ZeroMQ for massively parallel codes

Anthony Scemama[1] <scemama@irsamc.ups-tlse.fr>

[1] Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Introduction

- Today : machines with 100k - 1M cores
- Bulk synchronous parallel algorithms don't scale : every synchronization barrier kills the efficiency
- Hybrid MPI/OpenMP with asynchronous communications does a better job
  - MPI is not a universal solution
  - OpenMP works very well on independent data (tasks), but when locking is needed it becomes a nightmare or it doesn't scale with hundreds of threads
  - MPI and OpenMP are programmed very differently

- In a workflow, all steps don't scale equivalently : we need dynamic (elastic) resources
- The machines are more complex => less reliable
- The machines are more and more heterogeneous : CPUs, GPUs, XeonPhi, FPGA, Cloud resources etc...

System failures

Example:

- A job runs on 200 000 cores (10 000 nodes with 20 cores each)
- Each node has 64GiB of RAM : 8 modules
- The MTBF of a medium quality memory module if 500 years
- If all memory modules are statistically independent, the failure of a memory module will happen in 500 x 365.25 / (10 000 x 8) = 2.3 days !

On massively parallel simulations, failures happen all the time and they should be part of the design :

- File system (Lustre!) failures : use */dev/shm* if possible
- Memory failure : stop using one node, and restart the corresponding tasks
- Network failure : recover if possible
- Power failure : checkpoint/restart

MPI

Initial design (MPI 1):

- Single Program Multiple Data paradigm (avoidable since MPI 2)
- Very tightly coupled processes
- Synchronous communications
- Asynchronous send/receive exist but are more difficult to use
- The network is supposed to be fast, reliable, with low latency
- Designed for a fixed amount of resources and limited buffers
- All messages are supposed to be consumed
- **Not fault tolerant** : if one process crashes for any reason, all the simulation is killed (or the program is in an undefined state)

As this initial design does not scale to millions of cores:

- Coupling MPI with OpenMP is encouraged : reduces the number of MPI processes and more memory per process is available
- MPI 3 introduces one-sided communications (MPI_Put / MPI_Get).

Can we do better?

Program design

- SPMD encourages monolithic programs :
  - Pain for the programmer
  - If there is a bug in one task, everything crashes : handling failures is difficult
- Unix philosophy : https://en.wikipedia.org/wiki/Unix_philosophy
  - "Make each program do one thing well"
  - "Make every program a filter"
  - If one component fails, all the other components are unaffected

Microservices : *(Wikipedia)*

In computing, microservices is a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs. These services are small, highly decoupled and focus on doing a small task, facilitating a modular approach to system-building.

- Highly decoupled => scaling!
- Language-agnostic API : Use the most appropriate language.

