EASYMSG: TOOLS AND TECHNIQUES FOR AN ADAPTIVE OVERLAPPING IN SPMD PROGRAMMING

PASCAL HAVÉ

Abstract. During the development of a parallel solver for Maxwell equations by integral formulations and Fast Multipole Method (FMM), we needed to optimize a critical part including a lot of communications and computations. Generally, many parallel programs need to communicate, but choosing explicitly the way and the instant may decrease the efficiency of the overall program. So, the overlapping of computations and communications may be a way to reduce this drawback. We will see a implementation of this techniques using dynamic and adaptive overlapping based on the EasyMSG high level C++ library over MPI, a case of SPMD programming.

Mathematics Subject Classification. 31B10, 65R20, 65Y05, 68M10, 90B20.

Received: February 21, 2002. Revised: May 6, 2002.

1. Introduction

We study here an implementation with overlapping in a Parallel Fast Multipole Algorithm applied to Maxwell equations [6] solved by integral formulations [2]. This implementation contents a critical part: the transfers.

This article will present adaptive overlapping techniques used for our problem after a brief overview of the mathematical background which leads us to consider these techniques as a true alternative to the classical implementations. These techniques are wrapped into a library EASYMSG, which provides a abstraction of lower communication layer like MPI [10]. The performance given by a such implementation are really significant as we will see in Section 5.

1.1. Mathematical background

Let us consider the Maxwell’s equation in the frequency domain for an electromagnetic monochromatic wave of frequency $\omega$:
\[
\begin{align*}
\text{rot} E - i\omega\mu_0 H &= 0, & \text{in } \Omega_e \\
\text{rot} H + i\omega\epsilon_0 E &= 0, & \text{in } \Omega_e \\
E \wedge n|\Gamma &= -E^{inc} \wedge n, & \text{on } \Gamma \\
\lim_{|r| \to +\infty} |r| \sqrt{\epsilon_0 E - \sqrt{\mu_0} H} & = 0
\end{align*}
\]

where
- \( E \) and \( H \) are the electric and magnetic vector fields;
- \( \Omega_e \) is the complement of an open set of boundary \( \Gamma \) assumed perfectly conductive;
- \( \epsilon_0 \) and \( \mu_0 \) are the dielectric parameters of the medium assumed constant;
- \( n \) is the normal to the surface \( \Gamma \);
- \( i \) is the first square root of \(-1\) in \( \mathbb{C} \).

By a representation theorem [3], we can find the solution to these equations by solving the Electric Field Integral Equation which leads to solve \( Ma = b \) by a Galerkin method, based on the Rao-Wilson-Glisson \{J\} basis functions [4].

\[
M_{i,j} = \int_{\Gamma \times \Gamma} G(x-y) \left( J_i(x) \cdot J_j(y) - \frac{1}{\kappa^2} \text{div}_\Gamma J_i(x) \text{div}_\Gamma J_j(y) \right) d\Gamma(x) d\Gamma(y)
\]

\[
b_i = \frac{i}{\omega \mu_0} \int_{\Gamma} J_i(x) \cdot E_i^{inc}(x) d\Gamma(x)
\]

where \( G \) is the Green kernel defined by

\[
G(r) = \frac{e^{i\kappa|\mathbf{r}|}}{4\pi|r|}, \quad \text{with } \kappa = \omega \sqrt{\epsilon_0 \mu_0}
\]

and \( J_i \) is defined on edge \( i \) at the intersection of two triangles \( T^+ \) and \( T^- \); \( S^+ \) and \( S^- \) are the vertices opposite to the edge \( i \) in \( T^+ \) and \( T^- \); \(|T|\) is the area of a triangle \( T \):

\[
J_i(x) = \begin{cases} 
\frac{1}{2|T^+|}(x - S^+) & \text{if } x \in T^+ \\
\frac{1}{2|T^-|}(x - S^-) & \text{if } x \in T^-.
\end{cases}
\]

These matrices are complex dense matrices and the resolution of the system can be done using an iterative method (eg GMRES, QMR . . . ). The main part of the computation is matrix-vector products with the matrix given by \( G \).

### 1.2. The FMM algorithm

The standard FMM is based on the formula

\[
e^{i\kappa|P+M|}/|P+M| = i\kappa \lim_{\ell \to +\infty} \int_{S^2} e^{i\kappa(s,M)} T_{L,P}(s) ds, \quad \text{for } P, M \text{ vectors of } \mathbb{R}^3
\]

where \( T_{L,P} \) is the transfer function defined for any point \( s \in S^2 \), the unit sphere, by

\[
T_{L,P}(s) = \sum_{m=0}^{L} \frac{(2m+1)^m}{4\pi} h_{m+1}^{(1)}(\kappa|P|) P_m(\cos(s,P))
\]
with $L$ a truncation parameter, the spherical Hankel function $h^{(1)}_m$, $P_m$ Legendre polynomials, and $\langle \cdot, \cdot \rangle$ the usual scalar product.

