Meta-programming Applied to Automatic SMP Parallelization of Linear Algebra Code

Joel Falcou[†], Jocelyn Sérot[‡], Lucien Pech[*], and Jean-Thierry Lapresté[‡]

[†] IEF, Université Paris Sud, Orsay, France
[‡] LASMEA, Université Blaise Pascal,Clermont-Ferrand, France
[*] Ecole Normale Supérieure, Paris, France
falcou@ief.u-psud.fr - lucien.pech@ens.fr
lapreste,jserot@lasmea.univ-bpclermont.fr

Abstract. We describe a software solution to the problem of automatic parallelization of linear algebra code on multi-processor and multi-core architectures. This solution relies on the definition of a domain specific language for matrix computations, a performance model for multi-processor architectures and its implementation using C++ template meta-programming. Experimental results asses this model and its implementation on sample computation kernels.

1 Introduction

Scientific computing applications have become more and more demanding in terms of raw computing power. The old and easy solution of waiting for a new generation of processors is no more viable as the rise of CPU power has been slowing down. For a while, building clusters provided a realistic solution for highly demanding applications like particle physics [1], financial modeling or real time vision [2]. However, the multi-core technology [3, 4] changed the deal. As time goes, the upcoming **many-core** era will feature ready-to-use, affordable high performance computing platforms in a simple desktop machine [5]. It also becomes clear that a growing audience of developers will have to master these architectures. However, for non-specialists, writing efficient code for such machines is non-trivial, as it usually involves dealing with low level APIs (such as PTHREAD or OPENMP). Such an approach makes code more error-prone, hides domain specific algorithms and increases development cost. Several solutions have been proposed to overcome this problem, ranging from dedicated languages [6], libraries [7] or compiler extensions [8]. Those solutions, however, suffer from various limitations because they have to find an acceptable trade-off between efficiency and expressiveness. Trying to develop a generic tool for parallelizing **any kind of code** on a multi-core machine while providing a high level of expressiveness and efficiency is a daunting task. Instead, we think that such tools can be developed for smaller, **specific domains** of applications, for which accurate performance models and efficient parallelization strategies can be developed. Our work focuses on defining such a domain and providing a user-friendly tool performing automatic SMP parallelization of code, guided by an

analytical performance model. This model, similar to the threshold system used by OpenMP[8], is backed up by a high-level interface based on the linear algebra syntax introduced by MATLAB^{TM} [9] as it can be easily parallelized following a simple data-parallel scheme and fuels a large number of applications.. The tool itself is a template-based, object oriented C++ library which is able, at the same time, to provide performances on a par with hand-crafted, low-level C or C++ code on common architectures.

The paper is organized as follow : Section 2 presents our scientific computing library, NT2, and its implementation. Section 3 defines a performance model for SMP architectures, assesses its accuracy and shows how it can be integrated into NT2. Experimental results are provided in Section 4 and we conclude by proposing extensions of this work in Section 5.

2 NT2 : A High Performance Linear Algebra Library

NT2 is a C++ template library that aims at providing an efficient implementation of the most common linear algebra functions on multidimensional arrays by using a refinement of E.V.E. [9] code generation engine. It offers an API whose functionalities are close to MATLAB, including template-based wrappers to LAPACK and BLAS and transparent support for a large variety of optimizations : SIMD support for SSE2 and AltiVec, data tiling, loop unrolling, memory management options, copy on write and statically sized matrix.

2.1 A Simple NT2 Use Case

For example, consider the MATLAB code given below, performing a fixed-point RGB to YUV transformation:

```
function [Y,U,V]=rgb2yuv(I)
R=I(:,:,1);
G=I(:,:,2);
B=I(:,:,3);

Y=min(bitshift(abs(2104*R+4130*G+802*B+135168),-13),235);
U=min(bitshift(abs(-1214*R-2384*G+3598*B+1052672),-13),240);
V=min(bitshift(abs(3598*R-3013*G-585*B+1052672),-13),240);
```

