PMF: A Multicore-Enabled Framework for the Construction of Metaheuristics for Single and Multiobjective Optimization

Deon Garrett[1]

Department of Computer Science
University of Memphis,
Memphis, TN, USA
deong@acm.org

Abstract. This paper describes the design and implementation of the Parallel Metaheuristics Framework (PMF), a C++ framework for the construction of single and multiobjective metaheuristics utilizing Intel's Threading Building Blocks library to allow easy parallelization of computationally intensive algorithms. The framework demonstrates a generic approach to the construction of metaheuristics, striving to provide a general representation of core operators and concepts, thus allowing users to more easily tailor the system for novel problems. The paper describes the overall implementation of the framework, demonstrates a case study for implementing a simple metaheuristic within the system, and discusses a range of possible enhancements.

1 Introduction

Many metaheuristics for optimization problems, and almost all aimed at multiobjective optimization, exhibit a large degree of potential parallelism that can be exploited to provide significant gains in execution speed. Historically, most researchers have not focused on this potential gain. One prominent reason is simply that such researchers are accustomed to running many separate trials of each algorithm in order to report statistically significant results, and thus they already have a free form of parallelism in the form of simply running multiple independent trials at once.

However, in recent years, increases in processing power on the desktop have come almost exclusively in the form of additional processing cores, while the processing power of a single core has largely stagnated. This has led to an increase in interest in technologies and techniques that enable programmers and users to take fuller advantage of the parallel processing power on almost every desktop and laptop. It is generally expected that this trend will only accelerate reaching more and more cores in the next few years.

In 2007, Intel announced the release of their open source Threading Building Blocks (TBB) package. TBB provides a high level, task-oriented approach to developing software to better utilize the many processing cores available in modern CPUs. This work describes the design and implementation of a framework for constructing parallel metaheuristics called, appropriately, the Parallel Metaheuristics Framework (PMF) which uses TBB to enable researchers to build metaheuristics that take full advantage of the power of modern desktop and server class processors.

The remainder of this document is arranged as follows. Section 2 briefly describes several existing packages and compares them to PMF with respect to their design goals. In Section 3, the guiding principles driving the development of PMF are put forth. Sections 4 and 5 briefly describe Intel's TBB library with respect to its usage in PMF. Section 6 details a case study: building a simple genetic algorithm in PMF. Finally, Section 7 details areas of ongoing work with PMF.

2 Previous Work

There are a number of other high-quality open source frameworks for experimenting with different flavors of metaheuristics, each with their own focus and strengths. In this section, some of the most popular packages are briefly described.

ECJ [1], is one of the most complete packages available for research into evolutionary computation. It includes particularly strong support for genetic programming, but all manners of evolutionary algorithms are supported. ECJ provides excellent support for parallel execution of the algorithms as well. However, the coverage of multiobjective optimization in ECJ is somewhat sporadic, and the focus of ECJ is firmly on evolutionary methods, as opposed to general metaheuristics including various forms of local search.

In the C++ world, ParadisEO [2] is a very good option for general purpose metaheuristic research. It includes not only very complete support for numerous multiobjective techniques, but also allows inclusion of several local search and other methaheuristic algorithms. Furthermore, in the form of ParadisEO-MOEO [3], the framework has been extended to provide support for parallel execution of the search algorithms. However, much of the focus of the parallelism in Paradiseo-PEO is intended to model parallel metaheuristics such as Island model Genetic Algorithms rather than conventional algorithms executed in parallel.

3 The Parallel Metaheuristics Framework

PMF aims to serve a different role. By focusing exclusively on multicore parallelism as opposed to distributed multiprocessing, PMF aims to put a thinner layer of complexity atop the well-known terminology that researchers and practitioners are already familiar with. As well, TBB provides a fairly simple concurrency model which is used consistently throughout the framework, and it should be easy for researchers to adapt their own algorithms using this framework. It is the view of the author that there is a large opportunity, due to increases in desktop computing power, for persons outside the normal academic research environment to employ metaheuristics to solve important problems, using only the desktop class computers already available. PMF attempts to fill this niche by providing a simple configuration mechanism by which existing algorithms may be evaluated, and by providing a simpler model of multicore parallelism that can be exploited by practitioners without specialized training in concurrent programming.

