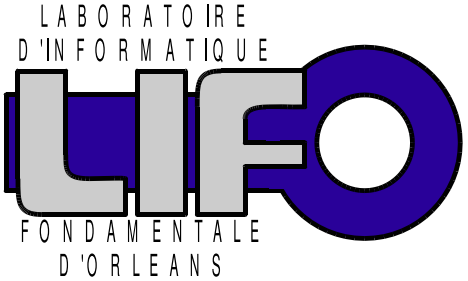


4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Parallel Programming with Orléans Skeleton Library

Noman Javed
Frédéric Loulergue

Rapport n^o **RR-2011-05**

Parallel Programming with Orléans Skeleton Library

Noman Javed LIFO, University of Orléans, France
Noman.Javed@univ-orleans.fr

Frédéric Loulergue
LIFO, Université d'Orléans, France
Frederic.Loulergue@univ-orleans.fr

March 2011

Abstract

Orléans Skeleton Library (OSL) is a library of parallel algorithmic skeletons in C++ on top of MPI. It provides a structured approach towards parallel programming. Skeletons in OSL are based over the bulk synchronous parallelism model. Applications can be developed using different combinations and compositions of the skeletons. This paper illustrates the expressivity of OSL with two applications: a two dimensional heat diffusion simulation, and an exact n-body simulation. Experiments using these applications are performed on parallel machines.

Contents

1 Introduction

Parallel programming is gaining the attention of the developers, specially with the emergence of the new parallel hardware architectures. But the majority of the sequential developers are not equipped with the tactics of parallel programming. The result is the under utilisation of the available resources.

Several approaches are put forwarded as a solution to this problem. Some of them like MPI and POSIX threads can be called as the assembly language of the parallel programming. To program in this way programmer should have the knowledge of the low level details of the architecture and parallelism. On the other hand the approaches like automatic parallelisation hide everything from the programmer. Programmer is unable to apply the domain specific optimisation in this case. This abstraction results in performance loss.

To ease the complexity of parallel program development without sacrificing performance more structured approaches are needed. Parallel design patterns [23] or parallel algorithmic skeletons [9, 22] are one of the structured frameworks like their counterpart design patterns in sequential programming. A number of libraries are there for a while based on this methodology. Bulk synchronous parallelism [29] is one of the well known parallel programming model. The advantage of the BSP over other abstract model of parallel computation such as PRAM is that it takes correct account of the communication and the synchronisation.

The Orléans Skeleton Library (or OSL) mixes the structured model of algorithmic skeletons with bulk synchronous parallelism. It uses expression templates and metaprogramming techniques to ease program development without compromising efficiency. In this paper we present the new version of OSL as well as applications cases. These applications are: a two dimensional heat diffusion simulation, and a N-body simulation. The extensibility of the framework by adding a new skeleton developed by using the existing ones is shown and used in the N-body application.

Next section deals with the details of the structured parallelism approaches on which OSL is based. In section 3 we present the an overview of the current version of OSL. Section 4 presents the various applications and the experiments performed with these applications are detailed in section5. Related work is presented in section 6. Finally we conclude and give perspectives in section 7.

2 Structured Parallelism

Our methodology is to rely on a *structured model of parallelism*. Putting constraints on the parallelism of programs have the following benefits:

- The global view, provided by the structured parallel model of algorithmic skeletons, offers new opportunities *to optimise parallel programs*.
- A theory of program calculation could be designed in order to provide a sound basis for a *methodology of systematic development of correct parallel programs*, as well as supporting tools.

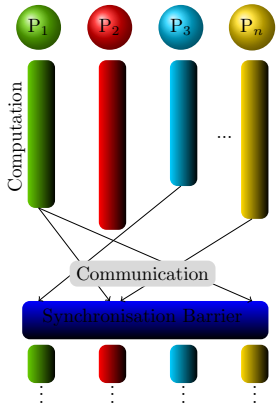


Figure 1: A BSP Super-step

- Restricting the parallelism is also a mean *to reduce the semantical complexity* of parallel programs. This eases the programming but also the performance predictability of programs as well as the formal verification of programs, making them reliable.

Algorithmic skeletons [9, 12, 22] are a form of structured parallelism. Skeletons belong to a finite set of higher-order functions or patterns that can be run in parallel. Usually the programming semantics of the skeletons is similar to the functional semantics of a corresponding sequential pattern (for example the application of a function to all the elements of a collection) and the execution semantics remains implicit or informal. Thus the skeletons abstract the communication and synchronisation details of parallel activities. To write a parallel program, users have to combine and compose the existing skeletons.

Skeletons are not in general any parallel operations, but try to capture the essence of well-known techniques of parallel programming such as parallel pipeline, master-slave algorithms, the application of a function to distributed collections, parallel reduction, *etc.*

The Bulk Synchronous Parallelism is another model of structure parallelism for general-purpose, architecture independent parallel programming. The BSP architecture model consists of three components, namely a set of processors each with a local memory, a communication network, and a mechanism for globally synchronising the processors. Although the BSP architecture model is a distributed memory architecture, it can be mapped to any general purpose parallel architecture.

