Mathematical Modelling and Analysis
Volume 7 Number 1, 2002, pages 11–20
© 2002 Technika

THE TEMPLATE PROGRAMMING OF PARALLEL ALGORITHMS

M. BARAVYKAITĖ[1] and R. ŠABLINSKAS[2]

[1] *Vilnius Gediminas Technical University*
Saulėtekio al. 11, 2040 Vilnius, Lithuania
[2] *Omnitel*
Vytenio 18, Vilnius, Lithuania
E-mail: `Milda.Baravykaite@fmst.vtu.lt, ramas@vdu.lt`

Received October 1, 2001

ABSTRACT

The parallel programming tools and packages are evolving rapidly. However the complexity of parallel thinking does not allow to implement many algorithms for the end user. In most cases only expert programmers risk to involve in parallel programming and program debugging.

In this paper we extend the ideas from [3] of template programming for a certain class of problems which could be solved by using general master-slave paradigm. The template is suitable for solution of the coarse grain and middle grain granularity problem set. Actually, it could be applied to solve any problem P, which is decomposable into a set of tasks $P = \cup_{i=0}^{N} t_i$. The most effective application cases are obtained for the problems where all t_i are independent.

The template programming sets some requirements for the sequential version of the user program:

1. The main program must comprise of several code blocks: data initialization, computation of one task t_i and the processing of the result.

2. The user has to define the data structures: initial data, one task data, the result data. These requirements do not require to rewrite the existing sequential code but to organize it into some logical parts. After these requirements (and naming conventions) are fulfilled, the parallel version of the code is obtained automatically by compiling and linking the code with the Master-Slave Template library.

In this paper we introduce the idea of the template programming and describe the layer structure of the Master-Slave Template library. We show how the user has to adjust the sequential code to obtain a valid parallel version of the initial program. We also give examples of the prime number search problem and the Mandelbrot set calculation problem.

1. INTRODUCTION

Unlike conventional programming, the template programming does not require from the user to know the parallel programming tools to create parallel programs. Instead the user has to recognize his/her program type and to choose the right template where some code pieces are inserted into a predefined place.

In our paper we describe the Master-Slave Template library. It is designed to work efficiently for the coarse grain granularity problems, i.e. the problems where computation time of a sub-problem is relatively large. Usually it is used on heterogeneous computer clusters to solve the problems which can not be solved by one computer in a moderate amount of time. Good examples would be: calculation a set of inverses of the large matrices, calculation of fluid dynamics problems in various conditions or with various fluids, a multi-dimensional function optimization. All the examples have a set of sub-problems, which could be solved in parallel. The Master-Slave Template library should not be used to solve an algorithm-specific problems like single matrix inverse computation or implementation of a finite element schemes, because the sequential algorithms of these problems have pretty complex parallel counterparts, which do not obey the simplified behavior scheme of the master-slave algorithm.

The Master-Slave Template library is ready-to-use for the C language programmers, although the FORTRAN interface could be derived easily by any user due to the code simplicity. The PVM or MPI packages are used as the underlying message passing engines. The user is not required to know the parallel programming techniques. On the other hand, PVM or MPI packages should be installed correctly on the workstation cluster and the user must have a general knowledge of them.

The Master-Slave Template library should be used by the user only when the sequential version of the program has been tested and debugged. The parallel version is obtained in one step [1]: the user must make adjustments to his/her program to comply the requirements stated in the subsection 2.1. After the adjustments are made, the new sequential version of the program should be tested and debugged. The parallel version is obtained by linking the code to the Master-Slave Template library.

2. THE MASTER-SLAVE TEMPLATE LIBRARY IMPLEMENTATION

The main purpose of the template programming is to save the user from creation of the parallel algorithms from scratch. The template programming and other semi-automatic tools for parallel programming hide the fact that the user is doing a parallel programming at all. To achieve this, we organize

[1]The additional testing and debugging may be needed in case the user does not follow the recommendations described in section 2.1

the Master-Slave Template library into a set of independent layers. The figure Fig. 1 describes the layer structure of a program. Each layer consists of a set of procedures which either implement the layer logic, or serves as an interface that do the transformation of names and/or parameters. In this section we describe the layer construction in detail.

