphore flag
- Restore processor state
- Restore the return address, i.e., retrieve the process Stack Pointer
- Enable interrupts
- Execute standard return from subroutine (RTS).

Kernel body. The following tasks are performed by the kernel body routine:

- Request for indivisible bus access, if the bus is busy, wait for four NOPs (No Operations) cycles and reqyest again
- If the request is granted, then execute the Process Manager
- Release the bus for other processors
- Run the thread until completion

The kernel body functions includes: the indivisible request for the critical region; on grant, perform kernel entry and process management by updating the table entries according to the order of the execution of the subroutines. This is required for subroutine execution and for passing parameters to the 'C' function. The final job of the kernel body is to execute the subroutine from the pointer address until completion.

Indivisible request for common bus. For process synchronization, two routines the VMEbus request and release are implemented, as presented (Dijkstra 1986; Beim 1984). The Pseudo code for the protocol for requesting and releasing of the VMEbus follows:

```
Request_VMEbus () {
        do {
                if (sem_flag == 0)
                        sem_flag = 1;

                else if (sem_flag == 1) {
                        sem_flag =1;
                        Busy_wait();
                }
        } while (sem_flag == 0);
}

Release_VMEbus() {
        sem_flag =0
}
```

Kernel process manager. The Kernel Process Manager perform the following jobs:

- Initialize the pointers for the processors. One pointer is required for each of the processors 2 to N, connected to the shared memory. "N" is the maximum number of processors connected to the MPS
- Signals other processors to transfer program code to/ from the shared memory. The supervisor processor transfers the code to a shared memory segment. Other processors transfers this code from the shared memory to their local memories
- Signals other processors to start their jobs. This is required for the supervisor processor only
- Searches the address of the current_task_pointer
- Fetches the pointer to the subroutine, pointed by the current task pointer in the RTR queue
- Updates the contents of current_task_pointer, to point to the number of subroutines in the RTR queue
- If the subroutine has a value parameter to be passed, then it performs the communication between the kernel and the subroutine, using the parameter passing technique. This involves the transfer of the contents of the parameters, using the data structure

Parameter passing via shared memory. The inter processor communication provides the means of passing the

parameters between the kernel and the high level language (Dijkstra 1986; Lister 1981). Inter processor communication, in the proposed design, is achieved by either shared variables or by passing the value parameters between the process and the kernel via shared memory. The following steps are performed during parameter passing between the kernel and the process:

- Fetch the pointer to the current task pointer
- Fetch the address of the consecutive long word. This gives the number of parameters in that function
- Add an offset to this address giving the address of the first parameter
- Copy this address to the data structure Par-pointer
- Update the pointers

The lowest layer in the multi-layered diagram is the RTR queue of processes, which is shared by all the processors involved in parallel processing. Every processor accesses the RTR queue on a FIFO basis by an indivisible bus request for the shared bus.

Two Run time routines have been written for the designed MPS, one for the supervisor processor and the other for the processors 2 to N. During compilation, Kernel_No._1( ), is called and linked with the user's application program, and is loaded along with the user program code, into the RAM of one of the processor connected in the system. This processor acts as the supervisor. The second, Kernel_No._2( ), needs to be loaded into the local memory of all the other SBCÆs connected to the MPS, prior to the program execution. Figures. 2 and 3 show the control flow diagrams for the two run time routines. In order to visualize the software allocation to the system hardware, Fig 4. shows the physical locations of the software in the hardware.

Proposed compiler design. The analysis of a typical high level language program running in a Uniprocessor reveals the fact that, the distribution of the references to the code, local data and the global data, with some variation in different languages, is as follows (Jones 1980):