A BSP program is a sequence of *super-steps*. Each super-step 1 proceeds in three phases: first processors may operate only on values stored in local memory, then values are exchanged between processors through the communication network, these sent values are guaranteed to arrive at the end of a super-step after a global synchronisation occurs. This structured parallelism allows a simple yet realistic performance model.

The performance of a BSP machine is characterised by 3 parameters: p is the number of processor-memory pairs, L (in flop) is the time required for a global synchronisation and g (in flop/word) is the time for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time $g \times h$ for any arity h. The BSP parameters can be determined in practice using benchmarks.

The execution time (or *cost*) of a super-step is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronisation time.

$$\max_{0 \leq i < \text{bsp\_p}} w_i + \max_{0 \leq i < \text{bsp\_p}} \max(h_i^+, h_i^-) \times g + L$$

where, at processor i, w_i is the local sequential work performed during the computation phase, h_i^+ is the size of data sent from i to other processors, and h_i^- the size of the received data by processor i from other processors.

BSP algorithms have been designed and implemented to solve a broad variety of problems: scientific computation [4], artificial intelligence [6, 13, 24], parallel databases [1], *etc.*

The Orléans Skeleton Library combines the advantages of skeletal parallelism and bulk synchronous parallelism by providing high-level abstractions to the programmer, with predictable performances.

3 Orléans Skeleton Library

Orléans Skeleton Library is the library of data parallel algorithmic skeletons on distributed vectors. It is implemented in C++ currently on top of MPI. C++ templates are heavily used for the implementation of the OSL for taking advantage of the functional programming paradigm. The goal of the skeleton approach is to provide a small set of basic parallel patterns. The applications are developed by finding the appropriate combination and composition of these skeletons.

3.1 An Overview of OSL

In the BSP model, the number of processors is fixed during execution. This value is accessible to the programmer, together with the other BSP parameters respectively by osl :: bsp_p, osl :: bsp_g, osl :: bsp_l, and osl :: bsp_r which is a measure of the processors computing power. All parameters, but bsp_p, are obtained by a benchmark program called `oslprobe`.

The data structure used at the base of OSL is distributed array. The data is distributed among the processors at the time of the creation of the array. DArray is implemented as a template class. Thus a variable of type DArray<T> is a distributed array with elements of type T. As there are bsp_p processors in a BSP machine, a distributed array consists of bsp_p partitions evenly distributed (the partitions on the processors with low processor identifiers may have one more element than the processors with high processors identifiers).

Figure 2, gives the informal notations for distributed arrays and an informal semantics for the OSL skeletons used in the applications of section 4. In this figure, bsp_p is noted p.

A distributed array can be seen as a usual array. map (resp. zip) is the usual combinator to apply a function to each element of a distributed array (resp. of two distributed arrays). The first argument of both map and zip could be either a pointer function, or a functor either extending std :: unary_function or std :: binary_function . It is thus possible to use BOOST binders for providing partially applied functions.

Type / Signature	Notation / Informal semantics
DArray<T> (sequential view)	$[t_0, \ldots, t_{t.size-1}]$
DArray<W> map(W f(T), DArray<T> t)	map$(f, [t_0, \ldots, t_{t.size-1}]) = [f(t_0), \ldots, f(t_{t.size-1})]$
DArray<W> zip(W f(T,U), DArray<T> t, DArray<U> u)	zip$(f, [t_0, \ldots, t_{t.size-1}], [u_0, \ldots, u_{t.size-1}]) = [f(t_0, u_0), \ldots, f(t_{t.size-1}, u_{t.size-1})]$
<T> reduce(T⊕(T,T), DArray<T> t)	reduce$(\oplus, [t_0, \ldots, t_{t.size-1}]) = t_0 \oplus t_1 \oplus \ldots \oplus t_{t.size-1}$
DArray<Vector<T> > getPartition(DArray<T> t)	getPartition $([t_0, \ldots, t_{t.size-1}]) = \langle\, [t_0^0, \ldots, t_{l_0-1}^0], \ldots, [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}] \,\rangle$
DArray<T> flatten(DArray<Vector<T> > t)	flatten $(\langle\, [t_0^0, \ldots, t_{l_0-1}^0], \ldots, [t_0^{p-1}, \ldots, t_{l_{p-1}-1}^{p-1}] \,\rangle) = [t_0, \ldots, t_{t.size-1}]$
DArray<T> permute(**int** f(**int**), DArray<T> t)	permute$(f, [t_0^0, \ldots, t_{l_0}^0]) = [t_{f(0)}^0, \ldots, t_{f(l_0-1)}^0]$
DArray<T> shift(**int** dec, T f(T), DArray<T> t)	shift $(d, f, [t_0^0, \ldots, t_{l_0}^0]) = [f(0), \ldots, f(d-1), t_0, \ldots, t_{t.size-1-d}]$

Figure 2: OSL Data Structure and Skeletons

getPartition exposes the partitioning of a distributed array, transforming a distributed array of type DArray<T> into a distributed array of type DArray<Vector<T> > containing one vector by processor. The inverse operation of getPartition is flatten . flatten is optimised in such as way that communications will occur (to obtain a evenly distributed array) only if it is necessary.

reduce is a parallel reduction with a binary *associative* operator $\oplus$. Communications are needed to execute a reduce. permute and shift are communication functions. permute moves the content of the distributed array, hence redistributes the array, according to a permutation function f on the interval $[0, t.size-1]$. shift is used to shift elements on the right (the case shown in the figure) or the left depending on the sign of its first argument. The missing values, at the beginning or the end of the array, are given by function f.