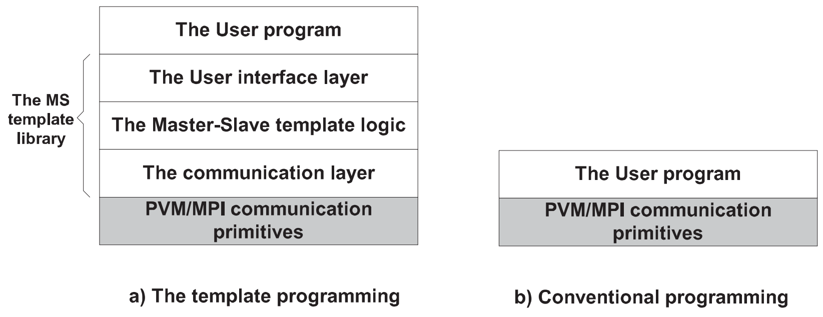
The User program
The User interface layer
The Master-Slave template logic
The communication layer
PVM/MPI communication primitives
The MS template library
The User program
PVM/MPI communication primitives
a) The template programming
b) Conventional programming

Figure 1. The layer structure of a program.

2.1. The User Program

We assume that the user already has implemented the sequential version of his/her algorithm. This sequential version will be the starting point for all the code reorganization needed to run the program in parallel. The reorganization step is the only action required from the user, so it is critical to conform all the requirements stated for this layer in this section to obtain a valid code for parallel version of the algorithm.

The program structure. We require that the user code comprise of several parts:

1. The data initialization part.
2. The computational process initialization part.
3. The computation loop which uses the following procedures:
   (a) Take a non-completed task t_i from the task pool,
   (b) Process the task t_i and return the result r_i,
   (c) Add up the result r_i to the totals.
4. The result output part.

We shortly describe each part.

The data initialization part consists of a procedure `M_prepare_job_pool` which may be invoked once during the program runtime. The procedure initializes and returns one argument which is an initial data structure created during the data initialization step (e.g. some parameters read from the initial data file).

The computational process initialization part consists of a procedure named `S_initialize` which is invoked for each computation process and is used to initialize the data structures needed for the computation process, i.e. allocate memory for temporary structures, assign initial values etc. The procedure has one input parameter which is an initial data structure prepared by the previous procedure (`M_prepare_job_pool`). We emphasize, that all the data needed for computational process initialization must be passed by initial data structure or generated inside the procedure (no global variables allowed to read the value from).

The procedure that takes a non-processed task t_i from the task pool is called `M_take_piece_from_pool`. It returns a structure of data which fully describes a task to be performed by the computation procedure. This procedure is called every time, when the data portion is needed, it returns 1 if it was successful to get a new task or returns 0 if the task list is empty. The task list could be either saved explicitly (like finite number of data records) or could be defined as a set of criteria (like accuracy achieved, iterations made, value combinations performed etc.).

The computation procedure named `S_compute` takes the task definition structure and performs computations over it. The output of this procedure is another structure, which contains the result data.

The procedure that processes the result `M_add_to_result` takes the result data structure from the output of the previous procedure and adds it up to the totals.

The output procedure `M_print_result` prints out the total results.

To summarize the above, the structure of the user's main program in C will look like:

```
int main()
{
  struct t_init_data id;
  struct t_data      t;
  struct t_result    res;

  M_prepare_job_pool(&id);
  if (! S_initialize(id)) return 0;
  while ( M_take_piece_from_pool(&t))
   {
     S_compute(t, &res);
     M_add_to_result(res);
   }
  M_print_result();
  return 1;
}
```

The data structures. We require that the user describe three data structures:

`t_init_data` – the initial data, prepared by `M_prepare_job_pool`;

`t_data` – the task definition data, which uniquely describes each task;.
`t_result` – the result data, that contains one task computation results.