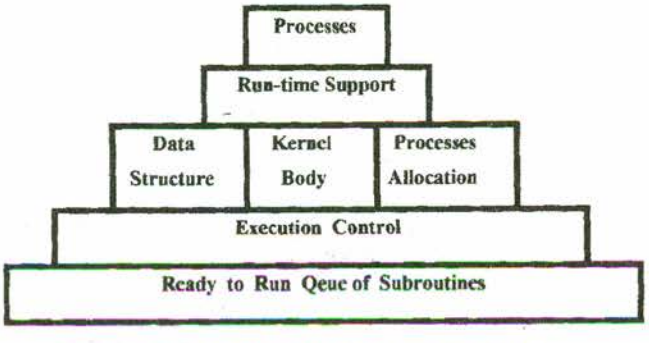

Fig 2. Logical representation of the software model

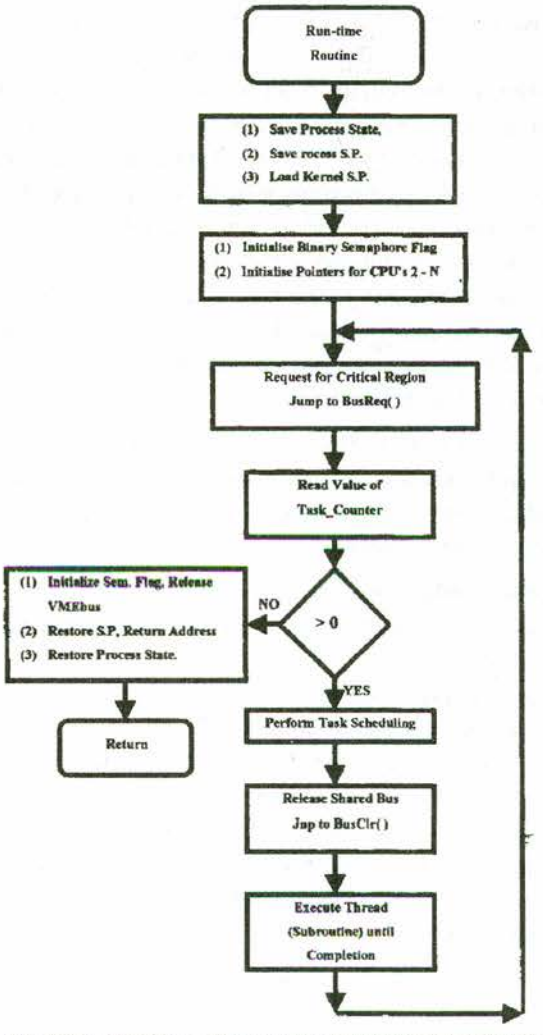

Fig 3. Control flow of the run-time executive routine for supervisor processor CPU # 1.

Code References	50%
Local Data References	40%
Global Data References	10%

Since, the global references are only 10 % of the whole program, only global data is placed in the shared memory of the designed architecture, to avoid the bottleneck. However, the large local memory in each processor board is used to store the application program as well as the run time support software.

The compiler at compile time generates the code for the assignment statements, for each of the parallel functions in "C" and its parameters, if any. At run time these statements prepare the RTR queue and provide other information about the parameters in the shared memory. Figure. 5 shows the sequence of information, stored in the process descriptor, during user program execution. The top of the sequence contains the pointer to the number of parallel functions in C. The next

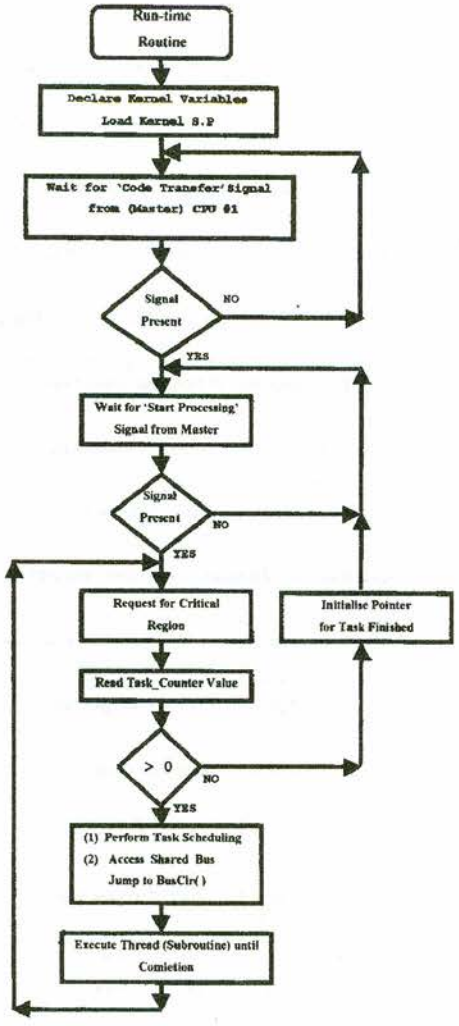

Fig 4. Control flow of the run-time executive routine for slave processors CPUs # 2 to # N.

item in the sequence is the pointer to the current function to be executed first. The next continuous locations in the sequence contain information about the number of parameters of the first function. Then, the pointers to each parameter are placed in sequence. This is repeated for each parallel function within a parallel block. For more than one parallel block, the same memory area is reused by over writing the new information for the next parallel block during execution.

