PopKern: A User-Tunable Package for Parallel Combinatorial Optimization

Daisaku Yokoyama and Takashi Chikayama
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8654, Japan

This paper introduces PopKern, a parallel combinatorial optimization library. PopKern is a general-purpose library package that allows domain-specific fine tuning. PopKern provides typical basic parallel combinatorial optimization algorithms. Users can enjoy parallel processing without paying any attention to parallel processing details. We have designed and implemented PopKern on KLIC, an implementation of a concurrent logic programing language. Through our preliminary evaluation with traveling salesman problems, its high potential in parallel computation performance has been shown.

1 Introduction

Combinatorial optimization problems are encountered frequently in various industrial fields. Solving these problems requires massive processing power. In recent years, parallel processing technologies are getting wider acknowledgment as inexpensive multiprocessor systems have become available in the market. Highly parallel processing seems to be the main stream in numerical processing and on-line transaction nowadays. Naturally, we want to apply the parallel processing technology to combinatorial optimization problems [1]. Fortunately, these problems can be easily divided into subproblems isolated well from one another, that is, computation of each subproblem does not depend much on intermediate results of other subproblems. This feature becomes an advantage for parallel processing.

In spite of this, parallel processing has not been applied widely to problems of this category. One of the reasons is the difficulty of writing parallel processing programs.

When writing parallel processing programs, we have to be careful on communication between and synchronization among many processors. Debugging parallel programs is quite hard, because bugs in synchronization often disappear or appear at a different place even when the same program is run with the same data. A subtle difference of communication timing may change the result drastically. This is a typical chaos situation.

Using a programming language and/or a system with automatic synchronization and side-effect free semantics would reduce the difficulty. However, learning a new programming language with unfamiliar semantics is accepted only as a last resort by programmers in application fields.

Libraries that parallelize the users' code would be an answer. In fact, several such libraries have been developed. Most experts in the combinatorial optimization area, however, would not use them. Most of these existing libraries are either for problems of a specific domain or with limited performance.

Solving combinatorial optimization programs efficiently requires domain-specific tuning technique. Such technique reduces the amount of required computation so drastically that domain-specific tuned sequential programs often beat more straightforward massively parallel programs. Most of such tuning schemes are too complicated to be represented as a small set of parameters of a general purpose package. On the other hand, building well-tuned systems for each of the many problem domains is impracticable, as it requires expertise in both parallel processing and domain-specific issues.

In this paper, we introduce a parallel library package, called PopKern (for Parallel OPtimizer KERNel). PopKern is a general purpose library package for combinatorial optimization that allows domain-specific fine tuning. With PopKern, users can define details such as representation of problems and solution candidates, routines handling them, and routines to visualize intermediate results for further tuning. These type and subroutine definitions can be given without worrying about details of parallel processing, such as communication, synchronization, or load distribution.

This paper is organized as follows. Section 2 gives an overview of the PopKern library. Its implementation is described in section 3. Preliminary evaluation results are given in section 4. Finally, conclusion and future work are given in section 5.

2 PopKern

2.1 Basic design

PopKern is constructed by the following three functional components.

Combinatorial optimization library PopKern provides solvers for several representative combinatorial optimization algorithms. We divided each of these algorithms into problem-domain-independent parts and problem-domain-specific parts. The domain-independent parts are built into the PopKern kernel, as components of the combinatorial optimizer library. All the domain-specific parts, data types representing the problem and candidate solutions, for example, are not defined in PopKern. The kernel treats data of these types as black-boxes and calls user-defined routines for handling them. Subroutines that directly access these data are to be