The definition of these types should be separated into a header file named `i_defs.h`. No additional information is allowed in this file (such as global variables or other data types)

The global variables. We recommend that the programmer avoid the usage of the global variables. Just because the initial sequential code is split among different processes in the parallel version, it means that the global variables (if used) have their scope limited to the code part which belongs to one process. For an unexperienced user it is hard to tell whether a global variable will be visible for a particular piece of code or not, so the recommendation is do not use the global variables, but exchange them by the means of data structures, defined in the previous subsection. Of course it is not convenient in all cases.

Further in this section we describe the specifics of the usage of global variables in parallel version of the program. The parallel project is in fact many copies (program instances) of the same code. No matter the process (or an active instance of a program) is called *Master* or *Slave*, it has the code of all parts of the program, including global variables. Except that the slave process uses one part of the code, the master process uses another. We must also recall that although the two slave processes use the same part of the code, they can not share the values of the global variables.

To help the user to restrict the scope of global variables we recommend to split the initial sequential code into two parts or modules. One module should contain the procedures owned by the master process, the other - owned by the slave process. It is also recommended to separate the part of the code which is used by both processes (utility procedures). In our case the procedures owned by the master process are:

```
M_prepare_job_pool(&id);
M_take_piece_from pool(&t);
M_add_to_result(res);
M_print_result();
```

The procedures owned by the slave process are:

```
S_initialize(id);
S_compute(t,&res);
```

The global variable definition must be done in each module separately, no name duplicates allowed. It is also recommended do not use global variables in other modules if present. In cases some variables are needed by both processes master and slave, we have to decide either to exchange them with the initial or the task data structures (`t_init_data` or `t_data`), or to initialize/compute/maintain the variables separately in each process.

Testing. After the user adjusts the sequential code to conform the requirements, the testing is made by compiling and running of the program in the sequential mode. In case of success, the user simply switches to another directory and compiles/runs the parallel version of the same program. All the possible result discrepancies could appear due to global variable misuse, which could be debugged by inserting some variable print sentences in either of the module's procedure.

2.2. The User interface layer

The purpose of the user interface layer is to transparently connect the user layer procedures to the Master-Slave Template library. In this section we describe the structure and implementation of the user interface layer procedures.

The user interface layer consists of a parser and a set of interface procedures:

`u_master_prepare_job_pool`,
`u_master_take_piece_from_pool`,
`u_master_add_result_to_total`,
`u_master_print_result`,
`u_slave_initialize`,
`u_slave_compute`.

The interface procedures use the code generated by parser to transform the user data structures. They also call for the user layer procedures to perform the logic defined by the user. The user interface procedures are invoked by the Master-Slave Template logic layer procedures.

The parser analyzes the user data structures defined in the user layer in the file `i_defs.h` and prepares a source code for data transformation needed to pack and send data among processes. It recognizes the basic data types of the C language and multi-dimensional arrays of standard types. The current version of the parser does not support the structure in the structure types. The parser analyzes each variable in the user data structure and assigns it to one of the data vectors (of different data type) which are packed and sent from one process to another later on.

2.3. The Master-Slave Template logic layer

The main goals of this layer is to create the master and the slave processes on the parallel machine, to organize the work flow of the master-slave algorithm and to exchange the data among the processes. This layer consists of two procedures [3]: `Master`, `Slave`. These procedures invoke the user interface layer procedures to perform the logic of the program and the communication layer procedures to pack and exchange the data (see the figure Fig. 2).

The implementation of both procedures is rather simple: there is a loop of *actions*. Each *action* is either a set of instructions that perform module logic, or invocation of the user interface or communication layer procedures. There is a possibility to control the sequence of *actions*, so the Master-Slave behavior logic is easily adjustable to sophisticated needs. This layer also implements the fault tolerance features of the master-slave algorithm: the task list is stored and repeated task processing is forced in case of communication failure or other kind of the slave process loss.

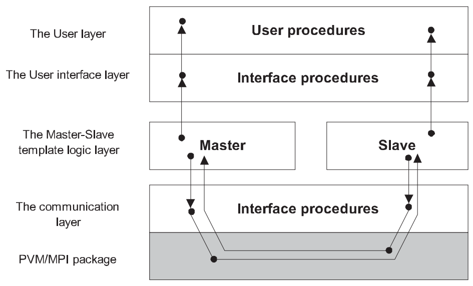

Figure 2. The Master-Slave template logic layer operations.

