A Template Library to Facilitate Teaching Message Passing Parallel Computing

Jerome L. Paul[1]
University of Cincinnati
Cincinnati, OH 45221-0030
+1 513 556 1818
jerry.paul@uc.edu

Michal Kouril[2]
University of Cincinnati
Cincinnati, OH 45221-0030
+1 513 556 4461
mkouril@ececs.uc.edu

Kenneth A. Berman[1]
University of Cincinnati
Cincinnati, OH 45221-0030
+1 513 556 5205
ken.berman@uc.edu

ABSTRACT
This paper discusses a template-based approach to aid in introducing the upper-division undergraduate (or first year graduate) to the rapidly emerging message passing parallel computing paradigm. Our template library facilitates an accelerated MPI programming learning environment that can realistically be included as one topic among many in an algorithms course. One template module is based on a backtracking solution to the satisfiablity problem (SAT), which the student first solves in the sequential setting. With the aid of a modified template, the student then develops a simple parallel SAT solver. The template includes such things as I/O functions, allowing the student to focus on the algorithm itself. The parallel part is partially provided by the template, with indicators given in places where the student needs to plug in missing MPI function calls. The students are excited about this hands-on-experience in the increasingly important world of message passing parallel computing, which might be missed if their curriculum does not include a course devoted to this topic.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education*

General Terms
Algorithms, Experimentation, Theory

Keywords
Teaching aid, parallel processing, Beowulf clusters, MPI, SAT, template library

[1] Partially supported by NSF Grants No. 9871345 and 0521189

[2] Partially supported by a fellowship grant of the Ohio Board of Regents

SIGCSE'06, March 1-5, 2006, Houston, Texas, USA.

1. INTRODUCTION
Cluster computing implemented in hardware by Beowulf clusters and in software by the message passing paradigm as supported by MPI (Message Passing Interface) [5] has recently emerged as a powerful and increasingly important computational environment. Thus, it is important that the CS undergraduate be exposed to this new paradigm. However, often the undergraduate curriculum is too tightly constrained to allow for a full course devoted to this topic. Also, since learning MPI imposes an additional requirement on students and teachers alike, the message passing parallel computing concept might be skipped due to its complexity and background requirements. With these things in mind we have created a template library to facilitate teaching message passing parallel computing modules that can be easily integrated into the standard upper-division undergraduate (or first year graduate) course in algorithms. Our focus has been on developing MPI programming modules that can be generated using a general parallel backtracking framework that we have developed, but we have also generated modules outside of this framework. The template library not only simplifies the material for the students, but also eases the burden on the instructor without compromising the concepts themselves. This library will eventually be available on-line.

Even though Beowulf clusters are relatively inexpensive to acquire and maintain as compared to other parallel computing environments, another impediment to covering message passing parallel computing might be a lack of access to such clusters. While MPI libraries can run on a single processor machine (that is, with no speedup), the learning experience is significantly enhanced by running MPI code in an actual parallel machine. In response to this we are in the process of dedicating a portion of one of our Beowulf clusters to serve the needs of the general educational community. Thus, not only is our template library available to the general public, but utilizing it on an actual Beowulf cluster will also be made available.

Most of our MPI programming templates involve the backtracking design strategy. There are several reasons for focusing on the backtracking (and, more generally, on parallel depth-first search) for our template library. First of all, backtracking is one of the major design strategies that are typically included in an algorithms course. Second, backtracking problems are well suited to parallelization using the fundamental master-worker parallel design strategy. Moreover, parallel backtracking illustrates nicely yet another fundamental concept, namely, the alternating computation/communication cycle paradigm. Third, there are important problems with many applications, such as the satisfiability (SAT) problem for Boolean expressions, for which

backtracking is the most commonly used design strategy. Indeed, there is a very active research community devoted to designing ever more efficient SAT solvers, and a large suite of benchmark problem instances is available on which to test solutions. Thus, the student can be exposed and gain insight into leading-edge research on SAT. Finally, we have developed a general parallel backtracking framework (BkFr, [3]) that can be used, if desired, to generate additional modules according to the instructor's preference.

