A HIGHER ORDER PARALLEL ABSTRACTION FOR FIXED DEGREE DIVIDE AND CONQUER PROBLEM SOLVING STRATEGY

T.S. Mohan and V.S. Ganesan,
Department of Computer Science and Automation,
Indian Institute of Science,
Bangalore 560 012

Abstract

In this paper, we present an algorithmic framework for the *Fixed Degree Divide & Conquer*(FDDC) problem solving strategy, that adapts the divide and conquer algorithm for parallel computations. We capture the notion of a thread of execution in an active *task* object which are instances of a data type called the **tasktype** and propose a higher order tasking abstraction called **maptask** that takes as its arguments, a tasktype, the (fixed) degree of the runtime task hierarchy and the arguments for the tasks. We have implemented the above abstraction by extending C with **task_type** and **maptask** in a package called the **CT Kernel** that currently executes tasks in a distributed environment consisting of 10 networked VAXstations. Using the above abstractions we have implemented the FDDC algorithms for mergesort, FFT, Cubic spline based curve fitting, Fractals, Matrix multiplication, as well as Iterative relaxation.

1 Introduction

Of the many numerical algorithms, there are algorithms for a class of problems that use the *Divide & Conquer* problem solving strategy. Typically, in such a strategy, the problem is decomposed into identical subproblems and the algorithm is applied recursively on each subproblem until further decomposition is not possible. Then the solutions of the subproblems are merged back appropriately to get the final solution of the problem. In a sequential programming language, recursion succinctly supports this paradigm though iterative versions of it also exist. In a parallel environment, subject to data dependencies, each of the subproblem can be executed in parallel with others and hence it may be taken as a separate thread of execution (which may execute on a different processor). Thus, a hierarchy of interacting threads can be triggered to solve the problem using FDDC.

There are many distributed programming languages that provide support for threads (Ex: Ada [4], Concurrent C [8] and C Threads package for the Mach Operating System[9]). These are general in approach with no specific support for FDDC problem solving. Very few parallel abstractions exist which support the above problem solving strategy even in an restricted form. However attempts have been made to use the generalized framework in conventional languages[6]. We believe that this is the one of the few attempts in that direction.

In this paper, we briefly explain the FDDC algorithm in section II, give the definition of the **task_type** abstraction in section III, and dwell on the FDDC tasking abstraction called **maptask** along with an example in section IV. We give some implementation details in section V, We discuss the results of the implementation and conclude in section VI.

2 The FDDC Strategy

The *Fixed Degree Divide & Conquer* Problem solving Strategy is a constrained form of the divide and conquer algorithm in the sense that the number of child nodes of the process tree is fixed at every level of the process tree. This methodology has a sequential specification and yet is amenable for efficient parallel implementation. Again, the constant number of child nodes at every level helps in more efficient mapping

*The Authors thank Prof.V.Rajaraman for the encouragement and support.

of the tasks on the underlying parallel architecture. Typical problems that fall in this class include the mergesort which has a process tree where every intermediate node having a degree of 2 throughout. So is the case with FFT. In case of algorithms using the *quad-tree* data structures, every intermediate node may have a degree of four.

A parallel implementation of the FDDC strategy can be concisely expressed as follows (in pseudo functional code):

```
(defun fddc(atomic? split join degree func input)
  (if (atomic? input)
      (apply func input)
    (join degree
      (map fddc atomic? split join degree func
        (split degree input)))))
```

The nature of the arguments of the function **fddc** or **fd_divide_and_conquer** as well as the constituent functions are as follows:

atomic? It is a predicate that returns true if the input cannot be subdivided further. The subdivided input is used to solve each of the subproblem to get the subsolution. This fixes the granularity of the task.

split This function splits the given input into **degree** parts so that each part can be given to each recursive invocation of the divide_and_conquer function.

join This function collects all the subsolutions returned by the algorithm when applied to each of the subproblem at the same level of the process tree.

degree This parameter specifies the degree of a node in a process tree. Since all intermediate nodes of the process tree have the same degree, we call this algorithm a fixed degree divide and conquer algorithm.

func This is the user defined function that is specific to the problem at hand.

input This gives the array (or list) of data structures that forms the arguments of the user defined function.

apply This is a higher order function that takes a function definition and applies it to the arguments presented.

map This is also a higher order function that takes in a function definition and a set of argument list. It applies the function definition to the corresponding elements of the argument lists. This is similar to the mapping function found in dialects of Lisp.

We capture the above abstraction in the form of **maptask** that takes in a **task_type** and appropriate arguments.

3 Tasks & tasktype

A task type is an abstract datatype which encapsulates a set of data objects and defines operations on them. These operations form the sequential body of an instance of the tasktype activation: called the Task. Some of the operations within the body of the task allow the pertinent task to communicate with other tasks either synchronously or asynchronously. The communication interface of a task is uniform over the multiple processors. A tasktype is different from the Ada's tasktype but has some implementation similarities[4]. Objects of a tasktype are declared like any other datatype.