Example in quantum chemistry

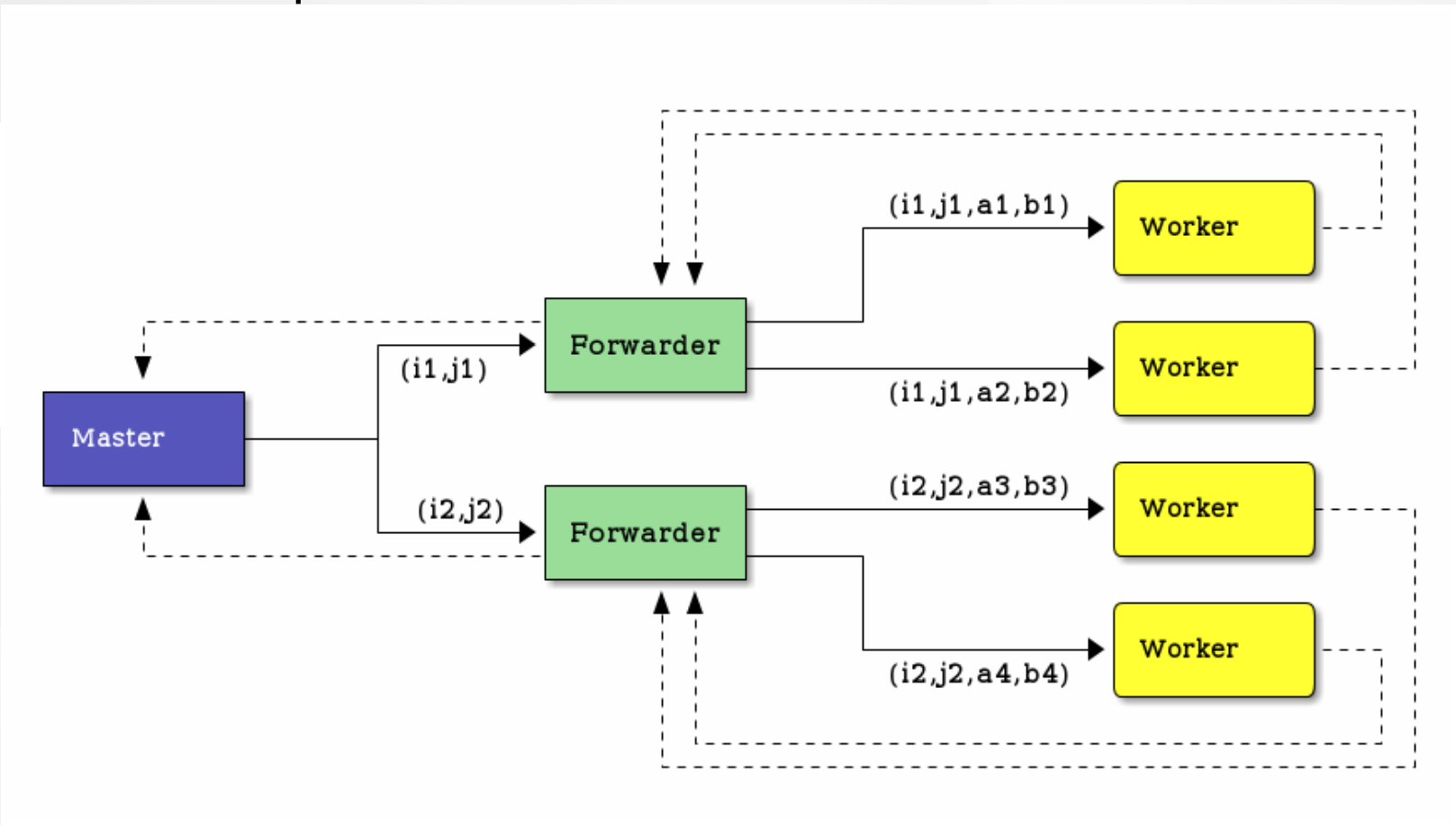
MP2 as a simple illustration (not the most optimal implementation)

$$E = 2\sum_{i,j}^{\text{occ.}}\sum_{a,b}^{\text{vir.}} \frac{\langle ij|ab\rangle\langle ab|ij\rangle}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b} - \sum_{i,j}^{\text{occ.}}\sum_{a,b}^{\text{vir.}} \frac{\langle ij|ab\rangle\langle ab|ji\rangle}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b}$$

In a human organization:

- The professor gives tasks to a few graduate students
- Each graduate student gives smaller tasks to many undergraduate students
- When the task of the undergraduate is done, he/she gives the result to any graduate student

- The graduate students combine the results and give the new result to the professor who creates the result



- Any meeting in the middle of the process will slow things down
- The individuals can work whenever they want
- Some are more efficient than others, so the work should be distributed dynamically
- If any individual gets sick, it is always possible to find a way to replace him/her

This very natural scheme has nothing to do with bulk synchronous parallelism and SPMD!

Master

- Prepare a list of pairs (i,j)
- When a forwarder requests a task :
  - Send the couple (i,j)
  - mark the task as running
- When a result is received :
  - Send the next couple (i,j)
  - Mark the next task as running
  - Mark the task as done
  - Accumulate the result
  - Checkpoint the current state

- When the queue is empty :
  - Continue to send currently running tasks to free forwarders
- When all the tasks are done :
  - Tell all the forwarders to terminate
  - Accumulate the result
  - Terminate (without waiting for the forwarders)

Forwarder

- Get a pair (i,j) from the master
- Prepare a list of triplets (i,j,a)
- When a worker requests a task :
  - Send the triplet (i,j,a)
  - mark the task as running
- When a worker sends back the result :
  - Send the next triplet (i,j,a)
  - Mark the next task as running
  - Mark the task as done
  - Accumulate the result