3.2 OSL Implementation

OSL is implemented in C++. It uses MPI for communication and synchronisation. OSL makes use of the templates, operator overloading and some meta-programming techniques. As the applications are developed by composing skeletons, efficient composition of skeletons is an important point of optimisation. It is achieved in OSL by using the technique of expression templates in C++ [30].

Expression Templates are a programming technique for implementing an efficient operator overloading within C++. It allow intelligent removal of temporaries and enable lazy evaluation. The basic idea of expression templates is to inline the expressions in the function body. Operator overloading technique is used to encode the expression in the form of the nested template classes at compile time. Then the nested expression objects become evaluated by one loop iteration. Optimising the code via the inlining, the resulted program becomes nearly as efficient as the corresponding C code. Some vector and matrix based linear algebra libraries benefits a lot from this technique.

In parallel programming, processors often need to communicate their data with the other processors. Sometimes the data (user defined classes) contain indirections and need to be serialised before being communicated. C++ doesn't support serialisation transparently. One solution is that the programmer provides the serialisation method for its classes. This increases the complexity of the program development from the user's point of view. Some serialisation libraries like TPO [16] or BOOST serialisation library [19] can make this task easier for the developer. We choose the Boost serialisation library and use Boost MPI for implementing OSL. The reason behind using Boost is that it is the most active set of libraries which is continuously under research and many of the boost

$$u(x, y, t + \Delta t) =$$
$$\frac{\gamma \Delta t}{\Delta s^2}\Big(u(x+\Delta s, y, t)+u(x-\Delta s, y, t)+u(x, y+\Delta s, t)+u(x, y-\Delta s, t)-4u(x, y, t)\Big)+u(x, y, t)$$

Figure 3: Heat Equation in 2D

libraries are already accepted by the draft technical report on C++ library extensions [28] (also called TR1), and other libraries will be included in the C++0x standard and are proposed for TR2.

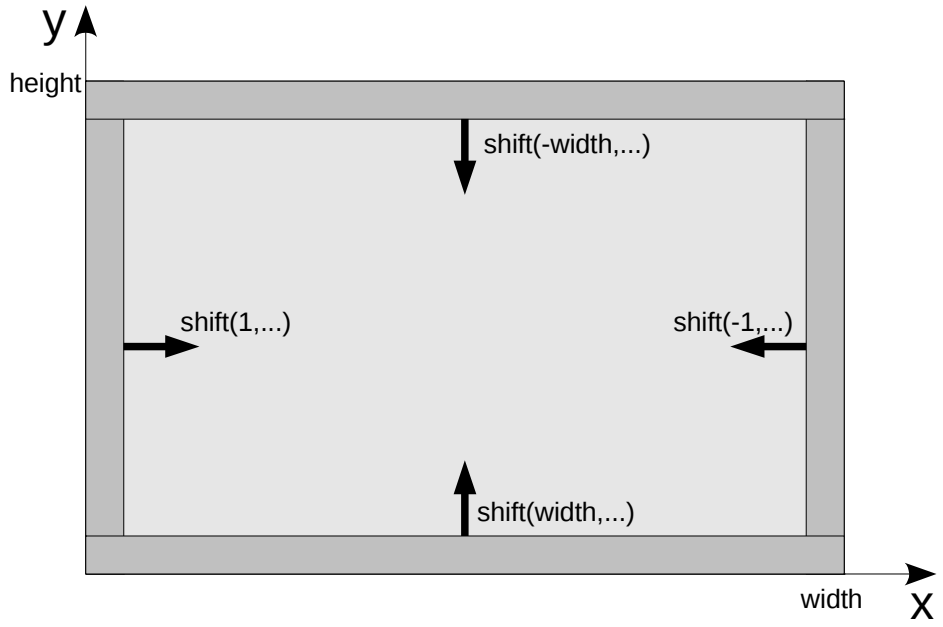
4 Applications

4.1 2D Heat Equation

Heat Equation is a parabolic second order differential equation which models the flow/propagation of heat across some region over time. The heat at particular point in a time step can be calculated by accessing the values of the neighbouring elements. The formulation of heat equation in two dimensions is captured by the equation of figure 4.1. We also have boundary conditions, in the following we assume that left and right (resp. bottom and top) boundary values are functions depending on y (resp. x).

