A Parallel Skeleton Library for Embedded Multicores

Xin Liu, Jingyu Zhou, Daqiang Zhang, Yao Shen, and Minyi Guo

EPCC, Department of Computer Science
MOE-MS Key Laboratory for Intelligent Computing and Intelligent Systems
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, P.R. China
{naveyliu, zhangdq}@sjtu.edu.cn {zhou-jy, shen_yao, guo-my}@cs.sjtu.edu.cn

Abstract—Many SoCs adopt multicore architectures. As a result, embedded programmers are also facing the challenge of parallel programming. We propose a parallel skeleton library that can be used on embedded multicores. Our library is implemented in standard C++ using template features. We propose two parallel skeletons to support common program patterns on multicores. In our skeleton library, programmers can easily choose underlying parallel implementations with no code changes. Experimental results show that many applications can take advantage of these two skeletons for performance improvement, sometimes better than hand-parallelized code.

Keywords—parallel skeleton; embedded multicore; template library;

I. INTRODUCTION

For high performance and energy efficiency, more and more processors in SoCs adopt multicore designs and embedded systems gradually embrace the thoughts of parallel computing. As a result, embedded programmers are also facing the challenge in multicore era — parallel programming. Furthermore, the term multicore in SoCs community could mean more than in the desktop and server fields. A multicore design in SoC could refer to a shared-memory SMP architecture, a heterogeneous (single-ISA or multiple-ISA) single chip and even an asymmetric multiprocessing (AMP). The diversity of designs brings extra confusion and difficulties when software developers writing parallel programs for them. Thus, it is desirable to have an effective approach to develop parallel programs for embedded multicores.

Parallel skeleton (a.k.a Algorithmic Skeleton [1]) is a high-level programming model for parallel and distributed computing. A skeleton is intended to be an efficient implementation of a commonly encountered parallel behavior on some specific machines. Starting from a basic set of skeletons, more complex patterns can be built by composing simple ones. Parallel skeleton intends to represent algorithms by common program patterns while hiding the complexity of underlying implementations. Historically, research on parallel skeleton has focused on large-scale parallel machines and distributed systems. These parallelization efforts are mainly implemented by using MPI. Recently, researching interests have shifted to multicore platforms, such as Sequoia [2], Skandium [3], Intel TBB [4], and Muesli [5].

To the best of our knowledge, there is no dedicated research on parallel skeletons for embedded multicores. Some obstacles occur when embedded programmers try to apply existing parallel skeletons. First, the underlying parallel supports may not be available for some embedded platforms. Few embedded systems implement MPI for parallel computing. Pthread is not pervasive either. In some extreme cases, embedded applications are running on bare board, where OS and thread runtime don’t exist at all. Second, many architectural assumptions may not hold on some SoCs. For example, many parallel skeletons based on thread models require a cache coherent shared-memory system, which may not be the case on some multicore SoCs. Third, high overhead makes it difficult for embedded programmers to apply modern languages such as C++ or Java. Existing parallel skeleton libraries heavily use object-oriented features, which incur significant costs comparing to native C.

We propose a parallel skeleton library for embedded multicores. It is implemented as a C++ template library leveraging rich static information of applications and template metaprogramming techniques [6] to perform source-to-source transformations. Skeletons in our library are template classes. By performing skeleton instantiations at compile time, applications can save the runtime overhead traditionally associated with object-oriented skeleton libraries. In addition, because parameters are static constants, it opens a wide door for compiler optimizations such as inline and constant propagation [7].

Our library ships two new skeletons for multicore developments:

• Par is a skeleton for parallel programs supporting arbitrary nesting, thus allowing parallel computation on multiple dimensions. This skeleton is inspired from the computing model of CUDA [8] and OpenCL [9].
• **Hier** is a skeleton to parallelize tasks by means of hierarchical division. It is useful to specialize algorithms for different levels of storage, as multicore processors usually organize memory in a complex hierarchy.

We have successfully applied these two parallel skeletons on a number of application benchmarks. Our experiences show that our skeleton library supports both CPU and GPU processors with different parallel models such as Pthread, Intel TBB, and OpenCL. Experimental results also demonstrate the performance of using the template library is comparable to those hand-parallelized code, sometimes even better.