We have tested our template library in our two quarter upper-division course in algorithms that is required for all CS majors. The students were very enthusiastic about including this introduction to the message passing parallel programming environment as part of the algorithms course. In the first quarter, with the aid of a template the students developed a sequential SAT solver using various heuristics to bound the search, and tested their results against a given standard suite of benchmark problems. Then in the second quarter, using our template library they developed a parallel SAT solver and compared it to the results obtained with their sequential solvers. In a semester course, this scenario could still be followed by assigning the sequential SAT solver problem early (or another problem of the instructor's choice), and the parallel SAT solver problem toward the end of the term. In any case, with the aid of the template library, it is quite feasible to give the student a solid introduction to the message passing parallel paradigm, but at the same time not spending more time than typically devoted to one of the many usual topics included in the standard upper-division algorithms course.

A textbook written by two of the authors ([1]) has a chapter devoted to message passing parallel algorithms, and includes high-level pseudocode for basic process communication functions such as synchronous and asynchronous send and receive instructions. An appendix in [1] includes actual MPI code for some of the algorithms given in pseudocode within the text. Our experience indicates that the treatment in [1] is adequate for those wishing to introduce MPI programming as one topic amongst many in an algorithms course. For those giving an entire course on the subject, there are now some good books devoted to teaching these topics ([6], [7]).

Template-based learning approaches in other programming contexts have been used in the past with success. To give some examples, in [9] a dynamic analysis framework for analyzing "fill-in-the-gap" Java exercises is discussed. The concept of "example-based" (Pascal) programming is presented in [4], and in [8] a project involving template-based programming in chemical engineering is described.

To illustrate our template-based MPI programming library, we focus on one template module, namely SAT. However, we have also developed modules for a number of other backtracking problems (sum of subsets, n-Queens, and so forth), as well as numerical problems such as matrix multiplication.

The rest of the paper is organized as follows. First we start with the SAT problem description, and then we show examples of the sequential version and how it is assigned to students. Then we discuss the extension of the sequential concept to the parallel environment and conclude with an evaluation of the success of the project and how we intend to enhance it in future work.

2. SAT PROBLEM DESCRIPTION AND BACKTRACKING SOLUTIONS

The problem of determining whether or not a given Boolean expression is satisfiable (i.e., has a truth assignment to the Boolean variables in the expression that renders the expression true) is a fundamental problem in computer science. Applications of this problem include circuit verification, cryptography, automatic theorem proving, combinatorial search, and so forth. In view of the importance of SAT, there is a great deal of ongoing research devoted to finding improved SAT solvers. There is even an annual international conference dedicated to theory and applications of satisfiability testing (International Conference on Theory and Applications of Satisfiability Testing).

A Boolean expression is in Conjunctive Normal Form (CNF) if it is a conjunction of clauses. Each clause is a disjunction of literals, where a literal is either a variable or negation of a variable. Since every Boolean expression has an equivalent CNF expression, focus has been given to CNF expressions. Moreover, there is even a DIMACS standard for inputting CNF expressions, and a large suite of CNF benchmarks are available on-line. This makes the problem particularly suitable to use as an introduction to message passing parallel computation. In particular, in our classroom projects we used random 3SAT (each clause has exactly 3 literals) benchmarks ([10]).

MPI is a library of functions that are used in connection with high-level languages such as FORTRAN, C, C++ and provides parallelism via the message-passing paradigm. We have chosen to implement our template library using the C version of MPI, since it allows the student to focus on the parallel programming issues related to the problem. CNF expressions are input (DIMACS standard) as shown in Table 1.

Table 1. Example of a partial CNF file

CNF File	Description
`p cnf 20 91`	DIMACS header CNF format 20 variables, 91 clauses
`4 -18 19 0`	$x_4 \vee \overline{x}_{18} \vee x_{19}$
`3 18 -5 0`	$x_3 \vee x_{18} \vee \overline{x}_5$
`-4 -8 -15 0`	$\overline{x}_4 \vee \overline{x}_8 \vee \overline{x}_{15}$
`-20 7 -16 0`	$\overline{x}_{20} \vee x_7 \vee \overline{x}_{16}$
`10 -13 -7 0`	$x_{10} \vee \overline{x}_{13} \vee \overline{x}_7$
`-12 -9 17 0`	$\overline{x}_{12} \vee \overline{x}_9 \vee x_{17}$
`17 19 5 0`	$x_{17} \vee x_{19} \vee x_5$
`. . .`	. . .