The code can be rewritten in a straightforward manner with NT2 as shown below. Most functions are similar, the only difference being the need to declare variables explicitly – the `view` container being used to prevent unwanted copy of data slices – and to fix some MATLAB syntax specificities – turning `:` into `_` for example. Table 1 reports the performances obtained with MATLAB and NT2 versions, along with those obtained with a direct C version, for several image sizes (from 128×128 to 1024×1024).

```
void RGB2YUV(const matrix<int>& I, matrix<int>& y,
                  matrix<int>& u, matrix<int>& v )
{
view<int> R = I(_,_,1);
view<int> G = I(_,_,2);
view<int> B = I(_,_,3);

y=min(shr(abs(2104*R+4130*G+802*B+135168),13),235);
u=min(shr(abs(-1214*R-2384*G+3598*B+1052672),13),240);
v=min(shr(abs(3598*R-3013*G-585*B+1052672),13),240);
}
```

The relevant information is the speed-up measured between NT2 and MATLAB (Γ_M) and the overhead introduced by NT2 compared to the C implementation (ω_C) on a single core Power PC G5. Results show that NT2 is 40 to 100 times faster than MATLAB in interpreted mode, while the overhead introduced is never greater than 5%.

N	128	256	512	1024
MATLAB	44.6ms	175.8ms	487.5ms	1521.2ms
C	0.41ms	2.0ms	11.4ms	40.5ms
NT2	0.43ms	2.1ms	11.6ms	40.8ms
Γ_M	103.71	83.7	42.0	37.3
ω_C	4.89%	5%	1.75%	0.74%

Table 1. Performance of the RGB to YUV algorithm

2.2 NT2 Implementation

The implementation of NT2 is based upon a meta-programming technique known as *Expression Templates* [10]. As shown above, this mechanism can virtually eliminate the overhead classically associated to object-oriented implementations of matrix and linear algebra libraries (comparable to hand written C or FORTRAN code). In the sequel, we give a short account on the principle of this technique.

Consider for example a simple expression, such as `r=(a+b)/c`, where `a`, `b`, `c` and `r` are matrices of integers (`matrix<int>`). In an object-oriented setting, the classical approach for evaluating this expression is to overload the `+` and `/` operators. However, such an approach produces unnecessary loops and memory copies (see [11, 9] for a complete account). The idea of *expression templates* is to overload operators so that they return an object that reflects the structure of the expression in its type. This has the effect of building an expression abstract syntax tree as a first class object at compile-time.

Technically, the leaves of the abstract syntax tree will be matrices and a custom class – `node` – will be used to encode binary operators:

```
template <class O,class L,class R> struct node
{
node(const L& l,const R& r) : l_(l), r_(r) {}
L l_;
R r_;
};
```

This abstract syntax tree is obtained by overloading the classical operators for all combinations of operand types. For example:

```
template<class T,class U> node<Add,matrix<T>,matrix<U> >
operator+( const matrix<T>& l,const matrix<U>& r )
{
return node<Add,matrix<T>,matrix<U> >(l,r);
}
```

With this approach, when evaluating the expression `(a+b)*c`, the compiler builds a temporary object whose type is:

```
node<Div,node<Add,matrix<int>,matrix<int>>,matrix<int>>
```

where `Div` and `Add` are placeholder types encoding the operation associated to a node. Then, we overload the `=` operator of the `matrix` class so that it actually evaluates the assignment operator argument. In this operator, a `for` loop iterates over the elements of each arrays which size are given by the `size()` method:

```
template<class T> template<class U>
matrix<T>& matrix<T>::operator=( const node<U>& tree )
{
for(int i=0;i<size();i++)
  data[i] = Eval< node<U> >::evalAt(tree,i);
return *this;
}
```

`Eval` recursively parses the `tree` argument to produce the corresponding residual code. Depending on the type of `tree`, it proceeds differently:

- When visiting a leaf, it evaluates the matrix element for the current index:

  ```
  template<class T> struct Eval< matrix<T> >
  {
  typedef matrix<T> type;
  static inline T Do(const type& m, size_t i) { return m[i]; }
  };
  ```

- When visiting a binary node this function first evaluates the values of both node's siblings and passes the results to the embedded operator. The embedded operator itself is a simple functor providing a class method called `Do` that takes care of the actual computation:

```
template<class O,class L,class R> struct Eval<node<O,L,R>>
{
typedef node<O,L,R> type;
static inline T Do(const type& n,size_t i)
{
  return O::Compute(Eval<L>::Do(n.l_,i),Eval<R>::Do(n.r_,i));
}
};
```

Since all generated functions are candidates for inlining, most compilers are able to optimize call overhead and empty structures so that the result is the same as if we generated the code in place. For the previous example (`r=(a+b)/c`), the generated code is:

```
for(int i=0;i<size();i++) r[i] = (a[i]+b[i])/c[i];
```

A closer look at the generated assembly code validates this process. This basic technique can be enhanced by using type lists [12] to handle n-ary operators in a seamless fashion and type traits [13] to perform correct return type computation.