The rest of this paper is organized as follows. Section II presents the overview of our skeleton library. Section III explains the design and implementations of two skeletons proposed for embedded multicores. Section IV evaluates performance our skeleton library using a number of parallel programs. Section V discusses related work on parallel skeletons. Finally, Section VI concludes the paper and discusses our future work.

II. A Skeleton Library for Embedded Multicores

A. Rationale

The primary goal of our skeleton library is to express parallel algorithms while hiding implementations of multicore. The advantage is particularly meaningful in the sense of the variety of SoC multicores. Embedded community lacks of industrial standard to define parallel APIs. Some of them even run on bare board without any runtime supports. Therefore, our skeleton library does not explicitly depend on any specific parallel programming models. It is designed to be extensible enough to adapt for different multicore platforms.

Because performance and energy must be considered for embedded applications, embedded programmers are usually reluctant to apply C++ or Java because the incurred runtime overhead is high. We decide to avoid the design of using object-oriented features such as function objects or virtual functions to implement higher-order functions, as our skeleton library strives to reduce runtime cost. We choose to apply skeletons as much as possible at compile-time. It is worth noting that SoCs are inherently application-specific. Hardware configurations are usually determined in software development stage. For example, programmers are aware of how many cores are available. Moreover, embedded applications have few motives to support other platforms, thus software parameters are also more determinable, such as the frame dimensions in video codecs. Similar to the ASIC design, the more specific, the more information are hardwired into chip. Our skeleton library exploits these static information in both hardware and software at compile-time to customize executed tasks.

B. Skeleton Library Overview

Our parallel skeletons is written in standard C++ and organized as a template library. We introduce two new skeletons, **Par** and **Hier**, to embedded developers.

**Par** skeleton aims to exploit simple data parallelism. Inspired from the computing model of CUDA [8] and OpenCL [9], this skeleton parallelizes kernel functions and supports arbitrary nesting. In other words, **Par** allows parallel computation on arbitrary dimensions.

**Hier** is an enhanced version of classic **Divide-and-Conquer** skeleton [10]. It is designed with memory hierarchy in mind, which borrows the idea from Sequoia [11]. This skeleton recursively divides user-defined tasks into sub-tasks and executes them in parallel.

These two skeletons are based on the emerging programming models for multicores and implemented in metaprogramming. We present their implementation details in Section III.

C. Mechanism

The foundation of our skeleton library is template metaprogramming. Template is an important feature in C++ that allows data structures and algorithms to be parameterized by types without performance penalties at runtime. In recent years, people have found that C++ compilers can execute algorithms along with delicate template classes [12] and C++ template is proved to be Turing-complete [13]. The programming technique of writing compile-time programs is known as template metaprogramming. Our approach takes advantage of it to develop parallel skeletons.

```cpp
// Template definition
template <int L, template<int> class Pred, class Func>
inline static bool __SENTINEL__ = Pred<L>::value;
void apply(){
  Func::apply();
  For<Next::value, Pred, Next, Func>::apply();
}

// Tail implementation of above template
template<int L, template<int> class Pred, class Func>
struct For<0, L, Pred, Next, Func, false> : __tail {};

// Definition of Pred class
template <int v>
struct p_lt_5 { enum { value = v < 5 }; };  // Definition of Next class
template <int v>
struct _inc { enum { value = v + 1 }; }

// Usage
typedef For<0, p_lt_5, _inc, dummy_func_wrapper<func>> I;
I::apply();
```

Figure 1. An example of parallel For.
Parallel skeletons need to manipulate functions or function objects, which are template parameters in our approach. Figure 1 illustrates the minimal example, i.e., parallel For in our library. After compilation, the equivalent code is:

```
func();
func();
func();
func();
```

In this example, the generative effect is achieved by recursive template instantiation. Note the fifth parameter of template For is a bool constant at line 5. Lines 14-18 define a partial-specialization [14] of For, where the bool value is false. In this example, template recursion happens at line 11. During compilation, the template parameter __SENTINEL__ is evaluated. If the value is true, compiler will choose the For template defined in lines 2-13 to instantiate; otherwise, specialized For defined in lines 14-18 is chosen, which ends recursion. The schematic graph is illustrated in Figure 2.