The designed compiler, also includes the run time library routines, and generates the linker files, 'ifile' and 'makefile'. The linker file 'ifile' is also used for data allocation, for the local and shared memory and to link the application program with the system software. The UNIX system utility (Kernighan & Padua 1987), 'make', with the help of the linker files, compiles the output produced by the designed compiler, via the UNIX 'C' compiler. The object code is then down loaded into the local memory of the supervisor, for parallel execution.

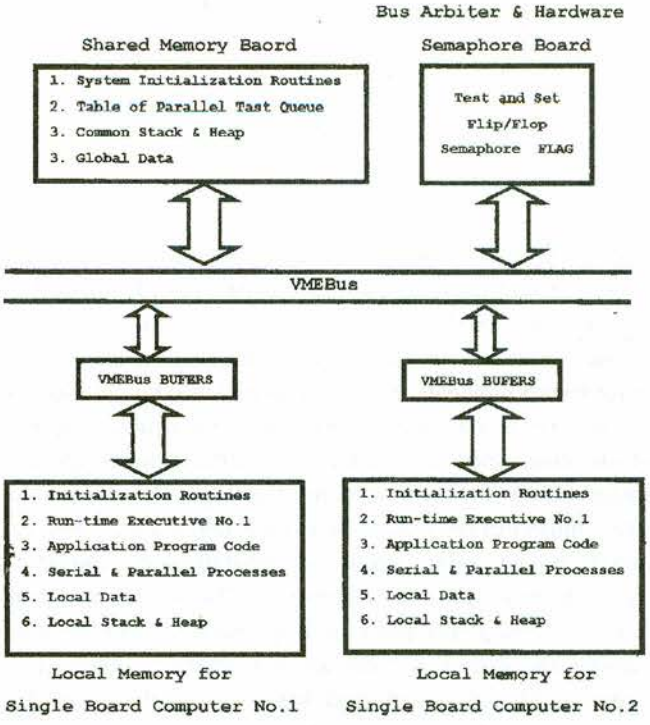

Fig 5. Location of software in the hardware configuration

Results and Discussion

The performance evaluation of the multiprocessor system is quite complicated, because, the same system may perform in a different way for different applications. The possibilities for parallelism depend upon the problems, as some applications may not be parallelised, while others may be partly converted into parallel portions. If the time taken to execute a program in a Uniprocessor is T_1, and that for N number of processors is T_N, then the speed factor may logically be defined as:

$$\text{Speed Factor} = T_N / T_1$$

This is the ideal case, however practically, there is always some overhead present due to the process synchronisation, task scheduling, processor communication and dependencies.

Digital Image processing for Solid Object Recognition (Hu 1962; Prewitt 1970; Jain 1989; Borghesi *et al* 1984), has been implemented to the designed Dual Processor Shared Memory System. The application software for object recognition, comprises different algorithms for the digital image smoothing and edge detection (Niblak 1987; Gonzalez 1987; Jain 1989; Rosenfeld 1986). Sigma Filter for noise smoothing (Niblak 1987; Lev *et al* 1977) and Sobel Operatorö for edge detection (Pavlidis 1983; Robinson 1976) have been adopted, because of better image quality and least computer time required. Hu invariant technique (Hu 1962; Bassam *et al* 1986) was adopted to compute the invariant characteristics of the object. Finally, the database is generated for different models during the learn-

ing stage, and used later during the recognition stage. The application software is written in language, "C". The main tasks were partitioned into subroutines, on divide by two basis, in order to allow execution in each of the connected processors, simultaneously. Different blocks of parallel subroutines are created, each having at least two subroutines for parallel execution.

During the partitioning of the object recognition software, nearly 65% of the entire software is partitioned, because of the data and precedence dependency. The two most time consuming processes: image preprocessing; and the calculation of invariants moments, are equally partitioned for parallel execution. The rest of the application program includes: capture of the image; cleaning the memory segments for temporary image storage before processing; tracing of the object contour; transfer of the images between frame grabber memory and the system memory (local and shared memory); and printing of the messages onto the terminal. Due to the sequential nature of these algorithms requiring complete image for processing and data & precedence dependencies, these processes could not be partitioned, hence, executed sequentially by the supervisor processor only.