Current SAT solvers typically use a backtracking-based DPLL algorithm [2]. In our exercise modules we use a simple SAT version without the DPLL extension. The solution to the backtracking algorithm involves a series of decisions that have an associated state space tree that models all such possible decisions. Backtracking is a systematic algorithm that searches for a solution in a given state space tree.

The sequence in which the variables are picked in the state space tree is typically determined by heuristics. Heuristics have a great influence on the speed of the solver. Students experience the

importance of heuristics in the sequential version where they implement various different heuristics.

3. SEQUENTIAL VERSION

As mentioned in the introduction, students are first assigned the problem of generating a sequential SAT solver. To accelerate the student's ability to create their own SAT solver, we generated a C program template for the sequential version that included the functions described in the Table 2. An example of the template code given to the students is given in Figure 1.

Table 2. Description of the provided functions

Provided function	Description
`read_cnf(FILE *fin);`	Read in CNF file in DIMACS format
`init_variables();`	Initialize all variables to unknown
`print_solution();`	If a solution was found this function prints it in the standard format
`print_unsatisfiable();`	If no solution exists this function reports it in the standard format
`check_all_clauses();`	Check all the clauses given the current assignment Returns SAT if the current assignment satisfies the formula, or UNSAT if a contradition was found or UNDECIDED if some unsatisfied clauses are left.

```
typedef enum {
       NOT_ASSIGNED,
       TRUE,
       FALSE
} t_variable;

typedef enum {
       UNDECIDED,
       SAT,
       UNSAT
} t_sat;

/* remember where each clause starts */
int clauses_array[MAX_CLAUSES];
/* all clauses delineated by zeros */
int all_literals[MAX_LITERALS];
/* variable assignment */
t_variable variables_array[MAX_VARIABLES+1];
/* number of variables, clauses */
int variables, clauses;

t_sat backtrack() { /* return SAT if the satisfiable
    assignment is found, UNSAT otherwise */
    /* FIXME */
}

int main() {
    int err = read_cnf(stdin);

    /* check for error during read_cnf */
    if (err != 0) return (err);

    /* assign all variables NOT_ASSIGNED */
    init_variables();
    if (backtrack() == SAT) print_solution();
    else print_unsatisfiable();
    return 0;
}
```

Figure 1. Example of part of the template function in the sequential version

The student's main responsibility is to write the backtracking portion utilizing various heuristics. In order to aid a student with the development environment, the following "quick start" is given to the student (see Figure 2.).

Quick start:

1) log in to gatekeeper (or your favorite UNIX machine)

```
gatekeeper ~>
```

2) download the skeleton source code

```
gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/↩
              class_sat/testsat.c
```

3) download the verifier

```
gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/↩
              class_sat/verifier.c
```

4) download a few cnf examples

```
gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/↩
              class_sat/uf20/uf20-01.cnf

gatekeeper ~> wget http://www.ececs.uc.edu/~mkouril/↩
             class_sat/uf20/uf20-02.cnf
```

5) compile the skeleton

```
gatekeeper ~> gcc testsat.c -o testsat -Wall -g
```

6) compile the verifier

```
gatekeeper ~> gcc verifier.c -o verifier
```

7) test the skeleton

```
gatekeeper ~> ./testsat < uf20-01.cnf

s SATISFIABLE
v 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 0
```

8) test the skeleton with the verifier

```
gatekeeper ~> ./testsat < uf20-01.cnf | ./verifier uf20↩
-01.cnf
c~~ verifier: ERROR some clauses left (v lines don't ↩
define an implicant)
c~~ verifier: ERROR: Original clause left: c~~ ↩
verifier: -5 -8 -15 0
```

As you can see the skeleton returns the wrong solution due to the missing backtracking function. Please come up with your own backtracking function.