![Figure 2. Illustration of using recursion to implement For skeleton.](image)

Acting as a higher-order function, For is a template class that owns a static function. Apply is the naming convention in our library. Func is the application-specific function to be invoked in each iteration. Essentially, the static function apply builds a calling chain while applying skeletons. Because apply is an inline function, the calling overhead could be saved by compiler’s inlining expansion.

In our skeletons, we use template recursion as an idiom to achieve control flows of both branch and loop. The above example is simplified for explanatory purpose. We’ll discuss more meaningful skeletons in the next section.

III. DESIGN AND IMPLEMENTATION

This section describes design and implementations of two parallel skeletons proposed for multicores. Our skeleton library is written in ISO C++[15], thus is platform-independent and can be easily ported to new platforms.

A. Par Skeleton

1) Design: Par skeleton is the basic construct to support parallel execution of certain functions. Similar to CUDA and OpenCL, programmers write code segments that need to be executed in parallel, and then compiler and run-time system run the parallelized code with the maximum capacity of underlying hardware.

```plaintext
1 typedef Par<tail, NUM_THREAD, kernel> par_kerl;
2 typedef Par<par_blk, NUM_GRID> ParCuda;
3
4 ParCuda::apply();
```

Figure 3. A sample Par skeleton that mimics CUDA computing model.

Figure 3 illustrates how we can use Par skeleton to mimic the 3-level CUDA computing model. In this example, NUM_GRID defines the number of grid, NUM_BLOCK specifies the number of blocks in a grid, and finally NUM_THREAD is the number of threads in a block. kernel defines the function to be executed on graphic hardware. The compiler tool chain invokes CUDA compiler to translate the above code into middle language (PTX assembly) and GPU executes the code by assigning NUM_GRID * NUM_BLOCK * NUM_THREAD threads on GPU hardware.

Note that in Figure 3, Par skeleton is used for three-level nesting. In fact, Par is designed to support arbitrary nesting levels and can be used to build a parallel multi-dimension iteration space. In practice, the nesting levels are restricted by the maximal level of template recursions that a compiler can perform, which in most cases is not a problem.

The skeleton deduces arguments for kernel functions. We refer to the arguments as an iteration vector, which indicates the position of the kernel function in the iteration space. In CUDA and OpenCL, programmers explicitly write code to query such information using certain built-in functions. In contrast, the iteration vector is calculated by compiler in our approach, thus saving runtime costs.

For an n-level iteration space of \( \overrightarrow{D} = \{D_0, D_1, \ldots, D_{n-1}\} \), the thread ID is uniquely represented as:

\[
ID = i_0 \times \prod_{k=0}^{n} \frac{D_k}{D_0} + i_1 \times \frac{D_k}{D_0} + \ldots + i_{n-2} \times \frac{D_{n-2}}{D_0} + i_{n-1},
\]

where \((i_0, i_1, \ldots, i_{n-1})\) is the iteration vector. Par skeleton calls each kernel function with its vector iterator. In implementation, we model iteration space like an array in C. Thread ID is a global index in the array and the skeleton deduces iteration vector using the above formula. Figure 4 is an example of 3-level iteration space.

It is worth noting that functions applying the Par skeleton are intended to have the similar characteristics of CUDA and OpenCL’s kernel functions — the needs of communication among function instances are few or regular. Par doesn’t imply any kind of synchronizations and it is programmers’ onus to perform proper synchronization in functions.

2) Implementation: Figure 5 is a simplified version to illustrate how Par skeleton works. There are 3 template
Figure 4. A 3D iteration space: a block is a kernel function, Par skeleton will call it with iterator vector (i, j, k) as function arguments.