The two dimensional region is parametrised by the width and the height parameters. It is represented by a distributed array of length (width × height). The systematic development skeleton based algorithm of the two dimensional heat equation is presented below. The skeletons are extracted from the equation in figure 3:

- Access to the neighbouring elements, as shown in the following figure, are done using the shift skeleton:



  – Left and right neighbouring elements are accessed using the shift skeleton

```
plate = Skel::zip(std::plus<double>(),
          plate,
          Skel::map(boost::bind(std::multiplies<double>(), ( diffuse * dt ) / (ds * ds), _1),
                Skel::zip(std::plus<double>(),
                      Skel::map(boost::bind(std::multiplies<double>(), -4, _1), plate ),
                      Skel::zip(std::plus<double>(),
                            Skel::mapIndex(rightBound, Skel::shift_right (rightBound,plate )),
                            Skel::zip(std::plus<double>(),
                                  Skel::mapIndex(leftBound, Skel::shift_left (leftBound, plate )),
                                  Skel::zip(std::plus<double>(),
                                        Skel::shift (-width, topBound, plate),
                                        Skel::shift ( width, bottomBound, plate)
                                  )
                            )
                      )
                )
          )
);
```

Figure 4: One Step of Heat Diffusion Simulation in OSL

when the offset is 1 or -1, and the replacement value is a single value rather than a function or functor. However in this two dimensional case the shifting is incorrect: we have to use mapIndex to replace the values at left and right boundaries by the one computed by the left and right boundary functions.

– To access the top and bottom neighbouring elements, the array should be shifted width times. This operation is done using again the shift skeleton.

- The multiplication of the element by the diffusivity and -4 is captured by the map skeleton
- The addition of the neighbouring elements and the final addition operation is performed by the zip skeleton.

The code snippet for one step of simulation of heat diffusion is presented in figure 3 As visible in the listing all the skeleton operations are composed in a single expression. The composition of the skeletons in this manner triggers the expression templates implementation of the skeletons which results in optimised performance.

4.2 N-Body Simulation

The N-Body simulation is an application that simulate the motion of n point masses interacting under the presence of gravitational force. Assuming a set of n point with mass m_i, position $\vec{p_i}(t)$, and velocity $v_i(t)$ as continuous functions with respect to time t, with $0 \leq i < n$, the evolution of the system is described by a set of differential equations

which could be transform into the following discrete form with time step Δt:

$$\begin{cases} \vec{p_i}(t+\Delta t) &= \vec{p_i}(t) + \vec{v_i}(t) \times \Delta t \\ \vec{v_i}(t+\Delta t) &= \vec{v_i}(t) + \dfrac{\vec{F_i}(t)}{m_i} \times \Delta t \\ \vec{F_i}(t) &= \gamma \times \sum_{j \neq i} \dfrac{m_i \times m_j \times (\vec{p_j}(t) - \vec{p_i}(t))}{|\vec{p_j}(t) - \vec{p_i}(t)|^3} \end{cases}$$

where γ is the gravitational constant.

Thus, the problem of calculating the sum of the forces for all the particles is of the order n^2.

In our OSL implementation, the particles are partitioned among the processors, with $\frac{n}{p}$ particles per processor. To calculate the sum of the forces, each processor communicates its particles with the others. This can be achieved in two ways:

- By using an all to all communication, so that every processor gets the current positions and velocities of all the other particles, to be able to update the positions and velocities of the particles it owns;
- By using a systolic loop, i.e in $p-1$ steps (in a BSP setting, super-steps), where at each step, each processor:
  - Computes the forces applied to the particles it owns all the time by $\frac{n}{p}$ other particles it owns at the current step (at the first step the two sets of particles are equal);
  - Receives $\frac{n}{p}$ new particles from its left neighbour, and sends the $\frac{n}{p}$ particles it was owning on the current step to its right neighbour.

Both versions have the following BSP computational cost for one step of the simulation:

$$\mathcal{O}(\frac{n^2}{p})$$

In the first version each processor receives $\frac{n}{p} \times (p-1)$ particles, thus the communication and synchronisation BSP cost is:

$$\mathcal{O}(\frac{n}{p} \times (p-1) \times g) + L$$

In the systolic version, at each step, each processor sends and receives $\frac{n}{p}$ particles, and there are $p-1$ steps. Therefore the communication and synchronisation BSP cost is:

$$\mathcal{O}((p-1) \times (\frac{n}{p} \times g) + L)$$

The total exchange version would thus have a better performance. However the advantage of the systolic version is that it allows a much smaller memory consumption than the total exchange version. In the next sections we proceed with the systolic version, its generalisation and the N-body problem specific optimisation.