3 An SMP-aware Implementation of NT2

The main motivation for an SMP-aware implementation of NT2 is that many linear algebra operations are regular, exhibit a high potential parallelism and can be easily parallelized by using a simple data-parallel approach in which each core or processor[1] applies the same operation on a subset of the matrix elements.

Parallelization can be beneficial, however, only when the amount of data to be processed is above a certain threshold, because of the overhead of creating and synchronizing threads in an SMP context. It is very easy to observe "negative" speed-ups (i.e. < 1) if data sets are too small and/or overhead is too large on a given architecture. This justifies the need for performance model by which it should possible to evaluate, *thanks to a compile time process*, whether relying on SMP parallelism at run-time is worthwhile or not. In this section, we present such a performance model and show how it can serve parallelization purposes in NT2.

3.1 A Performance Model for SMP Architectures

We propose a simple performance model based on a simple interpretation of SMP speed-up $\Gamma = \frac{\tau_s}{\tau_p}$, where τ_s and τ_p are the sequential and parallel execution times. τ_s can be defined as the sum of the computation time and the memory access time :

$$\tau_s = N \cdot (\psi_c + \psi_m)$$

[1] We refer to cores or processors as **processing elements** or **PE**s.

where N is the size of the data to process (the matrix size), ψ_c is the time spent in computation per element and ψ_m the time spent in memory access per element. Similarly, we can define τ_p as

$$\tau_p = N \cdot (\frac{\psi_c}{P} + \psi_m) + \omega$$

where P is the number of **PE**s in the considered architecture and ω the overhead introduced by the parallelization process. In this model, we assume that all **PE**s share a common bus to the main memory, thus forcing the memory access to be serialized and that, for a given architecture, the end user will always use all the **PE**s available, meaning that ω corresponds to the overhead of starting and handling P threads. Hence :

$$\Gamma = \frac{(\psi_c + \psi_m)}{(\frac{\psi_c}{P} + \psi_m) + \omega/N}$$

So we have:

$$\Gamma > 1 \iff N > \frac{P}{P-1} \cdot \frac{\omega}{\psi_c}$$

To check whether it is worthwhile to trigger SMP execution, we therefore only have to compare N to the threshold $N^* = \frac{P}{P-1} \cdot \frac{\omega}{\psi_c}$.

To assess this model, we measured ω once and for all and ψ_c for various basic operations (addition, division, cosinus, ...) and derived a *theoretical* value for N^* (N^*_{theor}) for these operations. Since we do not want the model parameters to depend on cache effects, we performed the measure of ψ_c on a data set whose size was made to fit into the L1 cache of the processor. We then obtained an *experimental* value of N^* (N^*_{exp}) by just running an SMP version of the code and observing when the speed-up got above 1. Results are summarized in table 2, where δ is the relative error between the theoretically and experimentally determined value of the N^* threshold on a dual processor PowerPC G5 ($P = 2$) on which ω has been evaluated to 366000 cycles.

ψ_c	N^*_{theor}	N^*_{exp}	δ
Addition			
0.03	24400000	23040000	5.9%
Division			
7	104572	99328	5.2%
Cosinus			
124.74	5869	5632	4.2%

Table 2. Comparison of experimental results with our prediction model.

Despite the very simple nature of the model, threshold values are estimated within a 6% error margin, which is fairly acceptable. Moreover, this predicted

threshold is always *above* the real one, meaning that the SMP parallelization will always be triggered when the resulting speed-up is greater than one. This overestimation is due to the fact that we purposely don't take into account the way compilers may reschedule or optimize instructions within our loop nests.