2.4. The Communication layer

The communication layer is an interface layer, its goal is to hide the communication primitives of the MPI [4] or PVM [2] packages. We define an interface for each MPI or PVM procedure which may have been used in conventional programming. The communication layer procedure list includes data packing, unpacking, message probing and the parallel program finalizing. Each procedure invokes either the PVM or MPI communication primitives depending on the active global setting.

3. THE APPLICATION EXAMPLES AND COMPUTATION RESULTS

In this section we present two application examples and the computational results on the homogeneous computer cluster of 7 IBM machines RS6000 running AIX operating system.

3.1. The prime number search problem

The problem definition. Find all prime numbers in $[0,\dots,M]$, assume the primes from $[0,\dots,\sqrt{M}]$ are already known.
The task list. The task list is obtained by dividing the $[\sqrt{M},\dots,M]$ into K intervals: $\{t_i, i=1,2,\dots,K\}$, where $t_i : \left(\sqrt{M}+m(i-1), \sqrt{M}+mi\right]$ and $m = \left\lceil \frac{M-\sqrt{M}}{K} \right\rceil$. The task list is not stored explicitly, it is generated during the program run-time.
The computation procedure. We made two sets of experiments. In the first set we use the sieve of Eratosthenes for each of the interval t_i. In the second set we used the Trial Division method.
Implementation. The procedure `M_prepare_job_pool` reads the initial data from the data file. It prepares the data structure:

```
struct t_init_data{ long int M, K; }.
```

The procedure `S_initialize` allocates the slave data structures and computes the primes from $[0,\dots,\sqrt{M}]$. The procedure `M_take_piece_from_pool` tracks the interval being processed and announces whether the last job has been submitted. It prepares the description of one job (the interval t_i):

```
struct t_data{ long int i; }.
```

The procedure `S_calculate` performs the prime number search in the requested interval. It returns the number of the primes found:

```
struct t_result{ long int plu; }.
```

The procedure `M_add_to_result` accumulates the result.

Table 1.
The prime number search in $[0; 10^9]$ using The Sieve of Eratosthenes algorithm.

Processors, p	Time, T_p	Speedup, $S_p = \frac{T_1}{T_p}$	Efficiency, $E_p = \frac{S_p}{p}$
1	834.37	1.00	1.00
2	455.51	1.83	0.91
3	296.96	2.81	0.94
4	221.03	3.77	0.94
5	178.66	4.67	0.93
6	153.02	5.45	0.91
7	129.66	6.43	0.92

Table 2.
The prime number search in $[0; 10^8]$ using the Trial Division algorithm.

Processors, p	Time, T_p	Speedup, S_p	Efficiency, E_p
1	1492.42	1.00	1.00
2	753.75	1.98	0.99
3	523.66	2.85	0.95
4	401.18	3.72	0.93
5	317.54	4.70	0.94
6	253.81	5.88	0.98
7	213.81	6.98	0.99

The parameter K should be selected $K > p$. The bigger value of K the better load balancing is achieved. On the other hand, each task t_i should be complex enough to consume up to several seconds of a processor time for the algorithm to be efficient. We used $K = 100$ in all computations of this problem.

The computational results are shown in Tab. 1 and Tab. 2. For both computational methods we obtain good speedup and efficiency values.

3.2. The calculation of the Mandelbrot set

The problem definition. Search the $N \times M$ points for the Mandelbrot set in the rectangle $A = [x_1, x_N] \times [y_1, y_M]$ (we use $A = [-2, 1.25] \times [-1.25, 1.25]$ in our computations).
The task list. The task list is obtained by dividing the second coordinate of A into K parts: $\{t_k, k = 1, 2, \ldots, K\}$, where