4.2.1 Extracting skeletons out of sequential algorithm

In this section we present how to extract a skeletal implementation out of the generic description of the algorithm. Following are the main operations along with their skeleton implementation for nbody simulation:

- make a copy B of the original particles A,
  B = A;
- for each A_i compute interactions with all B_j. These interactions are captured by rotating the copied partition implemented in terms of circular shift right.
  for(i=0; i < A. get_local_size (); ++i) shift_rcl (**true**,B); To avoid the self computation of the force a boolean is used in shift_rcl
- calculate force between two particles A_i and B_j,
  zip(calcF, A, B);
- sum all the forces applied to a particle B_i,
  zip(sumF, force, old_force);
- update the positions and the velocities of the particles.
  zip(move, A, force);

The steps 2–4 can be optimised by composing them in the following way:

```
for(int i=0; i < A. get_local_size (); ++i)
   zip(sumF, force, zip(calcF, A, shift_rcl (true,B)));
```

The parallel version can be developed in the same way just by replacing the circular shift operation by a permute_partition to shift the whole partition. A permute_partition can be implemented in terms of the getPartition , permute and flatten skeletons.

4.2.2 The systolic version

Two systolic loops are used to compute the sum of the forces on each particle. The outer loop models the partition level force computation while the inner loop represents the force computation between the processor's local particles and the received particles. Thus, the outer loop executes $p-1$ times while the inner loop executes local size -1 times. In each iteration of the inner loop three operations are performed:

- shifting the local particles circularly towards right: a circular shift right function is written for this purpose,
- computing the force between the two particles by using zip skeleton,
- add the newly computed force to the previous one by a zip skeleton.

A partial listing for calculating the force is presented in figure 5. Once the force for each particle is calculated a zip skeleton using a movement functor can be used to update the positions and velocities of all the particles.

```
totalForce = computeLocalForce(particles,tmp,true); // true: avoids self computation of force
for(int i = 1; i < bsp_p-1; ++i) {
  // sends partition to the right hand neighbour
   tmp = Skel::permute_partition (boost::bind(permuteRight(),i,_1),tmp);
   // computes local force as mentioned in section IV.B.1
   localForce = computeLocalForce(particles,tmp,false);
   totalForce = Skel::zip(sumF,localForce, totalForce );
}
// calculates new positions and velocities
particles = Skel::zip(boost::bind(movement(),dt,_1,_2), particles , totalForce );
```

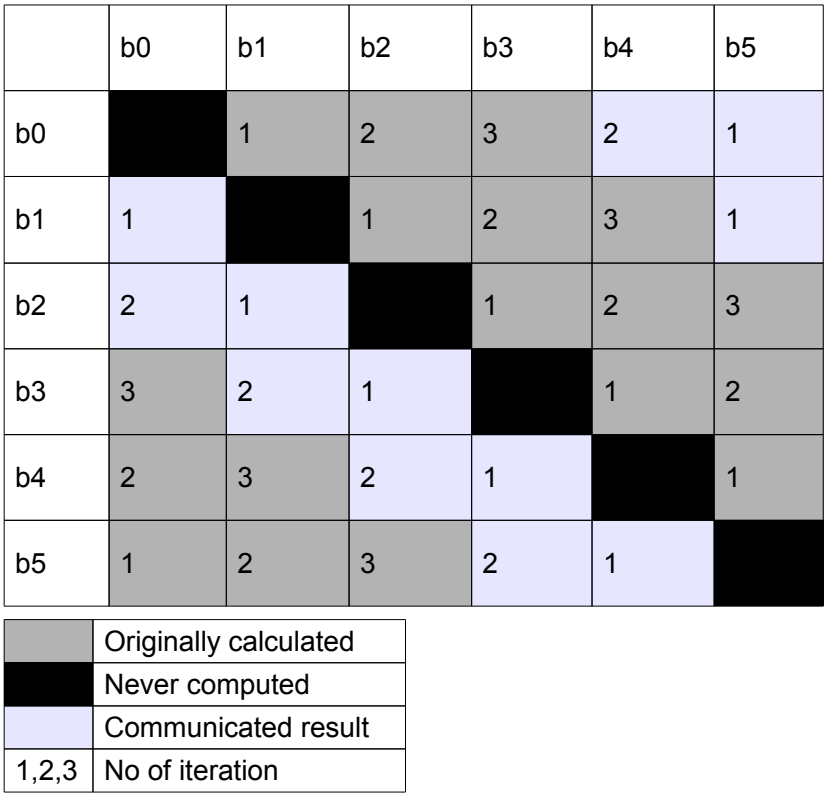
Figure 5: N-Body Simulation: Code Excerpt

4.2.3 The RaMP skeleton

Darlington [12] used RaMP (Reduce and Map over Pairs) for the computation of the sum of the forces for each particle. This is actually a generalisation of the systolic version. As this pattern may occur often in parallel scientific application, we added the new skeleton RaMP to the OSL, but not as a primitive skeleton but as a user-defined skeleton. The RaMP skeleton requires a reduction operator, a binary function, and the two expressions. The binary function is used to calculate the interaction between the two expressions. The result is than reduced by the reduction operator. As in the case of nbody the binary function is calcF to calculate the force between the particles and the reduction operator is sumF for summing up the forces. The body of the RaMP **operator**() is same as the one presented in systolic loop.