3.2 Meta-programming the Parallelization Heuristic

To integrate the analytical performance model to the NT2 library, we have

1. to compute, at compile time, ψ_c for any expression, and N^*;
2. to generate sequential or SMP residual code depending on the actual value of N.

The first step is performed by decorating the abstract syntax tree generated by the expression templates with information relative to the cost of operator nodes, so that the total cost ψ_c of an expression can be computed by accumulating costs of basic operations during a simple tree traversal. In practice, the values of ψ_c for every function supported by the library are evaluated and stored into a header file generated by a separate application run off-line. This application proceeds as follow:

- ω is evaluated by timing a group of P threads performing no computation;
- For each operation, the associated ψ_c is evaluated by benchmarking it for each supported numeric types (e.g. char,short,long,float, etc...) on a data block whose size is computed to fit in the CPU L1 cache. This ensures that all estimated ψ_c values are indeed independent of N.
- A header file containing the value of ω and, for each basic operation, a header file containing a structure encoding the value of ψ_c for each supported type [2]. Those constants, for precision purpose, are stored in hundredths. For example, here is a excerpt of the header associated to the `cos` function specialized for double precision values (whith $\psi_c = 124.74$):

```
template<> struct cost<Cos, double> : int_<12474> {};
```

The second step is performed when expressions are actually evaluated by the overloaded `operator=` of the `matrix` class, as illustrated in the following listing. The various static values needed to decide if parallelization is worth to be triggered are gathered at lines 4-6. A dynamic test is then performed (line 8). As all the required values are static, a single integer comparison is performed at run-time. This test either starts a thread group (line 9) or use a single thread loop (lines 11) to evaluate the expression. The `thread` template class performs boundaries computation, spawns the threads and takes care of threads synchronization using the BOOST::Thread encapsulation of PTHREAD.

[2] Technically, these constants are encoded as BOOST::MPL[14, 15] static integral constants.

```
template<class T> template<class U>
matrix<T>& operator=( const node<U>& tree )
{
const size_t proc = config::proc::value;           // Nbr of PEs
const size_t num  = proc*config::omega::value;     // P*omega
const_size_t den  = (proc-1)*node<U>::cost::value; // (P-1)*psi_c

if( tree.size()*den > num )        // Check if N > P*omega/(P-1)*psi_c
  thread<proc>::Eval(tree, this);
else
  for(int i=0;i<size();i++)  data[i] = tree.eval(i);
return *this;
}
```

4 Experimental Results

Experimental results for this implementation are given below. We measured the speed-up for SMP implementation of two applications of increasing complexity (the term ψ_c reflects this complexity): image difference and a trigonometric computation involving cos and exponential. The target platforms are:

- a 2x2.5GHz Mac Book Pro with 1 Gb of memory running MAC OS X 10.5;
- a 4x2.4GHz Intel Quad Core Q6600 with 4Gb of memory running Windows XP.

Image difference is performed on 8 bits array. The associated code is:

```
delta = abs(im1 - im2)
```

For this code, our performance predictor evaluates that $\psi_c = 4.75$, $N^*_{dual} = 154106$, $N^*_{quad} = 298443$. Experimental results are given in table 3 in which the rows *Naive speed-up* and *NT2 speed-up* respectively give the the speed-up – compared to single threaded C code – obtained with a hand-coded C multi-threaded version of the application and with the NT2 version of the same application.

N	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Dual Core							
Naive speed-up	0.003	0.01	0.05	0.19	0.59	1.26	1.74
NT2 speed-up	1.00	1.00	1.00	1.00	1.00	1.24	1.72
Quad Core							
Naive speed-up	0.0011	0.0046	0.02	0.07	0.27	0.91	2.16
NT2 speed-up	1.00	1.00	1.00	1.01	1.00	1.01	2.14

Table 3. Speed-up benchmark for the image diffence application

For the second application, the associated code is:

```
val = cos(z) - 0.5*( exp(i()*z) + exp(-i()*z) ) )
```

For this code, our performance predictor evaluates that $\psi_c = 660.02$, $N^*_{dual} = 1110$, $N^*_{quad} = 2149$. Experimental results are given in table 4.