Once your backtracking function works correctly the output from the verifier will change to

```
c~~ verifier: OK: SATISFIABLE claimed, and implicant ↩
is correct
```

Figure 2. Quick start given to the students

As indicated by Figure 2, the students will then complete their SAT solvers by fixing the template, compiling, executing, and checking their results using the verifier. Fixing the template involves writing a backtracking function, as well as implementing a heuristic to determine the branching order in the associated state space tree. The student is asked to implement three different versions based on the following orderings of the variables.

a) Order the variables by their indices. This is a static ordering, that does not depend on the particular problem instance.

b) Order the variables based on the number of clauses they appear in at the start of the program. This heuristic is problem dependent, but the ordering is fixed (static) throughout the computation.

c) Order the variables dynamically by choosing the next variable based on the number of remaining unsatisfied clauses each variable is in.

The reader is referred to Chapter 18 in [1] for further details.

4. PARALLEL VERSION

In the parallel version we utilize the master-worker paradigm, where the master process hands out tasks (initial paths in the state space tree) to the workers. The master (node with MPI rank=0) precomputes a fixed depth covering set of initial paths, (see Figure 3), and hands out the next unused path to any worker node currently idle. Typically the covering set has more leaf nodes than the number of nodes involved in the parallel computation.

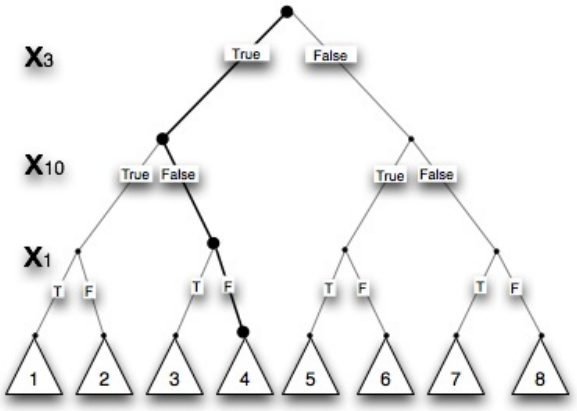

Figure 3. Typical covering set of fixed depth 3 for a suitable heuristic. An initial path $x_3, \overline{x}_{10}, \overline{x}_1$ is indicated.

A worker (node with non-zero MPI rank) upon receiving an initial path from the master starts the backtracking function that alternates between computation (backtracking) and communication. The communication portion is simply checking for a message sent by master indicating that someone already found a solution and the worker can shutdown. Upon receiving such a "die" message, the worker then propagates this message to its children in a user-defined broadcast tree so that eventually the message is received by all processes. This user-defined asynchronous broadcast is used instead of the MPI provided synchronous broadcast that locks all nodes. We can not use the MPI synchronous broadcast function since we don't know in advance the state (idle or working) of individual workers. The master initiates the user-defined asynchronous broadcast of a "die" message after exhausting the covering set and receiving an "I'm done" message from everyone, or upon receiving an "I found a solution" from a worker.

In order to focus on parallelization issues, the template for the parallel version includes code for the heuristic that orders variables before each decision based on the product of the number of remaining clauses the variable is in positively and negatively. However, the code for this heuristic is straightforward, and was actually part of the student's responsibility in the sequential version. Figures 4 and 5 show portions of the code from the parallel version of the template.

```
. . .
while(running==1) {
  if (idle == 0) {
     /* worker received task to work on */
     t_sat result=backtrack(myid, p);
     idle=1;
     switch (result) {
     case SAT: {
       /* SAT solution was found */
       /* get solution from the SAT solver */
       /* FIXME MPI, 0 */

       /* send it to the master */
       /* FIXME MPI, 1 */

       /* send idle message to the master */
       /* FIXME MPI, 1 */
     } break;
. . .
```