Figure 5. Code snippet to implement Par skeleton

```
// introduce ParImpl based on programming models
#ifdef PTHREAD_MODEL
#else
#include "par_pth.hpp"
#endif
#ifdef OPENMP_MODEL
#else
#include "par_omp.hpp"
#endif
#ifdef OPENCL_MODEL
#else
#include "par_ocl.hpp"
#endif

Par{;

template class Surr, /*the nested loop*/
uint32_t ITER = 1,
class FUNC = _undefined
>
struct Par{
// meta-program to resolve types of function and
// nesting par
...

// entry point:
inline static void apply() {
  typename mpl::vector_c<int>
  _carrier<empty>();
}

// the last level of loop, partial specialized Par<*/
template<uint32_t ITER, class FUNC>
struct Par<_tail, ITER, FUNC>{
...

inline static void
apply() {
  typename mpl::vector_c<int>
  _carrier<empty>();
}

// divide
template <class IV> inline static void
_carrier() {
  typename mpl::vector_c<int>
  _carrier<empty>();
 _Call<IV, mpl::int_<ITER>>
  _surr::template carrier<iv>();
}

// the last level of loop, partial specialized Par<*/
template<uint32_t ITER, class FUNC>
struct Par<_tail, ITER, FUNC>{
...

inline static void
apply() {
  typename mpl::vector_c<int>
  _carrier<empty>();
}

// divide
template <class IV> inline static void
_carrier() {
  typename mpl::vector_c<int>
  _carrier<empty>();
  Call<IV, mpl::int_<ITER>>
  _surr::template carrier<iv>();
  ParImpl<FUNC, _iv>::apply();
}

...}
```

parameters for a Par. The first template parameter Surr is used to support nesting, which is another Par type or a predefined _tail indicating the end of nesting. The second parameter ITER is the number of iteration. The third parameter FUNC is kernel function type. If _undefined is given, skeleton will try to determine the type of function from nested Surr. Compiler will complain if it fails.

Par processes information from input parameters by metaprogramming, redefines Surr as _surr, and then recursively apply it at line 31. We omit the details here because they are tedious and not straightly related to the skeleton. _carrier function is responsible for inserting iteration dimension into iteration vector. We implement iteration vector IV using meta-container mpl::vector_c from MPL library [16]. _carrier function can carry the IV as its own template parameter.

The template recursion lasts until the partially specialized Par (line 36-52) is instantialized, which calls platform-specific ParImpl. Different programming models have different ParImpl implementations that parallelize kernel functions in a platform-dependent manner. As shown in line 2-10, we include a specific implementation before compiler instantiates the Par skeleton, where the choice is determined by compilation macros.

B. Hier Skeleton

1) Design: Hier skeleton can divide a task into subtasks recursively. It can be used to implement Divide-and-Conquer strategy on multi-cores. In addition, memory hierarchy on SoC multicore is relatively complex comparing with commodity PCs. The strategy of customizing different subtasks on different memory levels is an effective way to express parallelism.

To use Hier skeleton, programmers need to provide a definition of a task, as illustrated in the pseudo code of Figure 6. First, the types of task parameters and returning value are defined by typedefs. Usually, they come from template parameters and correspond to the call operator function, which will be explained later. In this example, two parameters, i.e., Arg0 and Arg1, are listed. Secondly, a predicate named Pred is provided. It uses a static expression to control recursion of hierarchical division. Thirdly, programmers have to provide a Divide class that defines how to divide a task into subtasks. There are three important elements in Divide class:

- SubTask: defines the type of a sub-task. Sub-task is the same task as its parent, only with smaller inputs.
- divide: is a series of overloaded functions that are responsible for dividing input arguments into smaller ones.
- Dim: is a constant that specifies the number of dimensions used to divide input arguments. Currently the maximum allowed dimensions is two.
Optional class Reduce is used to achieve the reduction part of Divide-and-Conquer algorithm. Hier skeleton will automatically ignore reducing results of subtasks if the type of Result_t is void. Finally, a task requires a call operator definition, which is the implementation of the task in sequential semantics.

Figure 7 is an example of applying Hier skeleton to a matrix-multiplication task. It actually is the benchmark sgemm in Section IV-C.