- When the queue is empty :
  - Continue to send currently running tasks to free forwarders
- When all the tasks are done :
  - Accumulate the result
  - Send to the master
  - Request another pair (i,j)
- When the master requires termination :
  - Kill all the known workers
  - Terminate

Worker

- Ask for (i,j,a) to the forwarder
- Compute the sum over all b
- Return the result to the forwarder
- Request another task

- This scheme allows to add/remove resources at any time.
- Resilience : any point can fail without affecting the calculation
- Fortran/C can be used for workers, and Python/Ocaml can be used to handle queues easily
- This scheme *scales*

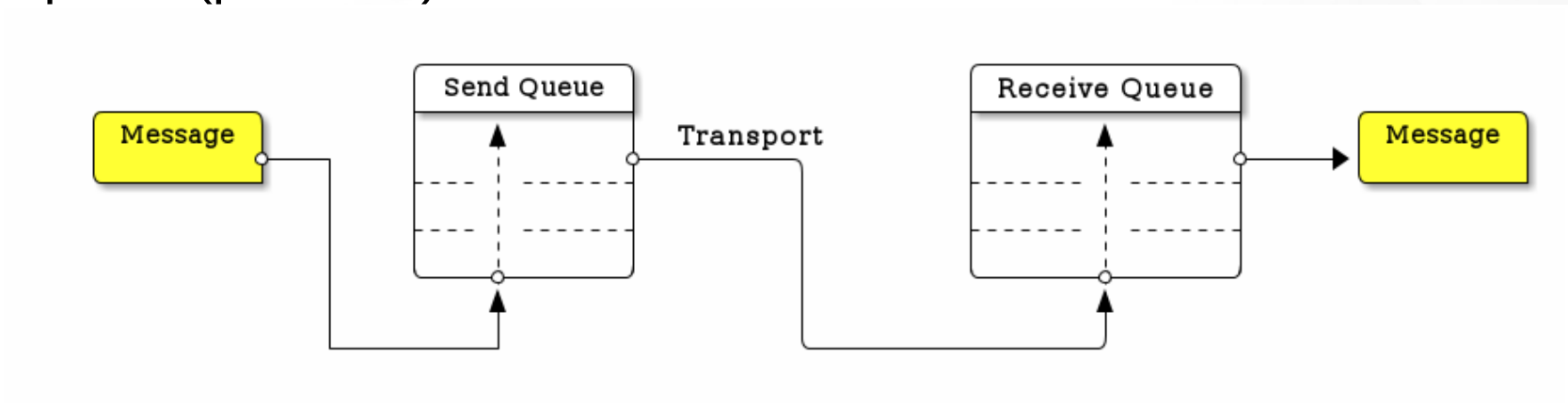
ZeroMQ

- Communication library based on *asynchronous* message passing
- Designed for finance (high volume trading)
- Lots of messages on the internet (unreliable, high latency network)
- Resilient : handles network disconnections
- High Performance : 13.4 microsecond end-to-end latency on IB network, > 8 million messages/second
- Same API for inter-thread, inter-process, network message passing

- API is very simple to use (similar to BSD sockets)
- Encourages microservices
- Available in multiple languages : C, C++, Java, Fortran, PHP, Python, Ocaml, Perl, Erlang, Ruby, Lua, C#, Haskell, Go, Scala, and more ...
- Open Source (LGPL)
- Very portable : library is in C++.
- I was able to compile it on a Xeon Phi and get communications to work in 5 minutes.
- Used for Multi-GPU programming in *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* (Shane Cook)

Sockets

- A socket is associated with a queue
- Sending a message : putting the message in the send queue (post box)
- Receiving a message : getting a message from the receive queue (post box)

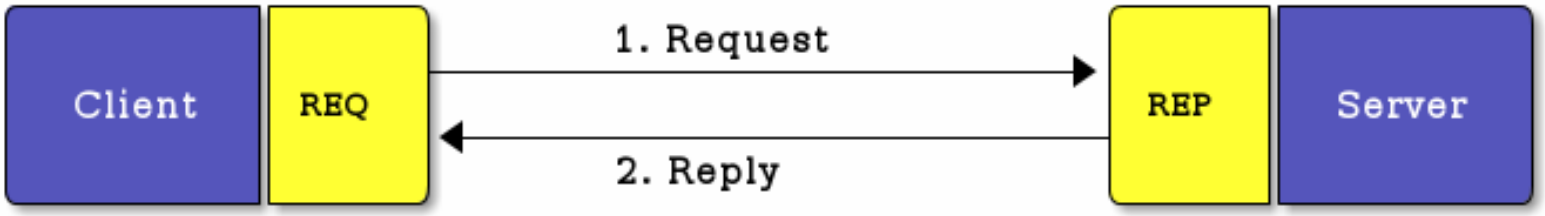


Message Patterns

Socket pairs express different messaging patterns:

- REQ / REP : Request / Reply, two-sided communications
- PUSH / PULL : One-sided
- PUB / SUB : Publish / Subscribe, one-to-all (can use multicast)
- ROUTER / DEALER : Involved in load balancing / high availability

REQ / REP



Always this sequence :

1. Client sends a request
2. Client receives a reply

Server code:

```
import zmq

context = zmq.Context.instance()
socket  = context.socket(zmq.REP)
socket.bind("tcp://*:12345")

request = ""
while request != "end":
    request = socket.recv()
    socket.send("Received %s"%(request))
```

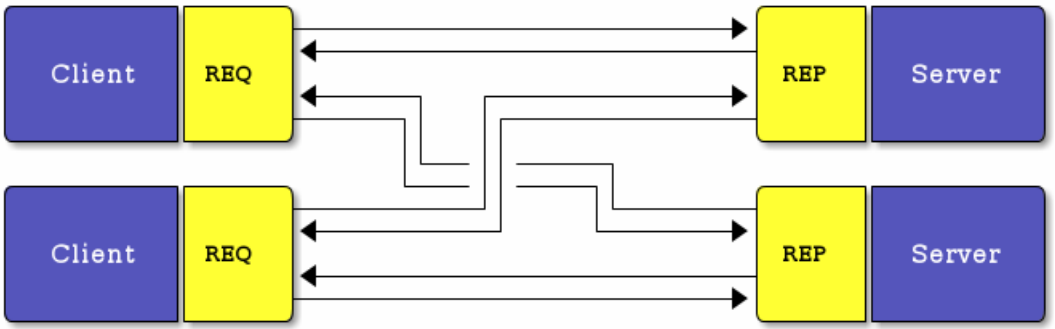
Client code:

```
import zmq, time
context = zmq.Context.instance()
socket  = context.socket(zmq.REQ)
socket.connect("tcp://localhost:12345")
for i in range(3):
    socket.send("my_request")
    reply = socket.recv()
    print "received ", reply
    time.sleep(1)
socket.send("end")
reply = socket.recv()
```

Similar to BSD sockets : the number of clients is not fixed

Different from sockets :

- Binding can be done before or after connecting
- Messages are fairly queued => All clients will be served even if one client floods the server
- One REQ socket can connect to multiple REP sockets : load balancing



New server code:

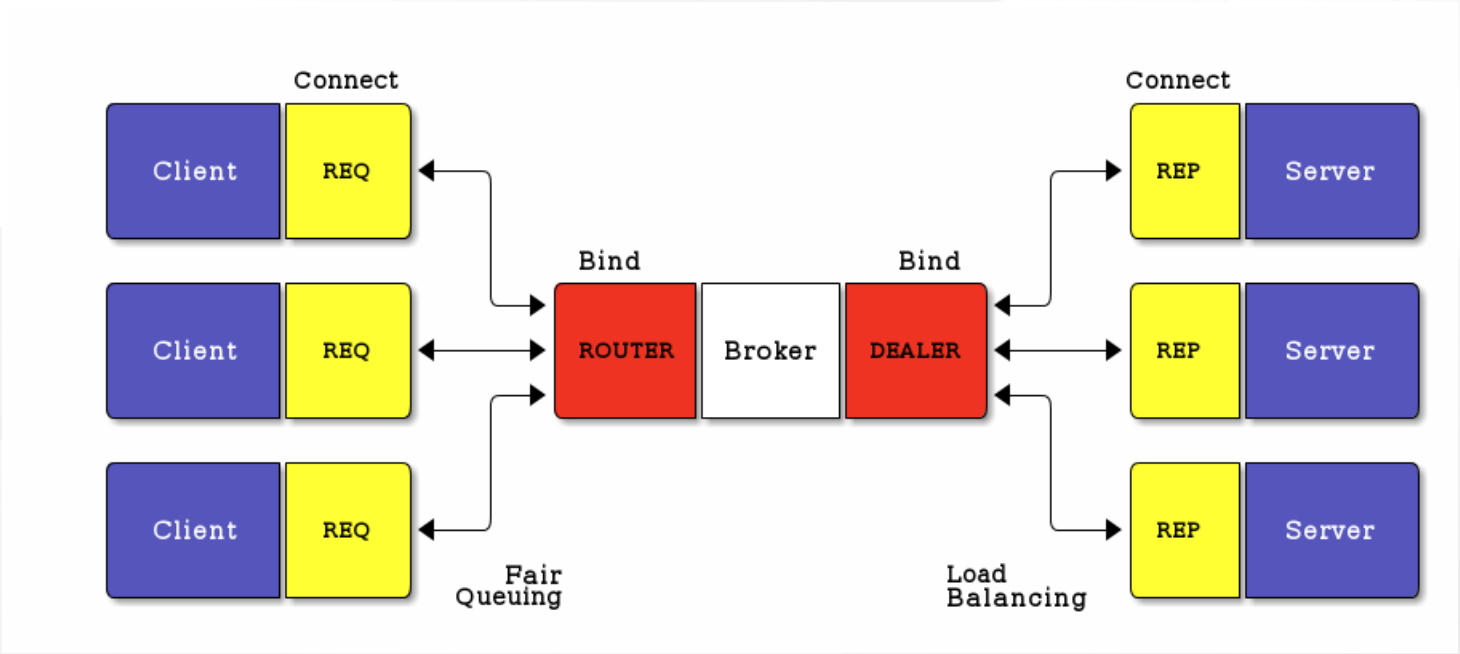
```
import zmq

context = zmq.Context.instance()
socket  = context.socket(zmq.REP)
socket.bind("tcp://*:12346")  # <- new port number here

request = ""
while request != "end":
    request = socket.recv()
    socket.send("Server 2 received %s"%(request))
```

Client code:

```
import zmq, time
context = zmq.Context.instance()
socket  = context.socket(zmq.REQ)
socket.connect("tcp://localhost:12345")
socket.connect("tcp://localhost:12346") # <- 2nd server here
for i in range(7):
    socket.send("my_request")
    reply = socket.recv()
    print "received ", reply
    time.sleep(1)
socket.send("end") ; socket.recv()
socket.send("end") ; socket.recv() # <- End both servers
```

Router / Dealer

In the previous example, for each new REP server added we had to give its address to the clients. The Router / Dealer pattern inserts a static broker in the middle of the connexions:



PUB/SUB

Publish / Subscribe.

- Analogous to the radio : if come too late, you miss the show
- All the subscribed clients receive the message
- If no client is subscribed, the message is silently dropped
- Example : logging

Server:

```
include 'f77_zmq.h'
integer(ZMQ_PTR) :: zmq_context, debug_socket
integer          :: rc

zmq_context = f77_zmq_ctx_new ()
debug_socket = f77_zmq_socket(zmq_context, ZMQ_PUB)
rc = f77_zmq_bind(debug_socket, 'tcp://*:12345')

allocate (temp_array(Nmax,Nmax))
write (message, *)  &
    'Mem: ', Nmax*Nmax*8*1024, ' KiB Allocated'
call f77_zmq_send(debug_socket, message, &
```

```
    len_trim(message), 0)

do i=1,Nmax
  read(10) integrals
  write (message, *) 'I/O: ', Nbytes, ' read'
  call f77_zmq_send(debug_socket, message, &
      len_trim(message), 0)

  do j=1,Nmax
    do k=1,Nmax
      do l=1,Nmax
        ...
      end do
```

```
    end do

    write (message, *) 'I/O: ', Nbytes, ' written'
    call f77_zmq_send(debug_socket, message, &
        len_trim(message), 0)
  end do

  write (message, *) 'CPU: ', real(i)/real(Nmax), '%'
  rc = f77_zmq_send(debug_socket, message, &
      len_trim(message), 0)

end do
deallocate (temp_array)
```