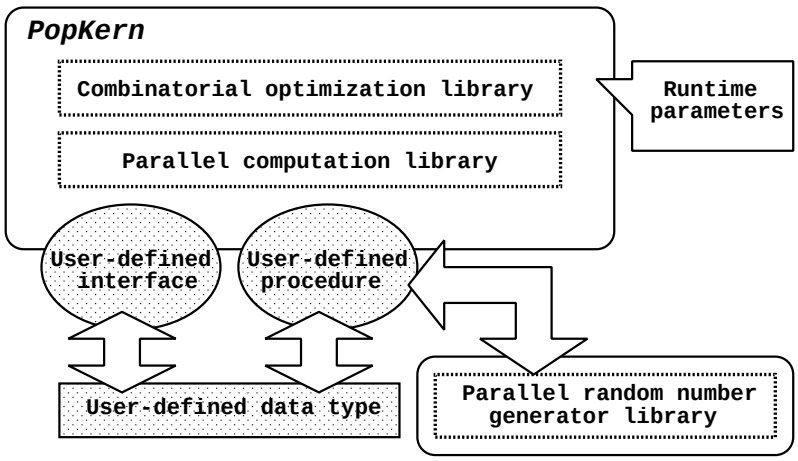

Figure 1: The design of PopKern

supplied by the users of the package, compiled and linked together with PopKern to obtain executable image. Writing these subroutines requires problem domain knowledge but not parallel processing expertise.

Parallel computation library This library component treats inter-process communication, synchronization, and load distribution. The combinatorial optimization library components use these functions to solve the problem in parallel. The user doesn't need to pay any attention to parallel processing at all. The kernel takes care of inter-processor communication and required synchronization, treating user-defined data as black-boxes. Serialization of user-defined data (encoding and decoding to and from byte streams for communication) is done by user-defined subroutines.

Parallel random number generator library This library provides a pseudo-random number generator suited for parallel processing. Using this random number generator, generated random number sequence of each PE becomes reproducible, even when parallel execution order of this library function is changed. Thus, the execution behavior of whole PopKern system is reproducible, even when the user calls this generator function freely and independently on each PE. This greatly helps debugging.

PopKern provides solvers using the following four representative parallel combinatorial optimization algorithms.

- Hill Climbing
- Branch and Bound
- Simulated Annealing
- Genetic Algorithm

The branch and bound algorithm searches all the possible combinations and finds out the optimal solution. Other algorithms find near optimal solution using heuristic methods.

2.2 Basic flow

Basic operational flow of PopKern used in common in all the four algorithms is as follows.

1. Reads the problem from a file, and distribute the problem to all the processing elements (PEs) .

2. Starts finding the solution on each PE with the selected algorithm. Each PE holds a solution candidate with the lowest cost the PE has ever seen. The cost is user-defined value that indicates the desirability of the solution candidate. This solution candidate is called the *local candidate* solution.

3. When the local candidate solution is renewed, calls the termination check routine, defined by the user. The user can define acceptable quality here. If the solution's quality is satisfactory, the computation is terminated. The renewed local candidate solution can be visualized on this occasion.

4. The best solution among all the local candidate solutions is called the *global candidate* solution. The global candidate solution is held in the manager process. When the global candidate solution is renewed, user's termination check and visualization routine are called in the same fashion as at local candidate solution's renewal.

5. Writes the final global candidate solution on termination of the computation. As described above, the user can define the termination condition by supplying check routines. In addition, runtime arguments indicating the maximum execution can be passed to PopKern.

```
typedef struct euclid_tsp_problem {
  int num_cities;
  struct city_coordinates *coords;
} popkern_problem;

typedef int *popkern_solution;
```

Figure 2: An example of data structure definition for TSP

2.3 Common program parts defined by users

To accomplish the basic operational flow described in the previous section, users have to define several program parts. Users can program these parts in C language, without paying any attention to parallel processing.

Problem/Solution type definition PopKern doesn't know how the problem and solution are represented. The user has to define these structures as types on C language. The problem structure contains the combinatorial optimization problem, which will be kept unchanged through the computation. The solution structure represents a possible solution, modified through computation. In the traveling salesman problem(TSP), for example, the problem structure contains coordinates of cities, and a solution structure contains a sequence of city numbers which reflects the (possibly incomplete) route of salesman traveling. Figure 2 shows an example of user-supplied code.

Problem input/Solution output Before computation, PopKern should read in the problem. For this purpose, the user have to prepare a routine which reads the problem from a given file. Similarly, a routine which writes out the computed solution into a file has to be defined.