Let us consider two well separated spheres $S_1$ and $S_2$ (of respective centers $C_1$, $C_2$ and radius $R_1$, $R_2$). For any points $x_1$ inside $S_1$ and $x_2$ inside $S_2$. Let us define $r_0 = C_2 - C_1$ and $r = x_1 - C_1 - C_2 - x_2$, by writing $x_1 - x_2 = r_0 + r$, we obtain

$$\frac{e^{\im \kappa |x_1- x_2|}}{|x_1- x_2|} \sim \im \kappa \lim_{l \to +\infty} \int_{S^2} e^{\im \kappa (s, x_1- C_1)} T_l(s) e^{-\im \kappa (s, x_2- C_2)} ds. \quad (1)$$

Let $N$ be the number of edges on the surface of the perfectly conductive object. The mid edges are the quadrature points. These points are gathered into clusters. Each cluster $P_r$ is bounded by a sphere $S_r$ of center $C_r$ and radius $R_r = \max_{x_i \in P_r} |x_i - C_r|$. We define the well separated clusters by the negation of “$P_r$ close to $P_s \iff |C_r - C_s| \leq \alpha (R_r + R_s)$” with $\alpha > 1$.

We can rewrite the matrix $M$ as the sum of far interactions and close interactions matrices

$$M_{ij}^{\text{far}} = M_{ij} \text{ if } x_i \text{ and } x_j \text{ are not close (i.e far), else 0}$$

$$M_{ij}^{\text{close}} = M_{ij} \text{ if } x_i \text{ and } x_j \text{ are close else 0.}$$

The FMM method is an efficient way to compress the $M^{\text{far}}$ matrix with the formula (1). To make the product $u = Mv$, we compute $u = u^{\text{far}} + u^{\text{close}}$ with $u^{\text{close}} = M^{\text{close}}v$ and $u^{\text{far}} = M^{\text{far}}v$; $M^{\text{close}}$ is computed exactly. The FMM compression of $M^{\text{far}}$ is a way to compute a fast product ($O(N \log N)$) but with an approximation and without storing all its coefficients.

The algorithm is based on a hierarchical subdivision of the space giving a hierarchical computational method of transfers (for reducing the size of the close interactions matrix). An efficiently way to store the points of the surface of the object, with a hierarchical subdivision, is the oct-tree (Fig. 1).

The oct-tree is a unique cell at the level 0 and 8 cells at a level $l > 0$ has one father cell $C$ at the level $l - 1$. We define the cells close to $c$ at level $l$ as neighboring cells at the same level. The far cells from $c$ are the cells which are not close to $c$ but with a father close to the father of $c$: $C$.

Let us consider an oct-tree with 3 levels, named $L_0$, $L_1$, $L_2$. We explain the method for 3 clusters $P_{r_0}$, $P_{r_1}$ and $P_{r_2}$ respectively at the levels $L_0$, $L_1$, $L_2$, such that $P_{r_2} \subset P_{r_1} \subset P_{r_0}$.

The first step is to compute the radiation functions for every cluster $P_l$ at each level.

$$F_l(s) = \sum_{x_j \in P_l} v_j e^{\im \kappa (s, C_l - x_j)}.$$
Then, from the lowest level $L_0$, we compute the transfers (Fig. 2)

$$G_{r_0}(s) = \sum_{P_{r_0} \text{ far from } P_{t_0}} F_{t_0}(s)T_{l,C_{r_0}-C_{t_0}}(s).$$

Then, at the level $L_1$ many transfers are already completed, and we need to perform much less transfers (Fig. 3)

$$G_{r_1}(s) = \sum_{P_{t_1} \text{ far from } P_{r_1}} F_{t_1}(s)T_{l,C_{r_1}-C_{t_1}}(s)$$

where for any $P_{t_i}$ at the level $L_i$, $P_{t_0}$ denotes its father: the only cluster at the level $L_0$ with $P_{t_i} \subset P_{t_0}$.