PMF is thus intended to fill a need within the metaheuristics community for an accessible package providing relatively simple support for constructing metaheuristics that take advantage of modern multicore processors, particularly with respect to multiobjective optimization using hybridizations of multiple techniques. The combination of, for example, multiobjective evolutionary algorithms with directed local search methods has proven popular and effective on a number of real world problems. PMF makes the process of building and experimenting this type of algorithm in particular very simple. It requires very little awareness on the part of the user of issues such as locking and synchronization which can often obscure the details of the algorithms. The use of Intel's Threading Building Blocks package to abstract away as much of the parallelization code as possible allows the user to concentrate on combining the ready-made modules in PMF with their own custom code as simply as possible.

PMF adheres to other guiding principles as well. One such principle is that all configuration information must be available at runtime in a simply expressed form, and must be easily managed from within the code. All configuration is thus performed through simple text files. In the author's view, XML is too cumbersome for the generally simple requirements of algorithm configuration. Instead, PMF attempts to provide easy ways for the programmer to retrieve configuration values in a completely type-safe way using template specializations. Figure 3 shows an example of a typical configuration file used with PMF.

```
# configuration file for running NSGA-II on
# an instance of the multiobjective QAP
algorithm = nsga2
encoding = permutation
problem = qap
problem_data = /path/to/data/file.dat
population_size: 100
terminator: evaluation_limit
max_evaluations: 100000
metric: evaluation_count
metric: generation_count
metric: hypervolume
hypervolume_reference_point: 5000000 5000000
trials: 10
```

Fig. 1. Sample PMF configuration file

There are a few things to note about this sample file. All options are specified generally as keyword/value pairs, separated by either a colon (":") or an equals ("="). Comments are allowed, and are considered to run from any "#" character to the end of the line. The value portion of the configuration lines may contain multiple values, denoting a list or vector, as in the case of `hypervolume_-reference_point`. In addition, keywords may be repeated, in which case PMF forms a list of values associated with that keyword in the order in which they appear in the configuration file.

This system provides a great deal of flexibility while retaining a very simple syntax. From the point of view of the programmer, a single (template) function is provided with several specializations to handle retrieving values from the user-specified configuration. Typically, all that is required is to declare a variable of the desired type to hold the specified value, then pass the variable to the `parameter_value` function, along with the keyword in question.

Figure 3 shows example code reading a value from the configuration file, and demonstrates a number of concepts. In PMF, all accepted keywords are defined in a dedicated namespace so as to serve as a convenient source of in-code documentation. Thus the expression, `keywords::PROCESSORS` refers to a string defined in that namespace to denote the parameter name used to specify the number of processors available. The second argument is the variable which will be used to store the retrieved value. Note that the `parameter_value` function is a template which provides instantiations for all supported data types (most integral and floating point types, `bool`, and standard template library lists and vectors of supported types). PMF attempts whenever possible to maintain strong and strict type safety; it encourages the use of unsigned types when appropriate, const correctness, etc, and the specification of `num_processors` as unsigned is an example of this goal. Finally, the third parameter indicates that specification of this value is optional. In the case of the number of processors, the default behavior is to allow the TBB library to apportion threads as it likes.

```
unsigned int num_processors;
configuration::parameter_value(keywords::PROCESSORS,
                               num_processors, false);
```

Fig. 2. Code to read a value from the configuration file.

Out of the box, PMF includes support for a large group of optimization algorithms and techniques. In single objective optimization, the broad class of generational and steady-state evolutionary algorithms are supported, including many popular operators for selection, crossover, mutation, and replacement or environment selection. Several local search operators are also available, and may be embedded either within a population based approach such as one of the evolutionary algorithms or within one of a number of local search containers that implement strategies such as random restarts, simulated annealing, and tabu search algorithms.

Multiobjective optimization is supported through the NSGA-II, SPEA2, and ϵ-MOEA algorithms. Again, each supports the incorporation of local search techniques using any of the supported implementations. In addition, multiple options are available concerning how the local search operators adapt the multiple objectives of the problem into a single-objective function suitable for hill climbing behavior. In addition, several multiobjective local search strategies are supported.