Serialize/Unserialize problems/solutions For parallel execution, the problem and solution data structures have to be moved among processors. Unfortunately, PopKern doesn't know the representation of problem and solution. Thus, the user has to convert these data from the user-defined representation to byte-strings that PopKern can deal with, and vice versa. The PopKern kernel is responsible for inter-processor communication and synchronization for the byte strings. Figure 3 and 4 shows

```
void
popkern_encode_problem(const popkern_problem *problem)
{
  int k;
  popkern_send_long(problem->num_cities);
  for (k = 0; k<problem->num_cities; k++){
    popkern_send_double(problem->coords[k].x);
    popkern_send_double(problem->coords[k].y);
  }
}
```

Figure 3: An example of serialize problem routine for TSP

an example code for TSP.

In addition, the user should define the cost function for solution candidates, termination check, and some algorithm-specific functions.

User can define the visualization function of the solution candidate if necessary. If user want to apply their own memory management technique such as reference counting, memory management primitives for user-defined data structures that are called from the GC mechanism should be defined. As in many cases, these optional user-defined routines are unnecessary, dummy routines are used by default.

For programming these user-defined routine, PopKern provides some primitives :

- Serialize/Unserialize primitives,
- Error handling primitives, and
- Message display primitives.

The serialization primitive converts from one number used in user's code to one byte-string allocated by PopKern kernel. Some examples are shown in figure 3 and 4. `popkern_send_long` etc. are serialization primitives. The error handling primitive controls the execution. `popkern_error` function in figure 4 is an example. The message display primitive is provided for orderly use of the console in a parallel processing environment.

```
void
popkern_decode_problem(popkern_problem *problem)
{
  int k;
  problem->num_cities = popkern_receive_long();
  problem->coords =
    (struct city_coordinates *)calloc(problem->num_cities,
      sizeof(struct city_coordinates));
  if (problem->coords == NULL)
    popkern_error(
      "Couldn't allocate memory in reading problem spec");
  for (k=0; k < problem->num_cities; k++){
    problem->coords[k].x = popkern_receive_double();
    problem->coords[k].y = popkern_receive_double();
  }
}
```

Figure 4: An example of unserialize problem routine for TSP

2.4 Parallel combinatorial optimization algorithms

Four different solvers are provided with four different algorithms in the current version of PopKern.

Parallel Hill Climbing

The hill climbing algorithm searches only a limited solution space, and is not so useful when applied alone. However, this algorithm has quick convergence. Thus, using this algorithm in combination with other algorithms for improving candidate solutions becomes a very powerful search technique.

Algorithm This algorithm repeats search for a lower-cost solution in the neighborhood of the current candidate. In parallel processing, each PE runs this method independently starting from a different initial solution.

User-defined parts Generation of random initial solutions. Selecting a better solution in the neighborhood of one solution.

Parallel Branch and Bound

Algorithm A set of nodes in the search tree is the data structure to be defined by the user. PopKern system orders this node-set to select one node and make branches, checking upper and lower bounds, and possibly prune branches. With this algorithm, PopKern tries dynamic load distribution.

PopKern creates a global manager that knows which PE has the node-set and which does not. When one PE exhausts its node-set, the PE asks the global manager for more nodes. The global manager asks other PEs that may have nodes to divide its node-set into two and send one to the requesting PE.

User-defined parts The user should define the node-set data structure. Also the user has to prepare routines for selecting a node from a node set and for branching from that node, checking upper and lower bounds, and so on. In addition, dividing and joining of node-sets are left for user definition.

Parallel Simulated Annealing

Algorithm The simulated annealing algorithm has difficulties in cooling schedule. To avoid this, PopKern uses the temperature-parallel simulated annealing algorithm[2] . The main idea of this algorithm is that each process performs annealing at a constant temperature that differs from process to process, and occasionally exchange solutions. The better solution candidate tend to shift to the lower temperature with this probabilistic exchange sequence. It is not necessary to match the number of annealing processes with the number of physical PEs.

User-defined parts The user should provide subroutines for generating random initial solutions and mutating candidate solutions randomly. Temperature setting of each PE and deciding solution exchange interval are also defined by the user.