4.2.4 Some optimisations

As the force "applied" by the body i to the body j is the opposite of the force applied by the body j to the body i, it is of course possible to avoid computing these two forces. We did so by adding circular shift left function at the inner level and permute_partition at the outer level. The newly computed force by body i is sent back to the body j:



This reduces the inner loop iterations to $\frac{\text{local\_size}}{2}$ and the outer loop iterations to the $\frac{p}{2}$. Experimental results comparing this version with the RaMP version are presented in section 5.

5 Experiments

All the experiments were conducted on a cluster of PC with 8 nodes. Each of the 8 nodes contains two Quad-Core AMD Opteron 2376 processors with a 2.3 GHz frequency. Each node has 16 Gb of memory. These nodes are linked by a Gigabit-Ethernet network, each node having one network card. The operating system is Ubuntu 10.04. The MPI library used was Open MPI 1.4.2. The compiler was GCC 4.3. All the examples were compiled using the third level of optimization.

Heat Diffusivity Simulation The experiments were conducted for scalability and for different sizes. The program heat equation takes the width and number of iterations as the parameters. The total length of the DArray is $width \times width$. Figure 6 presents the scalability of the heat equation by keeping the width parameter fix at 1000 and varying the processors from 1 to 64. Figure 7 presents the timings of the heat equation for different sizes of the plates for 64 processors.

***N*-body Simulation** The experiments show scalability and the comparision between the various versions were conducted. The progam takes the problem size, time of simulation and single time step as the input parameters. Figure 8 presents the graph that scales quadratically with increasing number of bodies, while keeping the number of processors