We do so up to the higher level, in our example $L_2$ (Fig. 4)

$$G_{r_2}(s) = \sum_{P_{t_2} \text{ far from } P_{r_2}} F_{t_2}(s)T_{l,C_{r_2}-C_{t_2}}(s).$$

Then, the last step is to propagate the information from the root (level $L_0$) to the leaves (level $L_2$).

$$\tilde{G}_{r_2}(s) = G_{r_2}(s) + e^{i\kappa(s,C_{r_2}-C_{t_2})}G_{r_0}(s)$$

$$\tilde{G}_{r_1}(s) = G_{r_1}(s) + e^{i\kappa(s,C_{r_1}-C_{t_1})}\tilde{G}_{r_2}(s).$$
Finally, we obtain $u^{far} = M^{far}v$, for any $x_i \in P_{r_2}$, with the formula (see Fig. 5)

$$u^{far}_i = \int_{S^2} e^{i\kappa(s,x_i-C_{r_2})} \tilde{G}_{r_2}(s) ds.$$

### 1.3. Problem statement

A critical part of this algorithm is the “transfers” or computations involving data of “far” cells.

The computational domain is mapped on an oct-tree (Fig. 1), where the data of each cell are used by exactly one process.

A transfer is a computation between two values located on two far cells (see Sect. 1.2 for more explanation). Figure 6 shows the far cells of a reference cell $c$.

However, these transfers (computations) may use cells (data) located on different processes. If the cells are located on the same process we have local transfers, else we have non-local transfers between different processes with migrations by communications.

This step of the Fast Multipole Algorithm may be summarized as matrix-vector products $y^l = M^l x^l$, where we have mapped the cells of level $l$ on a vector $x^l$. Each line of the matrix $M^l$ represents transfers to do:

- $M^l_{i,j} \neq 0$ if the cell mapped on $x_i^l$ is far from $x_j^l$, and by symmetry $M^l_{j,i} \neq 0$;
- $M^l_{i,j} = 0$ otherwise.

Moreover, each value $x_j^l$ of the vector $x^l$ is also a small vector, so each scalar operation must be replaced with a vectorial operation. For clarity, we simplify the computation of the transfer by choosing $M^l_{j,i} = M^l_{i,j} = 1$ or 0.
The matrices are very sparse and we can see that this part of the Fast Multipole Algorithm leads to an important exchange of data (non-local transfers) but we can overlap the communications with many local-transfers. For increasing the overlapping, we will overlap all the matrix-product $M^l x^l$ together.

Every parallel program needs to compute and communicate. In a MIMD\(^1\) approach, we can decompose the program into many parts where each process can compute or communicate when it wants. Moreover, when reaching the low level programming (OS, Hardware), the decomposition and overlapping appear clearly and may be optimized by a better knowledge of the behavior of communications (hardware interruptions, ...). However, for increasing the portability with a higher level of abstraction, we need to understand how the communication works regardless of the low level architecture, even more so in SPMD\(^2\) context.

In SPMD, a parallel program is written as a sequential program running on $N$ processes. Using a message passing library (like MPI-1) in a SPMD context implies choosing when sending and receiving messages along the sequential codes.

**Algorithm 1.1.** (Fig. 7) A ‘bad’ algorithm

```
while Communications needed do
  Start communications (non blocking sending)
  Do $n$ computations (if possible)
  Finish communications (waiting and receiving)
end while
```

Do remaining computations

Algorithm 1.1 uses a predefined parameter $n$ and the communications are uniformly distributed between computations. If $n = 0$ the algorithm does all the communications before the computations; if $n = \infty$, the algorithm does all computations between sending and receiving the data. This simple algorithm allows to embed the data transfers between computations.

The main drawback is the “waiting” for the communications, so each communication time may be variable.

As is well known [8, 9], a better choice is to use asynchronous communications: we can send or receive a message by testing if there is such an incoming or outgoing message, else we do a few computations.

A problem may occur when there are many processes: we need to define a schedule of the incoming and outgoing communications. Furthermore, each communication implies an incompressible time for negotiation; we call it latency time. Then, we must try to aggregate communications for reducing several latency times to only

\(^1\)MIMD stands for “Multiple Instructions, Multiple Data”.
\(^2\)SPMD stands for “Single Program, Multiple Data”.

---

**Figure 6.** A computation world mapped on a quad-tree for the solution of a 2D integral equation on $\Gamma$. Far cells of $c$ are shown with diagonal stripes. Only the gray cells are active for the computation.
One. Aggregating communications needs to know when to send the buffer and how to receive aggregated data. Here comes the EasyMSG library.

Many other libraries exist for distributed scientific computing, with many different objectives as providing efficient objects for scientific computing over MPI (POOMA [5], Petsc [1]) or with more distributed objects (C++ / [7], ProActive [11]). Our first approach was to simplify the usage of the message passing library by providing tools for an efficient overlapping of communications and computations without a special care for complex structures and their potential dead-locks.