PMF treats all problems as multiobjective, but provides comparators that can single out the first or any specific objective or scalarize the objectives to yield a single valued function. These mechanisms allow several simple means to provide classical single objective optimization.

4 Overview of TBB

Threading Building Blocks, or TBB, is Intel's attempt at a C++ library that abstracts much of the tedium of developing parallel versions of certain algorithms [4]. As evidenced by the name, TBB is a thread-oriented library, as opposed to one aimed at distributed memory parallelism such as MPI. TBB provides template functions implementing parallel algorithms such as `parallel_for`, `parallel_while`, and `parallel_reduce`, and also more fine-grained concepts related to parallelism such as locks and atomic types. To truly derive the most benefit from TBB, it is expected that the programmer will do as much work as possible within the specified parallel algorithms, resorting to manual locking and synchronization only when necessary.

As computers have continued to increase in performance, metaheuristics and optimization in general have become available to a wider audience. Many practitioners who could benefit from optimization will be limited to desktop class hardware, and without the support infrastructure to help manage the complexity of massively parallel clusters. Multiprocessing is notoriously difficult, and without this support infrastructure, the problems of writing correct multithreaded code for the desktop will likely be difficult to overcome for many smaller organizations. TBB aims to help solve this problem by removing as much manual intervention as possible from the process of writing multithreaded code. As such, it is hoped the PMF can provide a base on which new problems and algorithms may be built while requiring a minimum of explicit thread management. In practice, this means expressing metaheuristics in terms of the building blocks provided by the TBB library.

5 Evolutionary Algorithms and the Reduce Operator

In PMF, most forms of evolutionary algorithms are expressed as instances of the `parallel_reduce` function. In functional programming languages, `reduce` is a well-known function which combines a sequence of values into a single value through repeated application of a specified operator. For example, given the a hypothetical functional language, an expression such as

```
reduce + 0 [1,2,3,4,5]
```

generates an expression such as

```
(((((0+1)+2)+3)+4)+5)
```

Note that the initial zero in the expression comes from the second argument passed to the `reduce` function. In essence, `reduce` simply applies the given operator to the given primer argument and the head of the specified list. It then applies the operator to the just computed return value and the next element from the list, continuing in this fashion until the list has been exhausted, returning the result of the final application of the given operator.

One important thing to notice about the expansion of the `reduce` function is that, at least for the case of addition, the operator imposing the reduction is associative, and thus we are free to perform the calculations in any equivalent grouping. One such grouping might be expressed as

```
f [1,2,3,4,5] = ((0+1)+2)+((3+4)+5)
```

Expressed in this manner, we see an obvious opportunity for parallelism. A dual-core machine could perform the first "mini-reduction" on one core while simultaneously performing the second on the other core. With no communication overhead, the speedup could be perfectly linear.

The only remaining step is to specify how the algorithm is to combine the results of the two distinct reductions to obtain the correct solution. In the case of our simple addition, the answer is trivial. However, we are free to use any method of combining the results of the independent parallel reductions to obtain the desired solution.

At this point, it is worthwhile to step back and consider a canonical evolutionary algorithm. The basic steps of such an algorithm are shown in Listing 5. Consider the body of the while loop. The basic form is that we perform some initial setup (creating the child population), enter a for loop in which the evolutionary operators are repeatedly applied to generate the offspring population, and then we consult the replacement operator to combine the parent and child populations into the next generation's parent population.

In TBB terms, we could express the inner for loop as a straightforward instance of `parallel_for`, but each running thread would then need to coordinate access to the shared child population. Another option is thus to formulate the algorithm in terms of a `parallel_reduce`. Unlike the simple example above involving addition over a list of numbers, in this case two different operators are needed. The first must apply a single iteration of the inner GA loop, producing two offspring which are inserted into the child population local to that thread. The second operator is needed to combine all the thread-local child populations back into a single population that can be passed to the *Replacement* function to produce the next generation.