A task is explicitly created using the tasktype definition and the **maptask** primitive. On creation and activation, a task executes the sequence of statements in the body of the tasktype definition sequentially. The parent task now executes in parallel with all the child tasks that it has spawned. Tasks communicate with their parent tasks only either synchronously or asynchronously using the **hear** or **audit** statement. On completion of the statement execution, the task terminates automatically following well defined termination semantics. There can be multiple instances of a tasktype active at any time. Since each task is an independently executable unit, there cannot be global (shared) variables between tasks. All function and procedure invocations from the body of the task should have all necessary parameters passed to it explicitly. However the reentrant code for the procedures, functions and constant declarations are sharable between tasks. The tasktype cannot be nested. The C's type system has been extended with the 'tasktype'. The syntax of the definition of the **tasktype** is given in the Figure 1. Messages are essentially from the child tasks to the parent task for returning the results. All the messages are different C datatypes and are passed by copying it between the tasks. Hence, pointers have no meaning be it either directly or within structures. More details about an implementation variant is given in [7].

```
task_type_declaration ::=
    TASK_TYPE task_type_name (arguments_structure)
    argument_structure_type_declaration
    {
        Normal_C_local_function_declarations
        ...
        Normal_C_Statements
        | Inter_Task_Communication_statements
        ...
        ...
        END_TASK_TYPE
    }

Inter_Task_Communication_statements ::=
    Tell(Message, Message Length)
|   Talk(Message, Message Length)
Message ::=
    any_C_data_type
```

Figure 1: The EBNF for the **tasktype**

4 The maptask Abstraction

The **maptask** is a higher order abstraction which generates a full task tree at runtime, schedules the tasks to the processors appropriately and collects the results and returns it to the invoker. Issues such as generating the appropriate tasks, mapping them onto the appropriate processors, collecting the results sent by each task and synchronising the threads of execution on the termination of the tasks in taken care of by this abstraction. The results are collected by the maptask either synchronously (if the user defined task uses synchronous communication statements to communicate results) or asynchronously. The results are placed in the appropriate slots of the result variable corresponding the task communicating. Of course, appropriate information has to be given to construct and it is not flexible for all variants of the divide and conquer algorithm.

The general form of the **maptask** is as follows:

```
maptask(<user_defined_tasktype>,
        <Communication_Mode>
        <No_of_tasks_to_be_generated>,
        <input argument length>
        <user_defined argument_copying function>
        <arguments_for_argument_copying_function>
        <results_length>,<result argument>)
```

The input **tasktype** to the maptask is one of the tasktype definitions as given above. The user defined argument copying function is a function that takes as its first argument the index of the task to be spawned and taking the data from teh consolidated argument input, picks the necessary data which forms the argument to the corresponding task. The argument copying function is a convenient user defined function that makes for efficiency and returns a pointer to a copy to the pertinent part of the input. When a task gets spawned, it is put into the ready queue of the local scheduler (in a parallel environment). When such a task gets activated, it accesses the passed parameters through this pointer. Note that there is one level of copying of the argument of any type (and this may be across processors). Again, a **maptask** spawns tasks one level deep. By recursive application of it, we generate the whole task tree. Figure 2 gives the code for the core part of the mergesort program where all these features are explicit. In this program, the first element of the array to be sorted contains the length of the array.

The **maptask** generic abstraction substitutes for many lines of code typically written to achieve the same end which is common to the pertinent class of problems. The domain specific aspects of the code are taken care of single user defined tasktype. The depth of the runtime task tree is controlled by the user code within the tasktype definition by the appropriate recursive usage of maptask.

5 The Implementation

We have implemented the tasktype abstraction by augmenting C with **tasktype** data type. The generalized form of **maptask** is currently under implementation. The runtime libraries of this implementation, called the **CT Kernel**[1] is currently available on a network of VAXstation 2000 workstations running Ultrix (a variant of Unix). We have used internet protocols for interprocessor communication. It is important to note that a Unix process is different from our task (which is a lightweight thread). We use only one Unix process per processor to support tasks which are orders of magnitude higher. The most generalized form of the algorithm used for implementing the **maptask** is as follows:

- *Initialize and obtain the task assignment to the various processors dynamically depending on the load.*
- *Generate the tasks on the corresponding processors.*
- *Await the results from every child task and put it in the corresponding slot of the results area.*
- *Return the results.*

```
mscopy(task_no, inptr, outptr)
int task_no;
int *inptr, *outptr;
{ int len  = *inptr;

  *outptr = len / 2;
  if (task_no == 0)
    bcopy((char *)(inptr + 1),
          (char *)(outptr + 1),
          (len/2)* sizeof(int));
  else
    bcopy((char *)(inptr + 1 + (len/2)),
          (char *)(outptr + 1),
          (len/2)* sizeof(int));
}

/* The mergesort tasktype definition */
TASK_TYPE ms(array)
int *array;
{
  int len, tmp;

  len = *array;
  if (len == 2)
   { /* Swap the items if necessary */
     if (*(array + 1) > *(array + 2))
       { tmp = *(array + 1);
         *(array + 1) = *(array + 2);
         *(array + 2) = tmp;
       } /* Else it is in sorted form */
   }
  else
   { *array = len;
     maptask(ms,SYNC_MODE,2,
             ((len/2) + 1)*sizeof(int),
             mscopy,array,len * sizeof(int),
             (array + 1));
      merge(array);
   }
  tell((arr  + 1),len);
  END_TASK_TYPE
}
```