EasyMSG provides a few orthogonal features implemented over a Message Passing Library as communication layer (currently MPI):

- synchronous or asynchronous communications;
- unbuffered or buffered communications:
  a buffered communication is sent only when a buffer (the size is fixed by the user) is full (or when another datum cannot be added);
- block communications:
  may be used for sending vectors of data (need a buffer with enough space);
- special communication tagging:
  for sending a tag datum with data; used for identify received data;
- several algorithms for multiple receivers or senders:
  as several buffers to a target process or from a process;
- Wrappers for the identification of the type to transfer are also implemented. However, a limitation is to send only serialized data, i.e. vectors of data. An implementation of the serialization of any object may be done inside EasyMSG, if we can determinate the overall size of such an object.

Remark 1. An important characteristic of such communications is the unpredictable order of communication reception. If one sends two data $A, B$, we cannot know how they will arrive: $A$ then $B$, or $B$ then $A$! This is a very good reason for using tagged data.
2. Operations Decomposition

Even using asynchronous communications may be not enough for optimizing a program. Optimizing the overlapping of communications and computations may be necessary.

We call local computations, computations without any communication. Furthermore, we define communication tasks for each target process. Our goal is to know what kind of communication and computation we will do for each target at any time. Such decomposition of the work allows to use the following algorithm:

**Algorithm 2.1.** *(Fig. 8)* A simple way to overlap communications and computations

Let InBuffer, OutBuffers be collections of buffers for receiving and sending

while Something to do do
  if there is a ready datum in InBuffer then
    Treat the incoming message
  else
    for all cpu, do
      if OutBuffers[i] is not full then
        Prepare and push a datum into OutBuffers[i]
      end if
    end for
  else
    Do some local computations
  end if
end while

**Remark 2.** Each test on OutBuffers and InBuffer is a test which uses specific algorithms, like a rotating buffer (uses a circular multiple buffer), or a multiple buffer (uses free buffers even if the sending order is different).

Then, the important aspect for a good efficiency is the definition of “do some local computations”. Its granularity must be small enough for avoiding to spend lot of time (and not be ready to treat communications) and not too small to avoid useless tests (and to stress the network).

An optimal solution is obtained if we can do all communications and computations with a minimal time where the parameter is the number of computations by loop, called aggregation parameter.

3. C++ Implementation of EasyMSG

This library must be designed for heavy usage, with many calls to basis functions such as arrival test functions and many more. Another requirement is to allow simple interface for implementing new algorithms or container types. A way provided by the C++ language is inheritance and polymorphism via virtual functions. However, this way may be very inefficient, if we abuse of virtual functions. Virtual functions allow to access to functions of sub-classes from top level classes. Calling a virtual function from an abstract object (object with virtual function referring to sub-classes functions) needs to scan virtual function tables for defining where the corpus of the real function (in a sub-class). This is the cost of virtual functions. But, if we use them just for defining interfaces, this drawback disappears. We associate virtual functions to a well-defined interface and templates for a generic fast static access (defined at compile time).

**Remark 3.** Another way for associating a well-defined interface and genericity is to use static polymorphism.

The design of this library is organized around:

- **Locks**: objects designed for managing (single or multiple) communications: to know when they are done, available or buzzy;
- **Containers**: there are two kinds of Containers: for receiving (based on InBufferModel) and sending (based on OutBufferModel); they represent fixed buffers for storing linear data to transmit by applying (un-)serialization;
• **Communication algorithms**: to provide a common support for the *Containers* for managing *when* the data are received/sent; we can use multiple targets, multiple buffers, FIFO scheduling, …

• **External Parameters**: for the previous classes in order to provide the end user with a way to communicate with the communication level functions. In the MPI case, they allow to change/read MPI specific tags, targets, status.

Its main advantage is to allow to write easily efficient high-level parallel concurrent programs without any deadlocks.

The current implementation uses MPI as communication layer, but any other message passing library may be used by adapting the containers to the new library. The algorithms may be kept without any changes.

We will show three interfaces of common objects: the fundamental *InBufferModel*, a top level *MPI_InBlockComplexTagBuffer*, and an *MultiSender* algorithm.