N	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Dual Core							
Naive speed-up	0.37	0.96	1.57	1.87	1.97	1.98	1.99
NT2 speed-up	1.01	1.01	1.55	1.85	1.96	1.98	1.99
Quad Core							
Naive speed-up	0.15	0.55	1.55	2.87	3.64	3.90	3.98
NT2 speed-up	1.01	1.00	1.50	2.83	3.62	3.88	3.98

Table 4. Speed-up benchmark for the trigonometric application

4.1 Discussion

Those results demonstrate the following points:

- The model experimentally scales well with the number of **PE**s;
- The estimated N^* value is correct even for complex expressions;
- For the first application, speed-up is only obtained for large values of N, because of the low ψ_c value. This is detected by NT2, which correctly inhibits SMP parallelization when the actual N value is below this threshold;
- When NT2 triggers SMP parallelization, the measured speed-up is within a 5% margin of the hand-crafted code speed-up.

5 Conclusion

In this paper we introduced the need to provide a SMP-aware scientific computing library. We presented NT2 as a solution to the problem of efficient scientific computing in C++ and exposed the technical challenges to overcome when trying to provide a proper SMP parallelization process for such a library.

Our solution was to define a **performance model for SMP computations** that is able to predict if an expression is worth parallelizing. Then, we proposed an SMP-aware implementation of NT2, taking advantage of its inner meta-programmed core to **statically detect expressions to parallelize**. Experimental results showed that our model, despite its simplicity, was precise enough to trigger parallelization only when needed and provide a significant speed-up for various computation kernels on various multi-core architecture.

Work in progress includes fine tuning the prediction model to target emergent many-core architectures like the IBM/SONY/TOSHIBA CELL processor. Future work could target the TILERA TILE64 or the upcoming Intel Polaris 80. Regarding the cost model, an important issue would be to extend it to deal with situations where complexity depends non-linearly on the data size N. Moreover, in the case of many-core architectures, it can be worth to use only a subset of the available cores for NT2 computations (as several concurrent applications can run on the platform). In this case, a challenging question is whether the cost

model can be adapted to predict an optimal size for this subset. Finally, and a on more longer term, we are contemplating the possibility of targeting distributed memory architectures, by providing a message-based implementation model for NT2. Our ultimate goal would be the automatic parallelization of linear algebra code on heterogeneous architectures starting from a single NT2 source, adapted from a MATLAB application.

References

1. Team, T.B.: An overview of the bluegene/l supercomputer. In: Proccedings of ACM Supercomputing Conference. (2002)
2. Falcou, J., Sérot, J., Chateau, T., Jurie, F.: A parallel implementation of a 3d reconstruction algorithm for real-time vision. In: PARCO 2005 - ParCo,Parallel Computing, Malaga, Spain (September 2005)
3. Kalla, R., Sinharoy, B., Tendler, J.M.: Ibm power5 chip: A dual-core multithreaded processor. IEEE Micro **24**(2) (2004) 40–47
4. Kahle, J.: The cell processor architecture. In: MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2005) 3
5. Gepner, P., Kowalik, M.F.: Multi-core processors: New way to achieve high system performance. In: PARELEC '06: Proceedings of the international symposium on Parallel Computing in Electrical Engineering, Washington, DC, USA, IEEE Computer Society (2006) 9–13
6. El-Ghazawi, T., Smith, L.: Upc: unified parallel c. In: SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM (2006) 27
7. Bischof, H., Gorlatch, S., Leshchinskiy, R.: DatTeL: A data-parallel C++ template library. Parallel Processing Letters **13**(3) (September 2003) 461–482
8. Clark, D.: Openmp: A parallel standard for the masses. IEEE Concurrency **6**(1) (1998) 10–12
9. Falcou, J., Sérot, J.: E.V.E., An Object Oriented SIMD Library. Scalable Computing: Practice and Experience **6**(4) (December 2005) 31–41
10. Veldhuizen, T.L.: Expression templates. C++ Report **7**(5) (1995) 26–31
11. Veldhuizen, T.L., Jernigan, M.E.: Will C++ be faster than Fortran? In: Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97). Lecture Notes in Computer Science, Springer-Verlag (1997)
12. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. AW C++ in Depth Series. Addison Wesley (January 2001)
13. Myers, N.: A new and useful template technique: traits. C++ gems **1** (1996) 451–457
14. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)
15. Gregor, D., al.: The boost c++ library. http://boost.org/ (2003)