In the proposed system, the timing measurement, for different processes are done by taking the average of twenty runs of the individual processes. The performance evaluation, for a single processor and, eventually, for two processors, of the complete object recognition software shows that, the overall 'speedup' achieved for two processors is around 1.53, with around 65% parallelism, as shown in Table 1. It is obvious that the speedup factor achieved for Pre-processing is 1.92, while for the calculation of invariant moments it is 1.88. This is because the global data accesses during preprocessing is minimised by transferring the image into the local memories of the processor boards. However, the calculation of invariant moments involve in the manipulation of huge global data. The 'speedup factor', achieved in both cases, is in agreement with that presented by Chandra [Chandra et al 1994], and is much higher than the "upper bound limit", given in Hwang & Briggs. According to which, the performance of a shared memory multiprocessors with 'N' number of processors, is expressed as:

$$\text{Speedup Factor} = N / \ln N$$

This factor using the above formula, for 2, 4 and 8 processors, is calculated to be 1.33, 1.92 and 3.08 respectively.

The speedup factor, achieved in the proposed system, is also better than that suggested by Jones, according to which, for a 99.9% parallel program, the speedup factor achievable for 2, 3 and 4 processors is 1.80, 2.80 and 3.30, respectively.

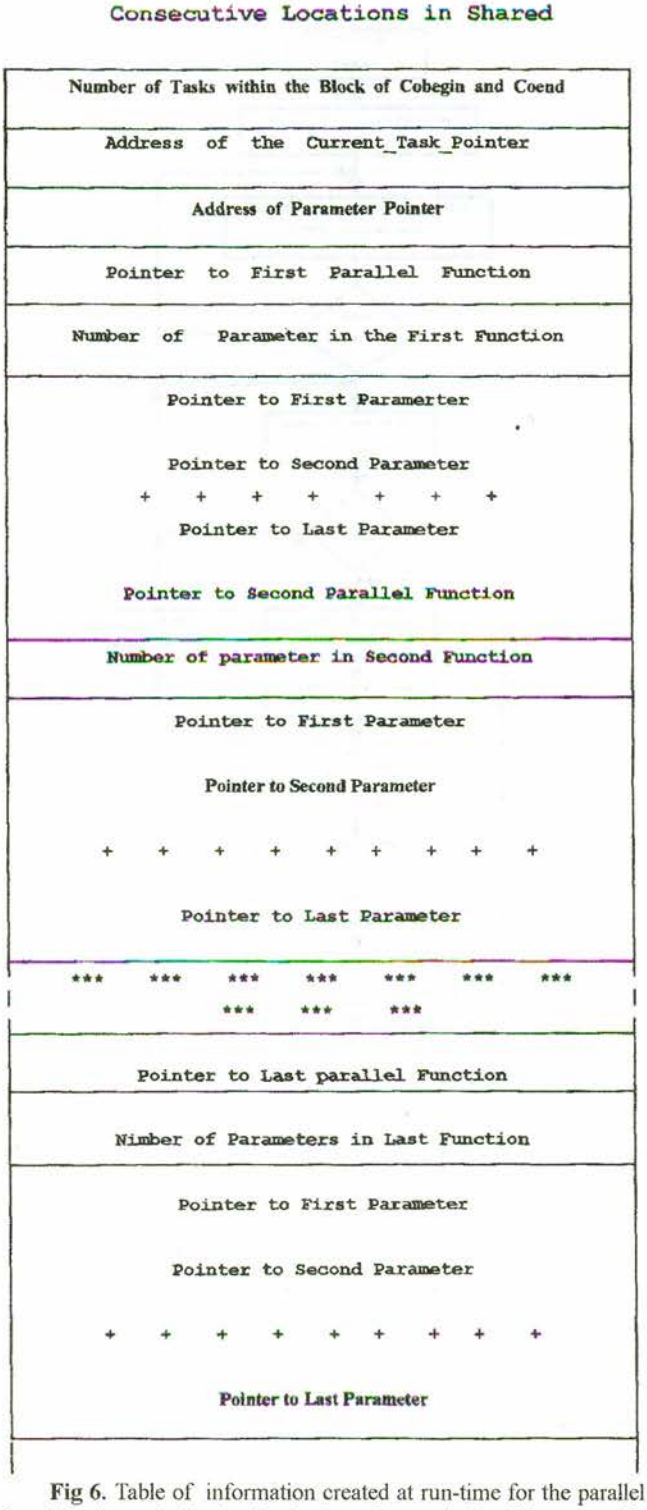

Fig 6. Table of information created at run-time for the parallel functions in each block of cobegin and coend (Parallel executable functions)

Conclusion

The design of the system software and a compiler for a single bus shared memory multiprocessor system, whose function

Table 1
Performance analysis of the Shared Memory Multiprocessor System for the object recognition software for single and dual processors.