Figure 4. A portion of the worker code template with missing parallel code

```
t_sat backtrack(int myid, int p) /* return SAT if the
satisfiable assignment is found, UNSAT otherwise */
{
 while(1) {
  /*
   * communication cycle:
   * check to see whether we should end the computation
   * prematurely - i.e., whether an MPI message with
   * MPI_TAG = DIE has been received
   */
#ifndef STANDALONE  //run parallel code
  int flag=0;
  MPI_Status status;
  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG,
             MPI_COMM_WORLD, &flag, &status);
  if (flag == 1) {
     if (status.MPI_TAG != DIE) {
       fprintf(stderr, "Worker(%d):
                    Unexpected message\n", myid);
       exit(1);
     }
     MPI_Recv(NULL, 0, MPI_BYTE, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);

     /* tree like continuation of
        broadcast of DIE message */
     /* FIXME MPI, 2 */

     /* if I'm the last processor
        send message back to the master */
     /* FIXME MPI, 1 */
     return UNDECIDED;
  }
#endif
 /* computation cycle: purely sequential code */
 /* FIXME MPI, 0 */
 }
}
```

Figure 5. Template version of the parallel backtracking function executed by workers

The template that we provide the students also includes a "stand alone" configuration that allows the student to debug the basic backtracking portion of their code in a purely sequential setting (i.e., independent of any MPI code). This stand alone feature allows them to verify their programming logic before introducing the parallelism provided by MPI. When proceeding to the parallel version, the template is missing MPI calls that they are required to fill in. The places where MPI is missing are clearly marked with a comment (FIXME), and even the number of recommended MPI function calls is provided (e.g., `/* FIXME MPI, 2 */` means that two MPI statements are required). Similar to the quick start given in Figure 2 for the sequential version, we provide the student with a quick start for the parallel version that reflects the implementation details of our current MPI environment.

Remarks: The template we generated for extending the sequential SAT version to the parallel version actually used a simplified version of a general backtracking framework (BkFr) that we have developed ([3]). In particular, BkFr includes the possibility for idle workers to request more work by splitting the state space subtree assigned to another worker.

5. CONCLUSION AND EVALUATION

Using template modules we have developed a method of introducing the message passing parallel programming paradigm as a topic to be included in the standard upper-division algorithms course. One particularly important module consists of a sequential SAT solver, together with a parallel version created using a master-worker parallel design strategy and implemented using MPI on a Beowulf cluster. We have used the SAT programming assignment in several quarters so far, with good success. Students really appreciate the opportunity to be exposed to message passing parallel programming. Another positive aspect to the SAT solver problem is the availability of standard I/O formats, as well as extensive benchmarks for evaluating code performance. As an additional incentive we ran a competition of the correct solvers on a set of benchmarks, where the fastest solver received extra bonus points. This emulates a scenario that exists in the SAT solver research community in which an annual competition is run to determine the currently fastest SAT solver based on a test-bed of problem instances. The student is thereby given an insight into the current state of the art in this important topic.

6. FUTURE WORK

Currently the only access to our Beowulf cluster is through the command line interface as delineated in our quick start handout. In our opinion it would improve the accessibility of this technology if a web version of this method existed. We are currently developing a simple web interface that will support log in, editing, debugging, execution, and profiling. We have several clusters available to us, and in the future we plan on dedicating one of these clusters to on-line access by the general educational community. We also plan on extending our template library to a wide range of additional problems.

7. REFERENCES

[1] Berman K. A. and Paul J. L., Algorithms: Sequential, Parallel and Distributed, Thomson Course Technology, 2005.

[2] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. *Communications of the Association of Computing Machinery* 5 (1962) 394–397.

[3] Kouril, M. and Paul, J.L., A Parallel Backtracking Framework (BkFr) for Single and Multiple Clusters. *Conf. Computing Frontiers*, ACM Press, 2004.

[4] Neal, L. R. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Wings For the Mind* K. Bice and C. Lewis, Eds. CHI '89. ACM Press, New York, NY, 63-68. 1989.

[5] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Special issue on MPI.

[6] Quinn M. J., Parallel Programming in C with MPI and OpenMP, McGraw-Hill Science/Engineering/Math, 2003.

[7] Pacheco P., Parallel Programming with MPI, Morgan Kaufmann, 1996.

[8] Silverstein D.L., Template Based Programming in Chemical Engineering Courses, *American Society for Engineering Education*, 2001.

[9] Truong N., Roe P. and Bancroft P.: Automated Feedback for "Fill in the Gap" Programming Exercises, *Australasian Computing Education Conference* (2005) 117-126.

[10] Uniform Random-3-SAT, http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/RND3SAT/descr.html.