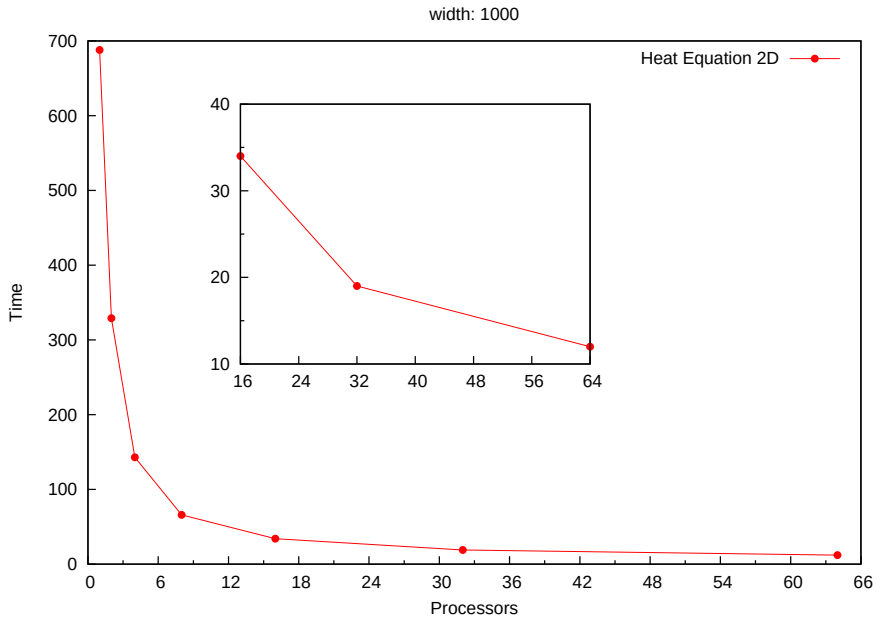


Figure 6: Increasing number of processors, 1000 width

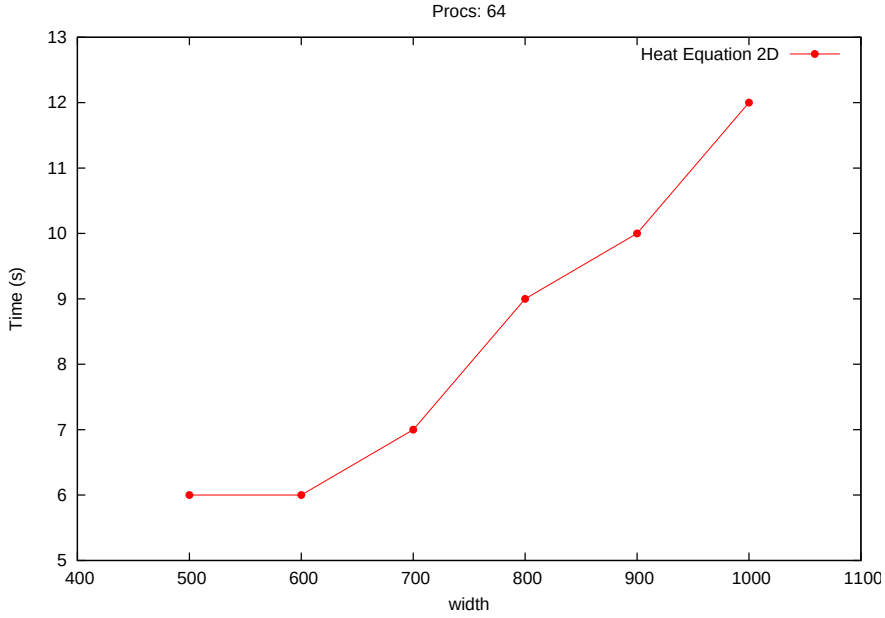


Figure 7: Increasing plate sizes, 64 processors

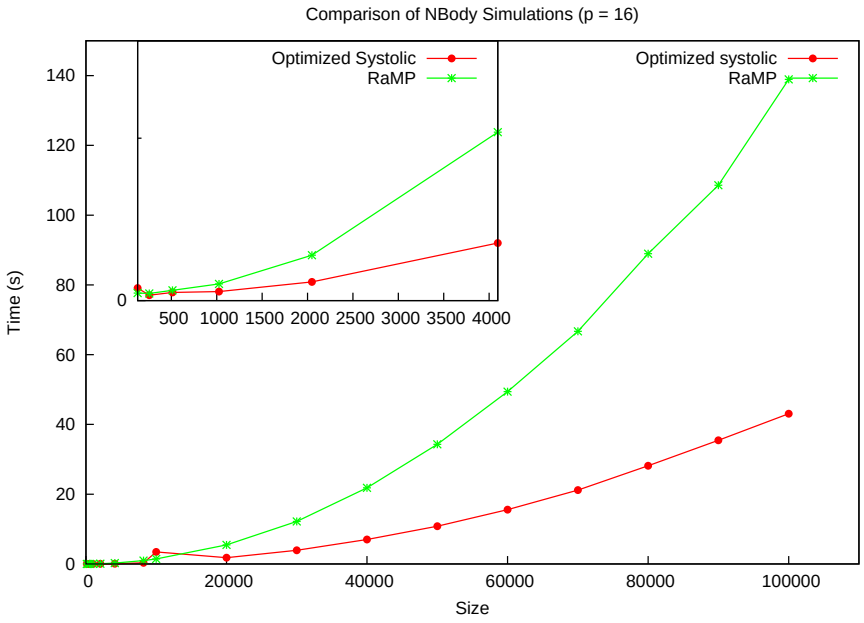


Figure 8: Comparison Optimized Systolic and RaMP, 16 processors

constant for 16 processors. Non serialized systolic version is used for this test. The optimized systolic version as shown in figure 8 is more efficient than the RaMP version. In RaMP version we have to calculate all the interactions while in the optimized systolic we tansfer the interactions already calculated thus reducing half of the calculations. The performance scales by increasing the number of processors as demonstrated in figure 9 by keeping the problem size constant and increasing the number of processors.

6 Related Work

Several parallel skeleton libraries have been implemented. Since the idea of skeletons has close relationship with functional programming, there are several implementations based on functional languages such as Template Haskell [15], SML [25] and Objective Caml [11].

Libraries based on C or C++ aim at being efficient and more widely used than the previous libraries:

- Muesli [8] and SkeTo [21, 20] share with OSL a number of classical data-parallel skeletons on distributed arrays. Our previous work showed that the previous version of OSL was more efficient than both of these libraries [18].

- Quaff [14] is highly optimised with meta-programming techniques. However to attain its very good performances, some constraints are put on the algorithmic structure of the skeletons (that are mainly task parallel skeletons).

- eSkel [10, 2] is designed to ease integration of algorithmic skeletons programs within MPI code and to avoid to put to many constraints on the user. Being closer to MPI, the signatures of eSkel's skeletons are more complicated than the signatures of other skeletons libraries.

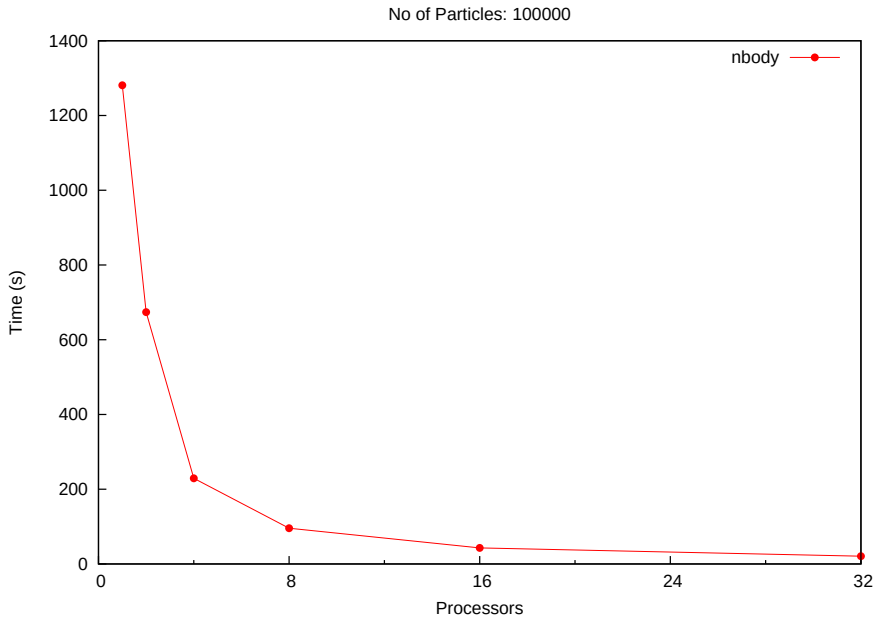


Figure 9: Increasing number of processors, 100K particles

- DatTel [3] is a partial parallel implementation of the STL library (for shared memory machines). It is also the design principle of STAPL [7, 27] which targets both shared and distributed memory machines.