S.No.	Object recognition processes	Single processor	Dual processor	Sequential parallel
1.	Digitisation and transfer of image to the Shared Memory	1.832	1.832	Sequetial
2.	Non-linear smoothing by Sigma filter	8.834	3.559	Parallel
3.	Edge detection by Sobel operator	3.324	1.713	Parallel
4.	Statistical smoothing by Borghesy	1.525	0.798	Parallel
5.	Object contour tracing by Pavlidas	4.059	4.049	Sequential
6.	Enhanced contour smoothing	1.383	0.712	Parallel
7.	1st and 2nd Order central moments	0.729	0.389	Parallel
8.	Higher Order (Order 4) Central Moments	6.338	3.353	Parallel
9.	Database search (for three objects in database)	1.681	0.889	Paralle
10.	Miscellaneous processes including eleaning of the image from memory; transfer of image to/from the frame Grabber memory and printing the Messages onto the monitor Screen etc.	2.027	2.027	Sequential
	Total time taken for all the processes	29.740	19.330	
	The Speed-up Factor is calculated to be	153		
	Consecutive Locations in Shared			
	Time taken for	13.66	6.782	
	The Speed-up factor is calculated to be	1.92		
	Time taken for the Calculation of Central/invariant	6.746	4.631	
	Moments Process Nos. 7,8, and 9			
	The Speed-up factor is calculated to be	1.88		

is to provide a subroutine oriented environment for parallel programming, is presented. The aim is to provide an upgrade path for the user to add processor boards for more computer power in a real time environment, without modifying or recompiling the system or the application software.

The user is able to program in the high level language C, and there is no imposition of a new programming style or new syntax. The only exception being that the user is required to enclose the parallel functions of the C programming language in between the keywords cobegin and coend. The processor synchronization, scheduling and data allocation is transparent to the user. The global variables are allocated as shared variable by the compiler generated linker file 'ifile'. The output produced by the designed compiler is again compiled by the NIX C compiler to produce object code for execution in the designed system. Currently the hardware comprises two identical SBCs, but the architecture is not restricted to two SBCs and any number of identical SBCs can be attached, in a suitable hardware environment. However, not more than six processor boards are recommended, because of the heavy load of data, the shared bus becomes the bottleneck.

References

Bassam B and DeFigueiredo R.J.P, 1986 "A General Moment Invariants/Attributed Graph Method For 3-D Object Recognition From a Single Image", IEEE Journal of Robotics and Automation", **RA 2,** No. 1, pp. 31-41.

Beims B, 1984 "Multiprocessing Capabilities of the MC 68020, 32-bit Microprocessor", Proc. of WESTCON, AR 217.

Borghesi P, 1982 Chappellini V and Bimbo A.D, "A Reliable Technique for Recognition of Moving Objects", IEEE Proc. on Information Theory, Les Arcs France.

Borghesi P, Cappellini V, Caria R, Bimbo A.D, Mecocci A and Pareshi M.T, 1984 *"Digital Image Processing Techniques*

for Object Recognition and Experimental Results", Digital Signal Processing, Elsevier Science Publishers B.V. North Holland.

Burns A, Prasad D, Bondavalli A, Giandomenico F. Di, Ramamritham K, Stankovic J.A, and Strigini L 2000, The Meaning and Role of Value in Scheduling Flexible Real-time *Systems Journal of Systems Architecture,* **46** pp. 305-325.

Chandra R, Gupta A and Hennessy J.L , "COOL: An Object Based Language for Parallel Programming", IEEE Computer Society, Aug 1994, pp.13-26.

Chia Shen, Gonzalez O, Ramamritham K and Mizunuma I 1999: User Level Scheduling of communicating Real-Time Tasks in Proceedings of the Fifth IEEE Real-Time Technology puter, **18,** (8). 97-112.

Deitel H and Klappholz D, "Refined C 1985: A Sequential Language For Parallel Programming, Proc. of Intl. Conf., on Parallel Processing, pp. 442 449.

Dijsktra E.W, 1986 *"Cooperating Sequential Processes"*, Programming Languages, F.Geunys Academic Press, NY pp. 43 112.

Fohler G and Ramamritham K 1997, Static Scheduling of Pipelined Periodic Tasks in Distributed Real-Time Systems, 9th EuroMicro Workshop on Real-Time Systems. pp. 128-135, Toledo, Spain

Gajski D, *et al,* 1985 "Essential Issues in Multiprocessors Systems", Computer, p. 9 27, C-4 (3).