Figure 2: The Merge sort tasktype definition along with the argument copying function mscopy

The CT kernel package is a combination of a preprocessor and a runtime system. The preprocessor transforms the **maptask** and **tasktype** code into conventional C prorgam with appropriate calls to the runtime support libraries. The runtime libraries support a good number of useful primitives which augment the capability of the **maptask**. The task context switching has been implemented using coroutines and in assembly. The average task context switch time is around 40 microseconds. Currently the CT kernel is being used as a testbed for trying out algorithms in dynamic load balancing, task partitioning and scheduling strategies as well as in designing heuristics for near optimal resource allocation.

6 Conclusion

We discuss the results of coding six simple well known algorithms which are amenable to efficient solution using the above abstraction. Given the fact that we are using a network based interconnection of workstations, the speedups got for each of the problem mentioned below was between 2 and 4 for a 10 processor setup. The effect of other packages running on the Unix was visible in the form of interprocessor communication delays. In brief, the algorithms coded are as follows:

Mergesort This is the classic example of FDDC with degree two. The input array is partitioned into two with maptask generating tasks which operate on it. Some part of the code is given above (Fig 2). We terminate the recursion when the input is of length two. Hence for a 128 sized array, we have 128 tasks.

FFT Given an array of time domain samples, we compute the frequency domain results using the above strategy. We find that the degree of runtime task tree intermediate node is two. Hence the input is partitioned into two and two **fft** tasks are spawned by the maptask for every level of the fft. All such tasks execute in parallel over the distributed environment. For a 32 input array, 63 tasks get generated.

Cubic Spline . Given the set of points, we compute the coordinates of the smoothly fitting curve using cubic spline interpolation algorithm. Initially, the second derivative of the curve at all the given points are computed. Next using this information, the interpolated points of the different parts of the curve are computed in parallel. The application of maptask here generates a task

tree of depth one. For 16 points, it generates 16 tasks. We have a display task that displays the curve using X window primitives.

Fractals We compute the contours of the mandelbrot diagram (Ex: Snow Flakes) by parametrizing the template to be applied recursively to the applicable sides of the generated figure. The depth to which this application extends (i.e., the depth of the runtime task tree) is controlled by the user. This is a case of pure recursion based divide and conquer algorithm with tasks running on different processors generating the final display coordinates of the fractal. With 4 generators and depth 5, 1024 tasks get generated. Here again, we used a display task to display the fractal computed using X Window primitives.

Matrix Multiplication This is a simple example which works with a runtime task tree of depth one. The rows and columns of the input matrices are appropriately partitioned and given to a task which computes the element of the resultant matrix. This has high communication overheads. An optimized & constrained version makes each task compute a submatrix of the resultant matrix. Thus, multiplication of two 36×36 matrices generated 36 tasks for computing the final result.

Relaxation Algorithm This is an iterative algorithm where at most the runtime task tree is of depth one. Tasks are spawned to compute the odd-even ordering with *Chebyshev acceleration*. We used a boundary value problem represented as a 12×12 matrix. Every iteration generates a 12 tasks which are given appropriate rows as input. The number of iterations is determined by the error function which checks for convergence and terminates when the values of the preceding and current matrices are within a fixed limit. Our problem took 47 iterations and generated 564 tasks.

Overall, the communication overheads hampered the extraction of parallelism and current efforts are on to make the implementation more efficient.

References

[1] T.S.Mohan *CT - A runtime support system for C with Tasktype* KBCS Tech. Report, KBCS Group, Supercomputer Education and research Centre, Indian Institute of Science, Bangalore, 1990.

[2] Micheal J Quinn, *Designing Efficient Algorithm for Parallel Computers* McGraw-Hill Co. pp 141-146, 1987.

[3] Hendri J. Nussbaumn,*Fast Fourier Transforms and Convolution Algorithms* Springer Series in Information Sciences - 2, pp 85-90, 1981.

[4] P.Krishnan, R.A.Voltz, R.J.Theriault *Implementation of Tasktypes in Distributed Ada* ACM Ada Letters, Vol:VIII, No:7, 1988 Spl.Edition.

[5] B.B.Mandelbrot, *The Fractal Geometry of Nature* W.H.Freeman Co, 1983.

[6] Murray Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation* Research Monographs in Parallel & Distributed Computing, Pitman Pub.,1989.

[7] T.S.Mohan *A Tasking Abstraction for Message Passing Architectures*, Proc. PARCOM - 90 Conference, Pune, Dec, 1990.

[8] R.F.Cmelik, N.H.Gehani, W.D.Roome, *Experience with Multiple Processor Versions of Concurrent C* IEEE Trans. Software Engineering, Vol:15,No:3, March 1989.

[9] E.C.Cooper, R.P.Draves, *C Threads* CMU Tech Report CMU-CS-88-154, June 1988.