### 3.1. InBufferModel

The *InBufferModel* class is the top level class of abstraction of buffers for incoming messages. Any higher Container receiving data must inherit from it.

```cpp
1  template<typename ObjT, typename InOutT, typename StorT>
2  class InBufferModel {
3    public:
4      typedef ObjT ObjectType;
5      typedef InOutT InOutType;
6      typedef StorT StorableType;
7    protected:
8      typedef InBufferModel<ObjectType, InOutType, StorableType> Model;
9    public:
10       InBufferModel();
11       InBufferModel(const unsigned _bufferSize);
12       virtual ~InBufferModel();
13       virtual InOutType pop() = 0;
14       virtual void resize(const unsigned _bufferSize);
15       virtual bool empty() const;
16       virtual unsigned sizeLeft(void) const;
17       virtual unsigned sizeOf(const InOutType &value) const = 0;
18       virtual bool canDoIt() const;
19       virtual bool mustReset() const;
20       virtual void reset();
21     template<typename Lock> void start(Lock &lock) = 0;
22     template<typename Lock> void stop(Lock &lock) = 0;
23   }
```

*The specific types are propagated by some typedef’s*

**Constructors and destructors**

```cpp
10     InBufferModel();
11     InBufferModel(const unsigned _bufferSize);
12     virtual ~InBufferModel();
```

*The main function: extracting a value of the buffer*

```cpp
13     virtual InOutType pop() = 0;
```

**Functions about managing the buffer size and its capacity**

```cpp
14     virtual void resize(const unsigned _bufferSize);
15     virtual bool empty() const;
16     virtual unsigned sizeLeft(void) const;
17     virtual unsigned sizeOf(const InOutType &value) const = 0;
```

**Functions about managing a new value (if it can)**

```cpp
18     virtual bool canDoIt() const;
19     virtual bool mustReset() const;
20     virtual void reset();
```

**Functions about starting/stopping communications**

```cpp
21     template<typename Lock> void start(Lock &lock) = 0;
22     template<typename Lock> void stop(Lock &lock) = 0;
23  ```
3.2. MPI_InBlockComplexTagBuffer

The MPI_InBlockComplexTagBuffer object allows to receive variable-length vector of data with an additional value: a tag. This tag must be of any fixed-length type. Moreover, this buffer carries an additional information for un-serializing the vector: the size.

In order to avoid large copied data as return values of the pop function, we use a couple of pointers to define the vector of data: one on the first element, another on the end of the vector i.e. equal to the first one plus the number of values. However, we want a single argument for the pop function, so the tag is also carried with the couple of pointers into a triple<const T*, const T*, Tag>.

```cpp
template<typename T, typename Tag, typename Parameter>
class MPI_InBlockComplexTagBuffer
: public InBufferModel<T, triple<const T*, const T*, Tag>, unsigned char> {
  protected:
  An external Parameter for driving low level communication function

  Parameter *parameter;
  public:
  A constructor referring to the top level class

  MPI_InBlockComplexTagBuffer(Parameter & _parameter, const unsigned _bufferSize)
  : Model(_bufferSize), parameter(&_parameter) { resize(bufferSize); }

  This main function manages the un-serialization.
  For performance reason, we avoid to use an additional buffer for un-serializing. However, we assume a non-growing size during the unpacking by MPI (with MPI_Unpack) (else we need an additional buffer and many copies).

  InOutType pop() {
    assert(canDoIt());
    unsigned data_count;
    MPICHECK(MPI_Unpack(toPointer(buffer.begin()), buffer.size(), &position,
                        &data_count, 1, MPI_UNSIGNED, MPI_COMM_WORLD));
    Tag tag;
    MPICHECK(MPI_Unpack(toPointer(buffer.begin()), buffer.size(), &position, &tag, 1,
                        MPI_Object<Tag>::Type_Id, MPI_COMM_WORLD));
    ObjectType *ptr = reinterpret_cast<ObjectType*>(toPointer(buffer.begin()));
    MPICHECK(MPI_Unpack(toPointer(buffer.begin()), buffer.size(), &position, ptr,
                        data_count, MPI_Object<ObjectType>::Type_Id, MPI_COMM_WORLD));
    return InOutType(ptr, ptr + data_count, tag);
  }

  The sizeOf function computes the size of a vector of data including tag and size description.

  unsigned sizeOf(const InOutType &value) const;

  The communication functions start/stop mix the two control parts Parameter, Lock for communicating. This allows the end user to drive the communication functions (here via MPI) and the algorithm to know the state of the communication.

  template<typename Lock> void start(Lock &lock);
  template<typename Lock> void stop(Lock &lock);
};
```
3.3. MultiSender

The MultiSender algorithm provides multiple buffers of the same type for sending data. The scheduling is “as soon as possible” for each buffer. The order may be not preserved but we have a larger capacity without waiting to fill every buffer before sending.