Parallel Genetic Algorithm

Algorithm Like the branch and bound algorithm, PopKern treats the set of genes as a user-defined black-box. To parallelize the algorithm, we use the island model. PopKern evolves the set of genes at all PEs independently, as an analogy to evolution in islands. After every certain evolutionary step, each pair of neighboring PEs exchanges a part of their genes, like migration of animals.

User-defined parts The user should define the data type representing a set of genes, subroutines for a generation change, and dividing and joining methods of sets of genes. The interval of migration also must be defined by the user.

2.5 Using existing software with PopKern

Programs using combinatorial optimization algorithms such as simple branch and bound, naturally consists of some standard basic components. We designed PopKern according to this natural style. Thus, adjusting existing source code for PopKern should be easy except for little additional code such as serialization routines.

On the other hand, PopKern cannot be used when a user wants to apply an algorithm that PopKern does not provide. For example, branch and bound algorithm with iterative deepening is not supported by the current PopKern system. Simulated annealing *without* temperature-parallel framework, or genetic algorithms *without* the island model, are not supported either. These frameworks are deeply related with the way of parallelization. Our principle of making casual users stay away from parallel processing details resulted in a system providing only those parallelization frameworks and thus search frameworks chosen by the system designers.

3 Implementation

PopKern is implemented upon KLIC [3] which is a portable implementation of concurrent and parallel logic programming language KL1 [4]. KLIC is an implementation of KL1 that translates KL1 programs into C programs. This scheme ensures portability of not only the function of the system but also machine level execution performance as far as appropriate C compilers are available on host systems. KLIC has parallel implementations [5,6] based on many standard and non-standard communication and synchronization mechanisms, including shared memory, PVM, and MPI.

KLIC has a unique system extension feature called "generic objects", a framework that enables adding new data types to the logic programming world. This feature smoothly bridges the gap between declarative concurrent programming and procedural sequential programming. We use this for treating user-defined data. The user-defined data structure and functions are encapsulated into the generic objects, and PopKern system treats these data as KLIC objects. Thus, communication and synchronization of these data are treated as those of KLIC objects.

4 Preliminary Evaluation

Preliminary evaluation of PopKern was carried out through sample traveling salesman problem (TSP) solvers. TSP has been studied very well and many powerful techniques have been developed [7]. Our preliminary evaluation, however, is done with quite simple and naive techniques.

The platform of evaluation is SUN Ultra Enterprise E/4000 (Ultra SPARC 166MHz $\times$ 10). The version of KLIC system is 3.003. KLIC provides several versions with different low-level implementation of interprocessor communication. In the evaluation, a version based on the message passing model using the shared memory as the communication media.

4.1 Sequential overhead

We implemented a sequential TSP solver with the hill climbing algorithm. The hill climbing algorithm searches near optimal solution. The evaluation program provides simple search with the 2-opt exchange method. Two different versions with the same algorithm are compared: one programmed totally in C and the other using PopKern for execution control. Comparing these two should reveal basic overhead of PopKern.

We measured execution time until the specific solution is found with the same problem and with the same pseudo-random numbers. The execution time is the average of five runs.

The overhead of PopKern system is only about 1%, which might be said negligible.

4.2 Speedup in branch and bound

We measure the speedup of the branch and bound algorithm solver with parallel processing. The branch and bound algorithm searches all the possible combinations and finds the optimal solution. Branching and lower-bounding are quite simple methods without any heuristics. Figure 5 and 6 show the result.

A12, A14, A16, and B14 in the figure mean four different TSP problems. (problem A12 has 12 cities, problem A16 has 16 cities, and so on). The speedup (figure 6) means execution time with one PE to compute the optimal route divided by execution time with n PEs for the same task.