To summarize, Hier skeleton breaks down a task hierarchically until the subtask satisfies a specific condition, and then performs the computation on a leaf node. After each level of computation, skeleton optionally performs reductions on the results of sub-tasks. In this process, Hier skeleton parallelizes and specializes tasks on different hierarchies.

```
1 template <typename ARG0, typename ...>
2 struct MyTask {
3     typedef ... Arg0;
4     typedef ... Argj;
5     typedef ... Result_t;
6     struct Pred { enum { value = [Condition Expression] }; };
7     struct Divide {
8         typedef ... SubArg0;
9         typedef ... SubArgj;
10     };
11     // optional
12     struct Reduce { ...; }
13     // leaf algorithm
14     Result_t operator() (Arg0 arg0, Argj argl) {...}
15 };
```

Figure 6. The pseudo code of a task

```
1 template <typename ARG0, typename ...>
2 struct MyTask {
3     typedef ... Arg0;
4     typedef ... Argj;
5     typedef ... Result_t;
6     struct Pred { enum { value = [Condition Expression] }; };
7     struct Divide {
8         typedef ... SubArg0;
9         typedef ... SubArgj;
10     };
11     // optional
12     struct Reduce { ...; }
13     // leaf algorithm
14     Result_t operator() (Arg0 arg0, Argj argl) {...}
15 };
```

Figure 8. Definition of hier skeleton

```
1 template <typename Task, class Tuple, class __Pred = typename Task::Pred
2     int __SENTINEL__ = Pred::value>
3 struct _hier {
4     static inline typename Task::Result_t
5         apply(Tuple & args) {
6             // platform-dependent implementation
7             return HierImpl<Task>::template
8                 apply(Task<Tuple>(args));
9         };
10     template <class Task, class Tuple>
11     struct _hier<Task, Tuple, typename Task::Pred, 1> {
12         static inline typename Task::Result_t
13             apply(Tuple & args) {
14             // leaf node, user-defined implementation
15             return hier_func_wrapper<Task>::leaf(args);
16         };
17     };
18     // hier derives proper interfaces
19     template <class Task>
20     struct Hier : hier_func_wrapper<Task>,
21         hier_basic<Task>::NR {};
```

Figure 9. Pseudocode of a HierImpl class that uses SFINAE to choose a proper function.

IV. Evaluation

This section evaluates our template library with following objectives:

- Demonstrate that our template library supports multiple running platforms, including both CPU and GPU processors using Pthread, TBB, and OpenCL libraries.
- Show that the performance of the template library is comparable to those hand-parallelized code. Specific-
cally, we evaluated the performance of our Par and Hier skeletons.

A. Settings and Benchmarks

Because our SoC board is under development, we verify and evaluate our skeleton library on off-the-shelf platforms in this study. Two multicore platforms used in our experiments are described in Table I. The harpertown contains two quad-core Xeon E5605 processors. On macbookpro, we use the chipset-embedded Nvidia GPU m5400 to execute OpenCL programs. M5400 contains two streaming multiprocessors (SMs), each consists of 8 scalar processors (SPs) and supports up to 128 threads.

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>processors</th>
<th>memory</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>harpertown</td>
<td>CPU</td>
<td>x86 quad-core 2.0G</td>
<td>4G</td>
<td>Fedora 10</td>
</tr>
<tr>
<td>macbookpro</td>
<td>GPU</td>
<td>Nvidia 9400m 1.1G</td>
<td>256M</td>
<td>Mac OSX 10.6</td>
</tr>
</tbody>
</table>

We developed and tested our template library using GCC 4.4 compiler. For GPU, we developed our library with OpenCL [9] shipped with Mac OSX 10.6. The biggest dependence of our library is boost MPL [16]. On harpertown, we link with Intel's MKL (Math Kernel Library) to perform BLAS procedures.

The benchmarks used in experiments are:

- **mandelbrot.** Calculation of mandelbrot set. The result image is 1024*768 and the iteration times are 256.
- **convolution2d.** Two-dimensional convolution operation on a 2048*2048 image.
- **mpeg2encoder.** A multimedia program in ALPBench [18]. Our input is a QCFI format(176x140) akiyo video sequence (300 frames).
- **sgemm.** A procedure in BLAS level 3, which represents two 4096*4096 dense matrices multiply.
- **dotprod.** Two vectors perform dot product. Each vector comprises of 32 million elements.
- **average.** This micro-benchmark calculates the average value of each element with its two neighbors. It is an example in TBB's reference manual [19].

B. Performance of Par skeleton

1) Micro-benchmark Performance: We apply Par skeleton to mandelbrot and convolution2d. In both cases, a two-dimensional Par transformation (i.e., a Par nested in another Par) is applied to reduce the original problem into many smaller and independent instances.