Algorithm 1 The Canonical Evolutionary Algorithm

```
 1: Generate initial population $P$ of size $n$
 2: while not done do
 3:     Create child population $P'$
 4:     for $i \leftarrow 1, n/2$ do
 5:         $(p1, p2) \leftarrow$ Selection($P$)
 6:         $(c1, c2) \leftarrow$ Crossover($p1$,$p2$)
 7:         $c1 \leftarrow$ Mutate($c1$)
 8:         $c2 \leftarrow$ Mutate($c2$)
 9:         $P' \leftarrow P' \cup \{c1, c2\}$
10:     end for
11:     $P \leftarrow$ Replacement($P, P'$)
12: end while
```

Because `parallel_reduce` (and all the other algorithms in TBB) are implemented as template functions, we must arrange some way to pass the "block" of code implementing the GA loop into the function. TBB, like the STL, makes heavy use of functors for this purpose. A functor in C++ is simply a class that provides `operator()`, thus allowing instances of the class to be called as functions. The basic model in TBB is thus to encapsulate all logic that we wish to be executed in parallel inside of a C++ functor. Any shared resources must be available inside the functor, and if the resource must be modified, we must arrange for a new copy to be allocated inside each copy of the functor. The copied resources may then be built independently in each thread, and TBB will call the functor's `join` method after all threads have completed, at which time the shared resource can be combined back into a single unified object.

This model of parallelism is highly adapted for the case of generational parallelism, i.e., algorithms which partition the work done during each generation across multiple cores. PMF has a strong focus toward hybrid algorithms, and this model of parallelism is especially useful in algorithms that embed time consuming local search algorithms inside a larger generational algorithm. Parallelization of the evaluation function, for example, is certainly possible, but PMF does not currently provide special support to address this issue. In the next section, we will demonstrate the process of building algorithms in TBB by walking through the implementation of the canonical genetic algorithm.

6 Case Study: The Simple GA in PMF

Implementing a new optimizer in PMF consists primarily of two steps. First, we must define the high level skeleton of the algorithm. Ideally, we need to express the algorithm in terms of repeated applications of some inner loop, the iterations of which may be performed in parallel. For the classic simple genetic algorithm, this task is fairly easy. As shown in Listing 5, this algorithm consists of a number of iterations of a loop which performs the defined genetic operators to fill a child population which becomes the input to the next iteration of the loop. We can thus parallelize the execution of the genetic operators. Using the parallel_-reduce template, we can then join the resulting partial child populations into a single unit in preparation for the next iteration of the loop. Figure 6 shows the code required to set up the basic skeleton of the algorithm. (Note that for brevity, the declaration and initialization of the genetic operators has been elided.)

```
void simple_ga::run()
{
    while(!terminate())
    {
        population offspring(m_pop.size());
        // note we pass pointers to ourself, our population,
        // and copies of pointers to each genetic operator
        simple_ga_loop loop(this,&m_pop,m_prob,m_sel_op,m_cross_op,
                            m_mut_op);
        // for simplicity, only show TBB's automatic method of
        // apportioning threads
        parallel_reduce(blocked_range<size_t>(0, m_pop.size()/2),
                        loop,auto_partitioner());
        // after all threads have been joined, copy the final
        // child population created by joining each thread
        copy(loop.offspring().begin(),loop.offspring().end(),
             back_inserter(offspring));
        m_rep_op->merge(m_pop, offspring, m_pop);
        iteration_completed(m_pop);
    }
}
```

Fig. 3. The simple GA control class. (from `simple_ga.cpp`)

The basic structure of this method is very simple. It creates a new child population, then initializes a functor object to perform the generation of offspring in parallel. Because a `parallel_reduce` is used, TBB will call the functor's join method before returning control to the main loop, and this method will be used to construct a single child population. That child population is then copied into the local population and merged using whatever replacement operator the user has specified (through the `m_rep_op` variable).