The figure 6 shows that this program achieves very high parallel processing speedups. Super-linear speedups are shown in all problems with less than four or five PEs. This is because, good solutions found at an early stage pruned branches effectively. It means that the sequential version failed to visit

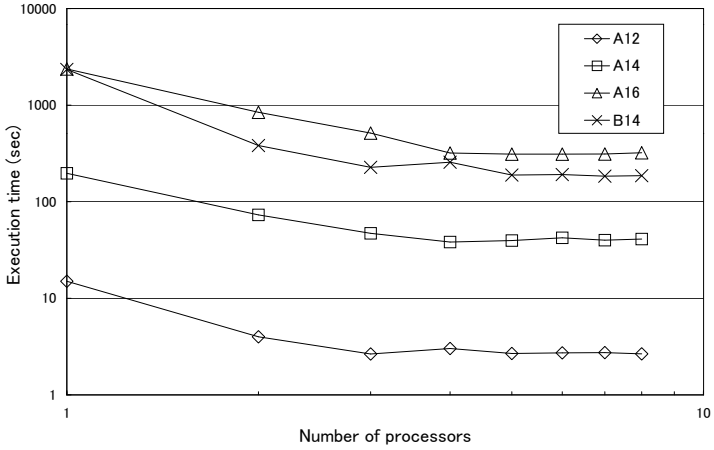

Figure 5: The execution time of the branch and bound algorithm

promising branches first and improving the algorithm by changing the branch visiting order would offset the merits of parallel processing. However, it can be said that parallel processing made up the deficiency of the algorithm.

Using four or five PEs achieves the most efficient parallel processing for all the problems. The performance tends to be worse with more PEs. This is because the effect of waste speculative search.

Although these problems have different scales of search space, all problems show similar speedups. Thus, the number of processors that achieves the best performance seems to be decided by the grain size of node-sets. The grain size is decided by the user through the dividing method of the node-set defined by the user. The grain size should be decided depending on number of processors and ratio of computation and communication performance. PopKern will be able to figure out these indicators and pass them to the user-defined routine for making better decisions.

4.3 Speedup of simulated annealing

Figure 7 shows improvement progress of costs of best solution candidates with parallel simulated annealing. The simulated annealing algorithm searches near optimal solution.

All these executions are with 32 temperature-bands: using a single PE means 32 temperature-bands on one PE, and using eight PEs means four

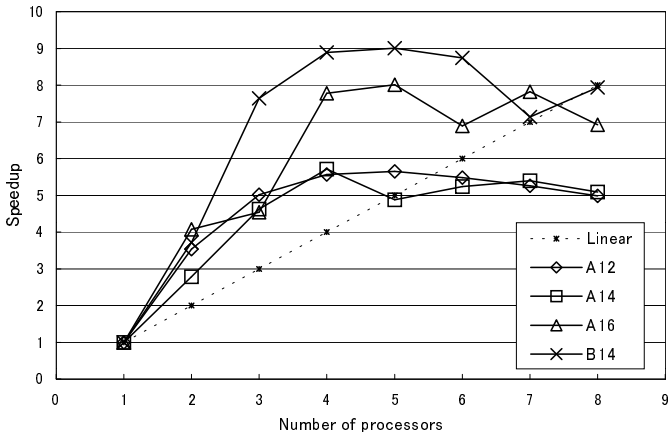

Figure 6: The speedup of the branch and bound algorithm

temperature-bands on every PE, and so on. Random swapping of two edges between cities is tried as mutation of candidate solutions. Hill climbing is also applied when no mutations are accepted for a while. We evaluate this algorithm with TSP of 280 cities. The optimal solution of this problem is known, and the optimal cost is about 4300. We see from figure 7 that the more processors we use, the better the solution candidates become. That is, parallel processing helps to improve solution candidates more quickly.

We try to pick up the time when the cost reaches to 10000, 9000, ... and so on. These execution time ratio is shown in figure 8. All these data show nearly ideal speedup.

Thus, we can say that the PopKern library is effective in simulated annealing algorithm.

5 Conclusion and future work

We see in shortcomings of existing parallel combinatorial optimization problem libraries lack of domain specific tuning abilities. We developed a parallel library package called PopKern that allows domain-specific fine tuning. To realize this, PopKern leaves definitions of types and handling routines for data structures for problem solving to users, while providing overall parallel processing control for representative optimization algorithms. According to our preliminary evaluation, PopKern shows high potential of parallel computational speedup.