For these two benchmarks, we compare their performance on both CPU and GPU. Specifically on CPU, the following three different schemes are used: 1) Par(OpenMP), where Par skeleton selects OpenMP implementation; 2) Par(Pthread), where Par skeleton uses Pthread implementation; and 3) Manual, which is a manually parallelized implementation based on Pthread. We divide programs according to the quadrants of image on CPU. For the Par(Pthread) and Par(OpenMP) schemes, no changes are made to user code. The selection of different underlying implementation is via a macro definition, i.e., at line 2-10 in Figure 5.

Figure 10 shows the speedups on harpertown. For both benchmarks, the speedups of Par(Pthread) and Par(OpenMP) schemes are comparable to that of the Manual scheme. Mandelbrot cannot scale linearly because the amount of computation for a pixel is unknown in advance. Thus, the workload for each thread is not balanced, even though there’s no communications among threads. This is the inherent limit of a static approach and can only be overcome by dynamic parallel algorithms.

On GPU platform, the Par skeleton invokes the underlying OpenCL implementation and tasks are divided into pixels. We compare the execution time with sequential execution on a SP of GPU. The achieved speedups for mandelbrot and convolution are 7.6 and 106, respectively. The reason for higher speedup of convolution2d is that convolution2d has dozens of memory accesses for each pixel. GPU’s memory system is optimized for coalesced memory accesses. Sequential execution on a SP results in significantly inefficient memory operations.

In summary, we have shown that Par skeleton can support parallel patterns running on both CPU and GPU with different underlying libraries. The performance of code using Par is comparable to those hand-parallelized programs.

2) A Case Study: Mpeg2 is a lossy video compression standard. Many embedded applications use it because Mpeg2 provides a good tradeoff between video quality and algorithm complexity. In this experiment, we applied our Par
skeleton on the original benchmark code, which already uses Pthread for parallel execution. Because mpeg2encoder spends 99% time on putseq function, the original code uses multiple threads to process this function for each frame. The number of threads is statically defined as a macro. Thus, when applying Par skeleton, we simply define a wrapper class that calculates workload and then calls original function.

Figure 11 shows the performance of parallelization on CPU in terms of processed frame-per-second (FPS). We can observe that both Par using Pthread and Par using OpenMP outperform the original one, because Par also parallelize the workload calculation step, done by main thread in the original benchmark code. The OpenMP version performs better than Pthread because the runtime library of OpenMP maintains a thread pool, which can avoid repeatedly spawning threads for each frame.

Note that the code for Par skeleton to use Pthread or OpenMP is the same one. We only need to give a different flag to the compiler. This study shows that writing efficient multi-thread programs in Pthread is not an easy task. By adopting our skeleton library, programmers may focus on writing kernel functions and leave parallelization to the library for producing more efficient code.

C. Performance of Hier skeleton

In this experiment, we use Hier skeleton to implement sgemm and dotprod. The division continues until L1 cache can cater the data of a subtask. Figure 12 illustrates the speedups of these two benchmarks on harpertown, where Pthread is the underlying library. Sgemm achieves good scalability, over 7 times speedup on 8 cores, though sgemm needs an extra reduction for each subtask. In the implementation, we use a special callback handler to invoke reduction on another thread. As a result, Hier avoids the join operation before a reduction operation. Dotprod doesn’t scale linearly because of limited memory bandwidth. The execution time of dotprod is mainly dominated by memory operations. With more than two cores, the system bus is quickly saturated, thus limiting the performance.

D. Comparison with Other Skeleton Libraries

This experiment compares the performance of programs using Intel TBB with programs using our template library. Specifically, the average benchmark from the TBB reference manual [19] is chosen. In our approach, a one-dimension Par skeleton is used to transform original program into an 8-way parallel one on harpertown. The comparison excludes the initialization time of TBB in tbb::task_scheduler_init.
Figure 13 shows the throughput of average with different sizes of dataset. Our template library using Pthread outperforms TBB version in most datasets. The sequential code has a constant throughput.

The high throughput of Par skeleton using Pthread can be mainly attributed to the use of low-cost thread pool inside. Because we use static information to parallelize programs, the overhead of library is mainly determined at compile time. TBB uses a special dynamic algorithm based on workstealing, thus can automatically adjusts tasks in each thread at runtime to balance load. The performance of TBB is mainly subject to datasets, with high throughput for larger dataset sizes.