The work of generating the child population is performed inside the functor, shown in Figure 6 below. Note that instead of looping over the entire parent population, TBB splits the iteration into disjoint blocks represented by the TBB type `blocked_range`. Inside the functor, we then iterate over the range passed to us by TBB. The important part to notice in the loop functor is that `m_offspring` is not a pointer to a shared resource. Rather, each copy of the functor creates its own local population. In the `join` method of the functor, which TBB will repeatedly call until only one such object remains, the offspring are combined into a single population. It is this unified population which is accessed by the outer loop when all threads have finished.

```
void simple_ga_loop::operator()(const blocked_range<size_t>& r)
{
    m_sel_op->set_population(m_pop);
    for(size_t i=r.begin(); i!=r.end(); ++i)
    {
        candidate p1 = m_sel_op->apply();
        candidate p2 = m_sel_op->apply();
        candidate c1 = p1;
        candidate c2 = p2;
        m_cross_op->apply(p1, p2, c1, c2);
        m_mut_op->apply(c1);
        m_mut_op->apply(c2);
        m_prob->evaluate(c1);
        m_ga->evaluation_completed(c1);
        m_prob->evaluate(c2);
        m_ga->evaluation_completed(c2);
        m_offspring.add(c1);
        m_offspring.add(c2);
    }
}

void simple_ga_loop::join(simple_ga_loop& that)
{
    for(unsigned int i=0; i<that.m_offspring.size(); ++i)
        m_offspring.add(that.m_offspring[i]);
}
```

Fig. 4. The simple GA TBB loop functor. (from `simple_ga.cpp`)

7 Conclusions and Future Work

PMF provides a ready-to-use system that implements a number of useful optimization algorithms in a form suited for efficient execution on today's multicore hardware. In addition, it provides a reasonable simple means by which new problems and optimization techniques may be incorporated using only knowledge of standard C++ constructs.

However, there are definite areas in which improvements can be made. While PMF performs very well, there is some overhead in the mechanism by which it supports arbitrary encodings. An earlier version of the software relied on extremely heavy use of C++ template metaprogramming techniques and type traits to automatically infer at compile time the correct type of numerous parameters. The result was that there was a statistically significant increase in performance in the earlier system. However, the usability of the earlier system suffered due to the drawbacks inherent in extensive use of template metaprogramming in C++. It remains an open question if there is a way to reclaim some of this performance without incurring the most severe of these penalties.

Another issue in PMF is that the focus on generational parallelism means that the framework provides little effective support for certain classes of algorithms. A principled and consistent mechanism for providing additional forms of parallelism is needed to allow maximum flexibility in extending PMF.

PMF has an extremely flexible method of providing performance metrics and termination criteria, but by nature, this type of information demands global access. Particularly for researchers making comparisons between algorithms, it is important that, for example, if we specify a maximum number of fitness function evaluations, that the algorithm halt in precisely that allotment of evaluations. However, this means that each thread must have shared access to these global pieces of information. TBB provides atomic variables and locking primitives, and PMF uses those to ensure data consistency, but this imposes a slight performance hit that becomes more significant as more and more cores are enabled. Thus a further avenue for improvement is to implement a generic and customizable mechanism by which the trade-off between absolute accuracy in the metrics and the inherent contention of resources can be managed by the user.

Finally, there are dozens of compelling search and optimization techniques available that can be incorporated into PMF. The current focus of PMF is on hybrid metaheuristics for multiobjective optimization, which implies a stronger representation in the areas of multiobjective evolutionary algorithms and local search techniques. However, additional paradigms are needed to provide a more complete package for researchers and practititions to build upon.

PMF is available on Github (*http://github.com/˜deong/pmf*) under a nonrestrictive BSD license. For length requirements, this paper omitted detailed performance measures. The reader is encouraged to download the PMF package and experiment with the built-in algorithms and problems to get a complete understanding of the performance characteristics of the framework.

References

1. Luke, S., Panait, L., Balan, G., et al.: Ecj 19: A java-based evolutionary computation research system
2. Cahon, S., Melab, N., Talbi, E.G.: ParadisEO: A framework for the reusable design of parallel and distribued metaheuristics. Journal of Heuristics **10**(3) (May 2004) 357–380
3. Liefooghe, A., Jourdan, L., Talbi, E.G.: A unified model for evolutionary multiobjective optimization and its implementation in a general purpose software framework: ParadisEO-MOEO. Technical Report Research Report RR-6906, INRIA (2009)
4. Reinders, J.: Intel Threading Building Blocks. O'Reilly (2007)