It is of course possible to write BSP programs without skeletons. The set of functions of the proposed BSPlib standard is quite small compared to MPI, yet efficient. The programming style of these libraries is not as high level than skeleton libraries and OSL. There are three libraries sharing the BSPlib specification: BSPlib [17], PUB [5] and BSPonMPI [26].

7 Conclusion and Future Work

OSL is the library of the data parallel skeletons based on the top of Bulk synchronous parallelism. The development of a 2D heat equation simulation and an N-body simulation with OSL demonstrates the capability of the OSL to develop scientific applications. Several versions of the problem are developed, showing the flexibility of parallel program development with OSL. The extensibility of OSL is presented by adding a new skeleton by combining the existing ones.

In future work we will consider the support of performance portability of the skeletons based on the BSP model, the development of new applications including parallel clustering algorithms, as well as work on the formal semantics of the programming and execution model of OSL.

References

[1] M. Bamha and M. Exbrayat. Pipelining a Skew-Insensitive Parallel Join Algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003.

[2] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In J. C. Cunha and P. D. Medeiros, editors, *11th International Euro-Par Conference*, LNCS 3648, pages 761–770. Springer, 2005.

[3] H. Bischof, S. Gorlatch, and R. Leschinskiy. DatTeL: A Data-Parallel C++ Template Library. *Parallel Processing Letters*, 13(3):461–472, 2003.

[4] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[5] O. Bonorden, B. Judoiink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29:187–207(2), 2003.

[6] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

[7] A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: standard template adaptive parallel library. In G. Haber, D. D. Silva, and E. L. Miller, editors, *The 3rd Annual Haifa Experimental Systems Conference (SYSTOR 2010)*. ACM, 2010.

[8] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli – A Comprenhensive Overview. Technical Report Working Paper No. 7, European Research Center for Information Systems, University of Münster, Germany, 2009.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989. Available at `http://homepages.inf.ed.ac.uk/mic/Pubs`.

[10] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[11] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis. Skeletal Parallel Programming with OcamlP3l 2.0. *Parallel Processing Letters*, 18(1):149–164, 2008.

[12] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel Programming Using Skeleton Functions. In *PARLE'93*. Springer Verlag, 1993.

[13] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[14] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: Efficient C++ Design for Parallel Skeletons. *Parallel Computing*, 32:604–615, 2006.

[15] K. Hammond, J. Berthold, and R. Loogen. Automatic Skeletons in Template Haskell. *Parallel Processing Letters*, 13(3):413–424, 2003.

[16] P. Heckeler, M. Ritt, J. Behrend, and W. Rosenstiel. Object-Oriented Message-Passing in Heterogeneous Environments. In A. L. Lastovetsky, M. T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI Users' Group Meeting*, volume 5205 of *LNCS 5205*, pages 151–158. Springer, 2008.

[17] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[18] N. Javed and F. Loulergue. OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays. In Y. Don, R. Gruber, and J. Joller, editors, *8th international Conference on Advanced Parallel Processing Technologies (APPT'09)*, LNCS 5737, pages 436–451. Springer, 2009.

[19] P. Kambadur, D. Gregor, A. Lumsdaine, and A. Dharurkar. Modernizing the C++ Interface to MPI. In B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI User's Group Meeting*, LNCS 4192, pages 266–274. Springer, 2006.

[20] K. Matsuzaki and K. Emoto. Implementing Fusion-Equipped Parallel Skeletons by Expression Templates. In *21st International Workshop on Implementation and Application of Functional Languages (IFL)*, LNCS. Springer, 2009.

[21] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *InfoScale'06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press, 2006.

[22] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[23] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

[24] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998.

[25] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming*, 15(4):615–650, 2005.

[26] W. J. Suijlen. BSPonMPI. http://bsponmpi.sourceforge.net.

[27] G. Tanase, X. Xu, A. A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. G. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pList. In G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, editors, *22nd International*

Workshop on Languages and Compilers for Parallel Computing (LPCP 2009), LNCS 5898, pages 16–30. Springer, 2009.

[28] The C++ Standards Committee. Draft Technical Report on C++ Library Extensions. Technical Report 19768, ISO/IEC, 2005.

[29] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990.

[30] T. Veldhuizen. Techniques for Scientific C++. Computer science technical report 542, Indiana University, 2000.