```cpp
template<typename Container, typename MultiLock>
class MultiSender {
protected:
    Container *currentBuffer;

    Internal function for cleaning the current (pointed by currentBuffer) buffer referring to reset function of the specific container
    void reset();

    A constructor including generic parameters for building vectors of Containers (buffers) and Locks (multiLocks)
public:
    template<typename Parameter1, typename Parameter2>
    MultiSender(const unsigned _bufferSize, Parameter1 &param1, Parameter2 &param2);

    Accumulate a new value: if the current buffer is full, search another ready buffer, else wait.
    This function may be written as two separated non-blocking parts for improving the performance: test for testing before and fpush for sending without any test. Moreover an again function can try to start the buffer with a “full enough?” a priori test; this is optional for outgoing communications but essential for incoming communications: starting a reception before needing it.
    void push(const typename Container::InOutType &value) {
        if (!test(value)) {
            flush();
            wait();
        }
        assert(currentBuffer->canDoIt(value));
        currentBuffer->push(value);
    }

    Functions for waiting/testing data of the current buffer.
    The management of communications over a buffer is delegated to its lock.
    void wait();
    bool test(const typename Container::InOutType &value) {
        if (!currentBuffer)
            return false;
        if (currentBuffer->canDoIt(value))
            return true;
        if ((currentId = multiLocks.test_any()) != -1) {
            currentBuffer = &buffers[currentId];
            reset();
        } else return false;
        return currentBuffer->canDoIt(value);
    }

    Send now the current (not empty) buffer, and invalidate it
    void flush() {
        if (!currentBuffer || currentBuffer->empty()) return;
        currentBuffer->start(multiLocks[currentId]);
        currentBuffer = NULL;
    }

    Stop all buffers by waiting the end of the communications.
    (An additional abort function exists for killing communications before exiting)
```
3.4. An example

This example shows how we use our library for writing overlapping programs. The goal is simply to exchange data (send/receive) between ncpus processes and to do local computations. This is the transfer step; which has been simplified (no explicit computation are given).

The performance of this code will be described in last part of this article.

Current EasyMSG I/O Type with a tag typed uaddr_t (universal address of a cluster)

```c
typedef triple<const BaseType *,const BaseType *,uaddr_t> InOutBufferType;
```

The Receiver parameters (start the receiving communication soon as possible)

```c
MPI_Parameters pop_params;
pop_params.setTarget(MPI_ANY_SOURCE);
pop_params.setTag(TRANSFER_TAG);
```

```c
MultiReceiver<
    MPI_InBlockComplexTagBuffer<BaseType,uaddr_t,MPI_Parameters>,
    MPI_MultiLocks<MPI_Parameters> > popper(TRANSFER_BUFFER_NUMBER,
    TRANSFER_BUFFER_SIZE,
    pop_params,pop_params);
```

We use an array of senders (one for each target).

```c
std::vector<MPI_Parameters> pushers_params(ncpus);
typedef MPI_OutBlockComplexTagBuffer<BaseType,uaddr_t,
    MPI_Parameters> SenderContainer;
```

```c
SenderArray<
    SimpleSender<
        SenderContainer,
        MPI_SimpleLock<MPI_Parameters> > > pushers(ncpus,TRANSFER_BUFFER_SIZE,
    pushers_params);
```

```c
for(int cpu=0;cpu < ncpus;++cpu) {
    pushers_params[cpu].setTarget(cpu);
    pushers_params[cpu].setTag(TRANSFER_TAG);
}
```

We assume the knowledge the number of expected requests (expectedRequests), the list of communications to send (transferGroups), and a list of local computations (localProgresser).