$t_k = \left\{(x_n, iy_m) : n = 1, \ldots, N,\ m = (k-1)\left\lceil \frac{M}{K} \right\rceil + 1, \ldots, k\left\lceil \frac{M}{K} \right\rceil\right\}$,
and $x_n = x_1 + \frac{x_N - x_1}{N}(n-1)$, $y_m = y_1 + \frac{y_M - y_1}{M}(i-1)$. The task list is not stored explicitly, it is generated during the program runtime.
The computation procedure. Each point $a = (x, iy)$ of the area A is tested by the iteration loop $z_{t+1} \to z_t^2 + a$, $z_0 = 0$, until the loop counter t reaches value of 5000 or condition $|z_t| > 2$ is satisfied.
Implementation. The procedure `M_prepare_job_pool` reads the initial data from the data file. It initializes the data structure:

```
struct t_init_data { int N, M, K; }.
```

The procedure `S_initialize` allocates the slave data structures. The procedure `M_take_piece_from_pool` tracks the t_k being processed and announces whether the last job has been submitted. It prepares the description of one job:

```
struct t_data{ int k; }.
```

The procedure `S_calculate` performs the search of the Mandelbrot points. It returns the the point map:

```
struct t_result{ char color[1000*50]; }.
```

The procedure `M_add_to_result` accumulates the result into a file.

Table 3.
The Mandelbrot set search problem for $N = 1000$, $M = 1000$, $K = 20$.

Processors, p	Time, T_p	Speedup, S_p	Efficiency, E_p
1	173.28	1.0	1.00
2	91.20	1.9	0.95
3	61.88	2.8	0.93
4	50.96	3.4	0.85
5	41.26	4.2	0.84
6	34.66	5.0	0.83
7	29.87	5.8	0.82

The computational results are shown in the Tab. 3. We note that efficiency in this example is lower due to the big amount of the result data which has to be passed from Slave process to the Master.

4. CONCLUSIONS

From the numerical results we conclude, that the Master-Slave Template library is efficient for a set of problems, where the initial problem is decomposable into a set of tasks. The tasks should be selected in such a way, that the execution time of one task is considerably bigger than the communication time needed for data exchange. The one-step parallelization procedure is convenient for the user of the library.

REFERENCES

[1] T.L. Freeman and C. Phillips. *Parallel Numerical Algorithms.* Prentice Hall, New York, London, Toronto, Sydney, Tokyo, Singapore, 1991.

[2] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, Cambridge, 1993.

[3] R.Šablinskas. *Investigation of algorithms for distributed memory parallel computers.* Vytautas Magnus University, 1999.

[4] M. Snir, S. Otto, S. Huss-Lederman, D. Walket and J. Dongarra. *MPI:The Complete Reference.* The MIT Press, 1996.

Programų išlygiagretinimas, panaudojant programinius šablonus

R. Šablinskas, M. Baravykaitė

Straipsnyje praplečiamos [3] idėjos apie uždavinių, kuriuos galima spręsti šeimininkas – darbininkai tipo algoritmais, lygiagrečiųjų programų konstravimą, panaudojant pusiau automatinio programų išlygiagretinimo įrankius (Šeimininkas – darbininkai (ŠD) biblioteką). ŠD biblioteka naudotina stambaus ir vidutinio grūdėtumo uždaviniams. Ji gali būti taikoma bet kokiam uždaviniui P, kurį galima išskaidyti į užduotis $P = \cup_{i=0}^{N} t_i$. Efektyviausia ŠD biblioteką taikyti, kai visos užduotys t_i yra nepriklausomos.

Straipsnyje parodoma, kad lygiagretusis algoritmas yra sukuriamas automatiniu būdu, panaudojant ŠD biblioteką, su sąlyga, kad vartotojo programa tenkina tokius reikalavimus:

1. Pagrindinę programą turi sudaryti duomenų inicializavimo, vienos užduoties t_i skaičiavimo ir rezultatų apdorojimo blokai.

2. Vartotojas turi apibrėžti pradinių duomenų, vienos užduoties duomenų ir rezultatų struktūras.

Pertvarkius programą, kad tenkintų šiuos reikalavimus, lygiagrečioji programos versija gaunama kompiliavimo metu.

Straipsnyje pristatomos pusiau automatinio išlygiagretinimo idėjos, kuriant nepriklausomus programinius sluoksnius/lygius.

Pateikiami pirminių skaičių radimo ir Mandelbrot aibės skaičiavimo programų pavyzdžiai.