V. RELATED WORK

Essentially, a skeleton is a higher-order function that takes a sequential function as input and outputs a parallel equivalent. Therefore, functional languages inherently match algorithmic skeletons. HDC [10] develops parallel skeletons based on Divide-and-Conquer paradigm using a subset of the functional language Haskell [20]. Eden [21] is another Haskell-based skeleton language for distributed memory environments. In Eden programs, processes are defined explicitly while their communications remain implicit.

Although function is not the first-class citizen, it is also possible to implement parallel skeletons in imperative languages. eSkel [22] is a skeleton library written in C and runs on top of MPI by manipulating function pointers. Object-Oriented programming paradigm brings many convenient features for writing skeleton libraries. C++ function object can mimic higher-order function [23]. Muesli [5] is a C++ skeleton library that supports nesting of tasks and data parallel skeletons via MPI and OpenMP. It uses function object to implement currying and higher-order functions. Different from these approaches, our C++ library exploits the template feature of C++ language and supports multicore platforms.

Historically, research on parallel skeletons has focused on large-scale parallel machines and distributed systems. Recently, researching interest becomes to apply parallel skeletons for multicore. Skandium [3] is a Java skeleton library for shared-memory multicore systems utilizing inheritance. Sequoia [11] is a cross-platform skeleton language, including CMPs and heterogeneous multicore CellBE. It exploits parallelism by dividing of tasks and then maps subtasks to sequoia’s virtual memory hierarchy. Intel TBB [4] is a concurrency library targeting CPU multicore platforms. TBB consists of a plentiful of parallel constructs and concurrent containers and is a typical dynamic approach to parallelize tasks. It is based on task-based dynamic execution engine, which is derived from Cilk [24] and has advantage of load-balance. TBB is not a skeleton library in the sense that skeletons are high-level parallel patterns.

C++ template is a powerful source-to-source compilation mechanism, which is a feasible way to implement parallel skeletons. By performing most skeleton instantiation at compile time, the overhead traditionally associated to object-oriented skeleton libraries can be kept very small. Quaff [25] is a skeleton library that first uses template metaprogramming. Quaff implementation calls MPI to parallelize programs. Our work is different from Quaff in two aspects. First, our library is designed for complex multicores on SoCs. It doesn’t explicitly depend on any specific parallel programming model, but keeps our template library extensible to use different underlying parallel facilities. Second, Quaff reimplemented existing skeletons in metaprogramming, while we in this paper propose two new skeletons, Par and Hier, for multicores architectures.

VI. CONCLUSION

We present a parallel skeleton library designed for SoC multicores, which is implemented in standard C++ and organized as a template library. To address the challenges raised by the diversity of multicores on SoC, our skeleton library doesn’t rely on any specific parallel programming model. Instead, the library is designed with the extensibility to choose appropriate underlying implementations via macro definitions. For performance and energy-efficiency in embedded systems, our skeleton library focuses on using static information to reduce runtime overhead with template metaprogramming.

We propose two new parallel skeletons in our template library for parallelization for multicores. Inspired by the programming model on GPU, Par skeleton can generate an arbitrary level of iteration space and execute used-defined kernel functions with iteration vectors as their function arguments. Hier skeleton enhances traditional Divide-and-Conquer by supporting hierarchical division. It can parallelize tasks by mapping subtasks on multicore.

Our experimental results demonstrate that applying skeletons can make the choice of different parallel models easy. The two skeletons are effective to express parallelism for different benchmarks, and the runtime performance of proposed skeletons are comparable to the hand-parallelized parallel equivalences. In addition, comparison with Intel TBB shows that dynamic approach of parallel library could incur much more runtime overhead than our static approach. Applications running on embedded systems usually have fixed configurations, our approach thus can maximize benefits of such static information for parallelism.

In future work, we plan to implement more parallel skeletons. For instance, Pipe is a classic skeleton to execute multiple tasks in order and we plan to implement it by template expression [26] and to support complex topology graph of tasks. Additionally, we are porting our skeleton library to a FPGA board with an in-house multicore processor.
We plan to refine Hier skeleton for customized memory hierarchy of SoCs.

REFERENCES