Robinson G.S 1977, "Edge Detection by Compass Gradient Masks", Computer Graphics and Image Processing, **6**, pp. 402-501.

Gonzelez C.R, Paul Writz, 1987 "Digital Image Processing", Addition Wesley,.

Haritsa R, and Ramamritham K 2000, Real-Time Databases in the New Millenium Real-Time Systems, 19 (3) pp 205-208.

Hu M.K, 1962 "Visual Pattern Recognition by Moment Invariants", IRE Trans. Information Theory, pp. 179 187.

Hwang K and Briggs F 1984, *"Parallel Architectures and Systems"*, McGraw Hill, .

Jain A.K 1989, "Fundamentals of Digital Image Processing", Prentice Hall Inc.,

Jones A.K and Schwatrz P 1990, "Experience Using Multiprocessor Systems: A Status Report", Computing Survey, Vol. 12, No. 2, pp. 121165.

Kernighan B.W and Padua, 1987 The UNIX Programming EnvironmentÆ, Prentice Hall Software Series, Englewood Cliffs, New Jersey 07632.

Kernighan and Richie, 1989 *C Programming Language* Prentice Hall Inc., pp 55-70.

Kirrman H.D and Kanilimann F 1984, Poolpo A Pool of Processors for Process Control Applicationsö, IEE Trans., on Computers, Vol. C-33, No.10, pp. 869-879.

Lister A.M 1983. "Principles of Operating Systems", Prentice Hall Inc.,

Lev A, Zucker S.W and Rosenfeld A 1998, "Interactive Enhancement of Noisy Images", IEEE Trans. Systems, Man and Cybernetics, C-14, pp 22-32.

Manimaran G, Murthy S.R and Ramamritham K 1998, A New Algorithm for Dynamic Scheduling of Parallelizable Tasks in Real-Time Multiprocessor Systems, Real-Time Systems Journal **15,** 39-60.

Midkiff S.R, Padua D.A, and Cytron R, "Compiling Programs With User Parallelism", Research Monographs in Parallel and Distributed Computing, David Gelernter and David Padua (Editors), The MIT Press, Cambridge, Massachusetts, 1990.

Niblack W 1986, *"An Introduction to Digital Image Processing"*, Hamel Hempstead, Prentice Hall,.

Pavlidis P 1977, *"Structural Pattern Recognition 2"*, Springer-Verlag.

Prewitt J.M.S 1970, *"Object Enhancement and Extraction", Picture Processing and Psychopictories,* B. S.(Lipkin and A. Rosenfeld, Eds., Academic Press, N.Y., pp. 75 85.

Riganati J.P and Schneck P.B 1984, "Supercomputing", Computer, 18 (8)97-112.

Robinson G S 1977, "Edge Detection by Compass Gradient Masks", Computer Graphics and Image Processing, 6, pp. 402-501.

Roning J 1983, "An Image Analysis System for Industrial Application", Proc. Third Scandinavian Conf. on Image Analysis; Copenhagen,

Rosenfeld A 1986, "Introduction to Machine Vision", IEEE Control Systems and Management, **5,** (3) 14-17.

Schwederek T and Seigel H. G 1986, "Adaptable Software For Supercomputers", IEEE Computer, **19,** (2) 40-57.

Sega Holzner 1989 C with Assembly Language BPB Publication, Printed in India with Brady Books, a division of Simon & Schuster Inc.,

Stankovic J, Spuri M, Ramamritham Kritha and Buttazzo G C 1998, Deadline Scheduling for Real-time systems EDF and Related Algorithms, Kluwer Academic Publishers,

Tabak D 1990, "Multiprocessors", Prentice Hall Inc., Englewood Cliff, NJ,

Xiong M and Ramamritham K 1999, Deriving Deadlines and periods for Update Transactions in Real-Time Databases, 20th IEEE Real-Time Systems Symposium pp. 32-43.

Yaqub Mohammed 1991, "A Shared Memory Multiprocessor System and the Parallel Compiler", Ph. D Thesis, University of Bradford,

Yaqub Mohammed, Deka R and Singh B, "On the Hardware Description of a Stand alone System Incorporating Fixed and Floating Point Coprocessor Board", Poc. of Mediterranean Electro-technical Conf., Melecon-91, Ljubljana, Slovenia, Yugoslavia, pp. 973-976.