```c
unsigned expectedRequests = totalTransferRequests.sum();
for(int cpu=0;cpu<ncpus;++cpu) transferGroups[cpu].again();
TransferIterator localProgressor;
while(something_to_do(expectedRequests,transferGroups)) {
    Treat the incoming message from pop_params.getSource(). We have to call the again function in order to inform popper that we have successfully treated the incoming request and it can release now the buffer area allocated for this datum.
```
if (popper.test()) {
  do {
    InOutBufferType val = popper.pop();
    /* Treat the incoming request */
    popper.again();
  } while(popper.test());
}

Send tagged vector \( v \) tagged by \( \text{tag} \) (its address) to the other processes while it can. We use a combination of test and fpush (faster than push after test), which avoid deadlock.

for(int cpu=0;cpu<ncpus;++cpu)
  while(!transferGroups[cpu].empty()) {
    InOutBufferType val(v.begin(),v.end(),tag);
    if (pushers[cpu].test(val))
      pushers[cpu].fpush(val);
  }

Do a few local computations according localProgresser, where /* how many? */ is a iteration limiter
Do remaining local computations
Wait for the end of the previously sent communications

4. Adaptive dynamic overlapping

4.1. Context definition

Usually, the network traffic is non-linear, even unpredictable during high traffic load. However, many scientific programs do repetitive tasks such as in an iterative solver, and by iterating we accumulate informations about the context of execution for improving the average timing. We will illustrate a way to manage the network traffic in order to increase the overall speed of parts mixing communications and computations used into our parallel solver of Maxwell equations by Fast Multipole Method 3.4.

Applying generic Algorithm 2.1 needs to define “do some local computations”. The main goal of our approach is to reduce the global execution time.

Complex communications (with many sources, targets, buffers) may be expensive to test for nothing, when no communication is ready. That is why increasing the local computations for each loop may reduce the overall execution time. However, if local computations are too long (or the granularity of local computations is too coarse), incoming/outgoing communications may wait to be treated, then the execution time may decrease.

Furthermore, the architecture and the load of the computers and the network may be variable between different machines and between different times. Then, defining this aggregation statically will never be optimal for every computer.

Our first step is to optimize the global time of a part of code by computing a more optimal aggregation parameter \( p \).
4.2. Explorative exponential search

The optimization may be done, by minimizing the "execution time" as a nearly convex function with a dichotomic method with a single starting point $p = 0$ (no aggregation). Usually, the dichotomy with a single left (resp. right) starting point, find a right (resp. left) bound by an exponential search. Then, a standard dichotomy can start with two starting points.

However, the 'execution time' is not really a well-defined function: evaluated at several times with a same aggregation parameter $p$ does not give a stable value, especially if the evaluations are not consecutive. This 'function' is linked to the global state of the computer load (CPUs, network, ...).

Thus, we need a very stable and responsive algorithm, if the 'optimum' changes during the optimization (the environment may change during execution).

Our main difference with a standard dichotomy is our explorative exponential search is not based on placing one new better point between two other points, but placing one better point from the knowledge of one point and one vector of exploration: the length of this vector is increased when the next try is better; the direction of the vector is changed and the length is reduced when the next try is worse (Fig. 9).

Remark 4. A better point means a point that produces a more “near to optimum” point than previous points, all previous points for the dichotomic scheme, the previous one for our explorative scheme.

4.3. Optimizing in a real world

The previous explorative exponential search algorithm is used at each execution of Algorithm 2.1 for adapting the aggregation parameter $p$.

Some remarks for algorithmic improvements.

- The execution time is a very fuzzy function: even several consecutive measures with a same parameter $p$ may give different results. We use a mean of $n$ evaluations as reference values, i.e. we change the aggregation parameter $p$ every $n$ executions (currently, $n$ is 3 or 4).
- Our search domain for $p$ is bounded on the left by 0.
- The first explorative vector is always rightwards.
- If the algorithm tries to go beyond 0, we slow down the search by decreasing the length of the explorative vector so that $p > 0$.
- We interpret non integer value of $p$ as an aggregation parameter mean.
  $\forall p \in \mathbb{R}_+, \text{ we aggregate } p_i = \lfloor p \cdot i - p_{i-1} \rfloor $ local computations at loop index $i \geq 1$ ($p_0 = 0$, $\lfloor \cdot \rfloor$ represents the integer part operator).
- If the length of the explorative vector is very low, the algorithm becomes less responsive to a significant modification of the global state. Moreover, the execution time is not well-defined and widely varying.
between several evaluations, so two too close parameters may be not significant: we bound the minimal length of the explorative vector.

5. Computational results

We will study the efficiency of our optimization for adaptive overlapping on three problems of different sizes, all arising from the multipole method.

- Small: $N = 75,000$ degrees of freedom, 4MB of data to send divided into 700 vectors for each process;
- Medium: $N = 300,000$ degrees of freedom, 14MB of data to send divided into 1300 vectors for each process;
- Large: $N = 600,000$ degrees of freedom, 31MB of data to send divided into 2000 vectors for each process.

For each case the global number of transfers to compute is $O(N)$. We will use 8 CPUs Intel PIII/1266MHz-1GB for each computation and a switched Ethernet 100Mb network.

5.1. Without our adaptive overlapping

Our first tests are without our adaptive overlapping but a simple static overlapping (1 local computation for each “do some local computations”).

The following graphs (Figs. 10–12) show the execution times of a part of code by the 8 processors during 140 iterations for the three problem sizes: Small, Medium, Large.

These measures are our reference execution times.
5.2. With our adaptive overlapping