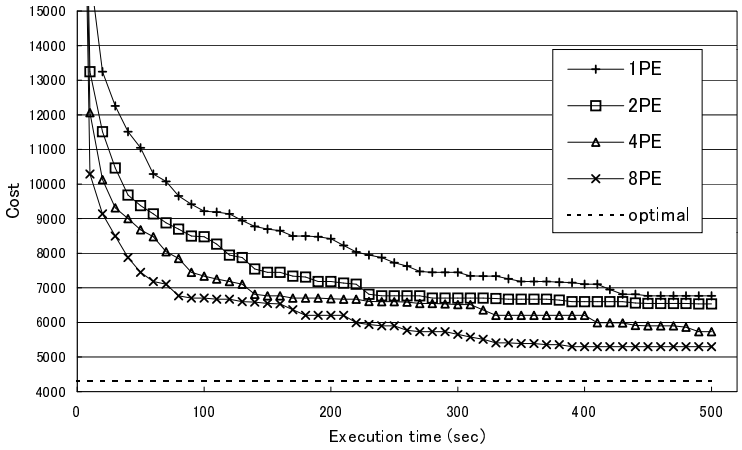

Figure 7: Improvement of solution candidates of the simulated annealing

PopKern has a flexible structure to be applied to a wide area. We would like to try out the systems to diverse problem domains, especially in practical industrial domains with huge search spaces, more complicated restriction, and effective domain-specific tuning techniques.

To improve PopKern system further, two directions are considered.

First, we want to make PopKern more friendly to users. For example, providing a graphical interface that can visualize the optimization process and control the computation interactively. Providing a rich variety of GUI templates would benefit users.

Second, we want to implement more algorithms for enabling to use in wider application domains. For instance, mini-max tree search will make PopKern applicable to game programming.

Acknowledgments

This research was partly supported by the Information-technology Promotion Agency, Japan.

References

1. Panos M. Pardalos, Leonidas S. Pitsoulis, T. Mavridou, and Mauricio G. C. Resende. Parallel search for combinatorial optimization: Genetic algorithms, simulated annealing, tabu search and GRASP. In *Parallel*

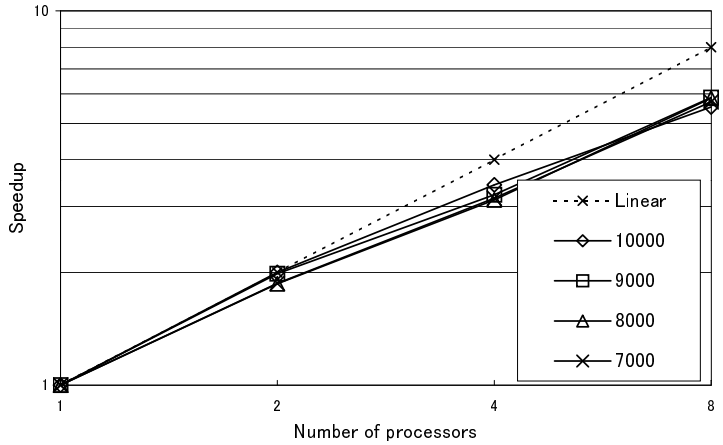

Figure 8: The speedup of the simulated annealing algorithm

Algorithms for Irregularly Structured Problems, volume 980 of *LNCS*, pages 317–331. Springer-Verlag, 1995.

2. Kouichi Kimura and Kazuo Taki. Time–homogeneous parallel annealing algorithm. ICOT Technical Report 673, ICOT, 1991.
3. Takashi Chikayama. KLIC: A portable parallel implementation of a concurrent logic programming language. In *Parallel Symbolic Languages and Systems*, volume 1068 of *LNCS*, pages 286–294. Springer-Verlag, 1995.
4. Kazunori Ueda and Takashi Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, December 1990.
5. Masao Morita, Nobuyuki Ichiyoshi, and Takashi Chikayama. A shared-memory parallel execution scheme of KLIC. ICOT technical report, ICOT, 1994.
6. Kazuaki Rokusawa, Akihiko Nakase, and Takashi Chikayama. Distributed memory implementation of KLIC. Technical Report 3, 1996.
7. Gerhard Reinelt. *The Traveling Salesman: Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer-Verlag, 1994.