We will study the same problems with our algorithm by optimizing the aggregation parameter $p$ using our explorative exponential algorithm (Sect. 4.2). This method leads to a uniform decomposition of the local computations as fixed-size packets of local computations during the communication step. When there is no more communication, the main loop stops and does the remaining local computations.

We show the execution times and the “convergence” of the aggregation parameters for each problem. According to these graphs, we can see independent timings and convergences for each process, even if our homogeneous architecture provides very close values. The algorithm starts with the worst value (when $p = 0$) and try to reduce the average timing; at the second iteration $p = 1$ and we obtain the same execution timing as the references cases (see Sect. 5.1). This parameter $p$ grows exponentially up to obtain a execution timing worst than the previous result.

We obtain a more “optimal” timing (about 18% faster) after about 20 evaluations (i.e. 5 updates of $p$), and the convergence of $p$ is near to be stabilized after 15 updates (i.e. 60 evaluations) (Figs. 13–18).

The number of updates before giving a good result, even with a nearly chaotic function, is really the main goal for our algorithm without falling in a local optimum; many other classical algorithms, using very smooth functions, are really inefficient here.

5.3. With a more responsive adaptive overlapping

The previous case uses a constant aggregation parameter for the aggregation of local computations in the main loop, and updates it between each full execution. However, the network traffic is not homogeneous during one whole execution: at the beginning, many processes start their communications and there are many communications to treat. Nevertheless, our previous implementation was designed for treating as a priority
Figure 16. Small.

Figure 17. Medium.

Figure 18. Large.

Aggregation parameter $p$ convergence ($p$ is updated after an average of 4 evaluations).

the communications, so performing as a special case the inhomogeneous communications may not improve the global execution time in the same context.

However, our previous test was done in an ideal network without any other traffic. In many cases, a shared cluster may produce a very inhomogeneous traffic without any foreseeable behavior. That is why we will compare our previous algorithm with another one with a better responsiveness to the unstable environments.

This new test uses the same algorithm for optimizing the execution time but the aggregation parameter is replaced by a variable aggregation based on an increasing evolution when there is no communication (according to the last communication tests), and a fast decreasing evolution when there are communications: we will call it fast adaptive algorithm.

In fact, we use the following heuristic: if the previous test says “there are some incoming/outgoing data”, the associated buffers were full, so there may remain some incoming/outgoing data for the same buffers, then we want to test them as early as possible and do less local computations; else we can do more local computations because if the last test fails and this may be an indicator of a lower network activity.

The following graphs (Figs. 19 and 20) show the same computations with a stressed network (an additional program providing an irregular high traffic on the cluster network)

Even if the mean of timings is better for our fast adaptive algorithm, these results are not so significant. In fact, the previous algorithm was already written for aggregating communications (a loop try to fill/flush the current buffer). But, if we remove this optimization (Figs. 21 and 22), we can see the effect of the last algorithm against a manual optimization by the programmer.

**Remark 5.** The new algorithm converges faster to the solution, because it is more adaptive to the environment (the aggregation parameter grows when there are few communications) and its evolution parameter of the
adaptive communication aggregation is smaller, so it needs fewer steps of exponential search for finding good candidates.

The last results show a real better efficiency (about 10%) for the fast adaptive algorithm: it tries itself to aggregate the communications or increases the local communications according to the network traffic. However, these results are less efficient than a manually optimized version because the manual tuning was done in order to avoid local computations between consecutive communication tests, which is our final goal: the algorithm understands how reducing the execution timings.

5.4. **Low speed CPU environment**

Finally, we will test the adaptability on a different CPU environment. Here, we use the same network but with 8 PIII-800MHz processors.

As planned, the algorithm converges towards another parameters without any help (Figs. 23 and 24). This is the first motivation for using these algorithms: the tuning is automatically done.

6. **Conclusion**

This adaptive overlapping method with our EasyMSG communication library over MPI provides an improved efficiency even for codes already optimized by statically defined overlapping. Our tests show a performance up
to 20\% better into a massively communicant program as the transfer step of the Fast Multipole Algorithm applied to High Frequency Maxwell Equations. Moreover, we have seen another type of optimization with the fast adaptive algorithm, where the inner loop is also optimized on the fly according to the network traffic. This last algorithm may be more efficient than the first algorithm when the programmer did not aggregate explicitly the communications as long as there are, even with more unstable computing environment: the algorithm tries to aggregate dynamically communications (\emph{i.e.} with less local computations between communications).

However, these algorithms use a too competitive optimization method between processes (each process tries to optimize according its own criterions), where a more cooperative method (using global criterions) may be more efficient (for the global execution timing, not the convergence rate).

There are many other small parameters to tune for improving the results (as the buffer size), but they are very difficult to optimize with few evaluations (about 20) and measuring the environment is also a part of the solution (measuring the bandwidth may provide a optimal buffer size for a given transfer time).

Furthermore, the convergence to an other “optimal” parameter may be improved when the environment changes during the computations. Our current implementation needs many steps to adapt again the new optimal parameter by increasing the explorative vector, even with a low bound on this parameter.

Future improvements trend towards optimizing communicant codes with few efforts.
Figure 23. Execution timing.

Figure 24. Aggregation parameter.

REFERENCES