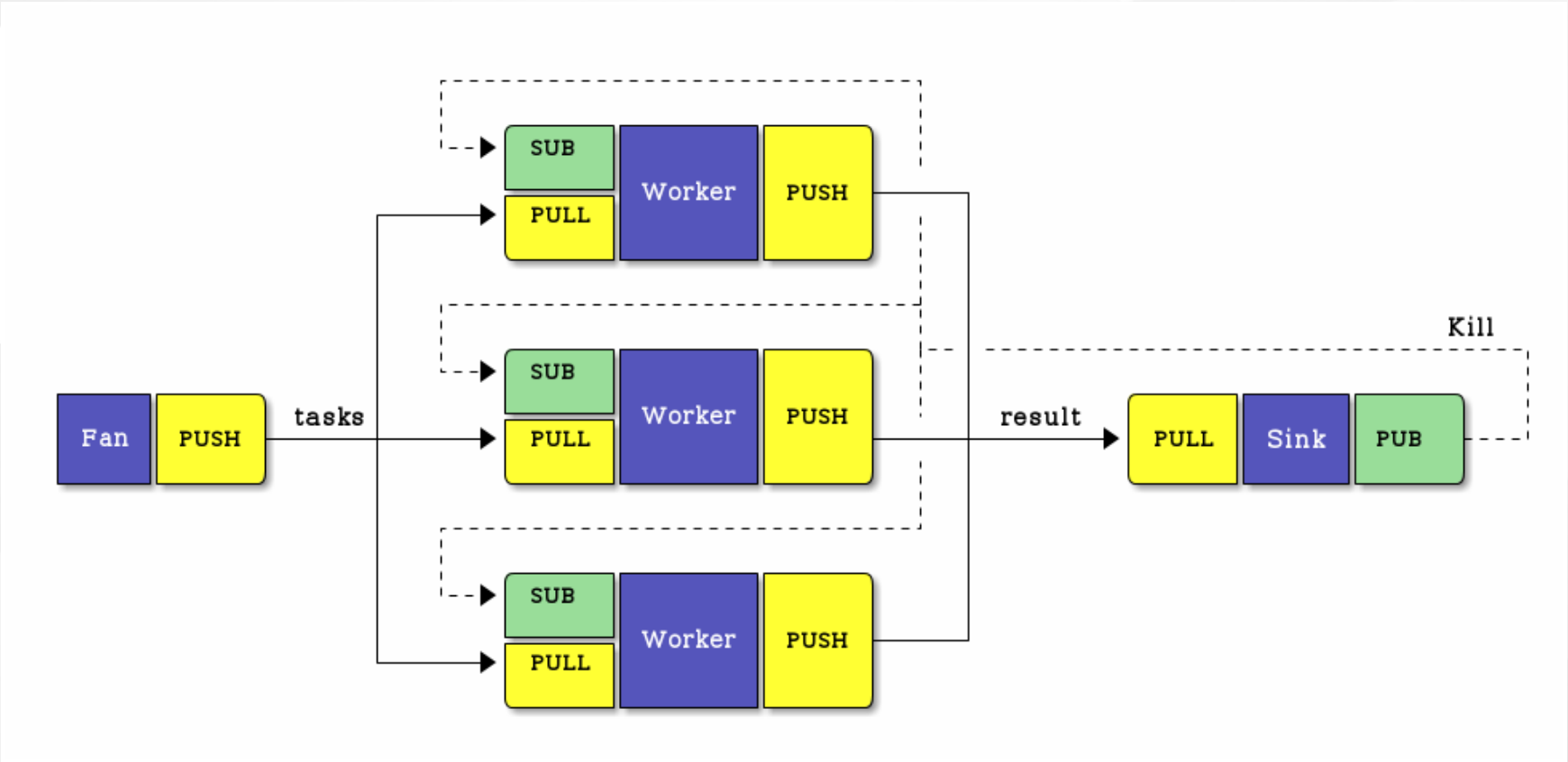
```
write (message, *) 'Mem: ', Nmax*Nmax*8*1024, &
    ' KiB Deallocated'
call f77_zmq_send(debug_socket, message, &
    len_trim(message), 0)

rc = f77_zmq_close(debug_socket)
rc = f77_zmq_ctx_destroy(zmq_context)
```

Client:

```
import zmq
import sys

context = zmq.Context.instance()
socket  = context.socket(zmq.SUB)
socket.connect("tcp://localhost:12345")
for i in sys.argv[1:]:
  socket.setsockopt(zmq.SUBSCRIBE, i)
while True:
  message = socket.recv()
    print message
```

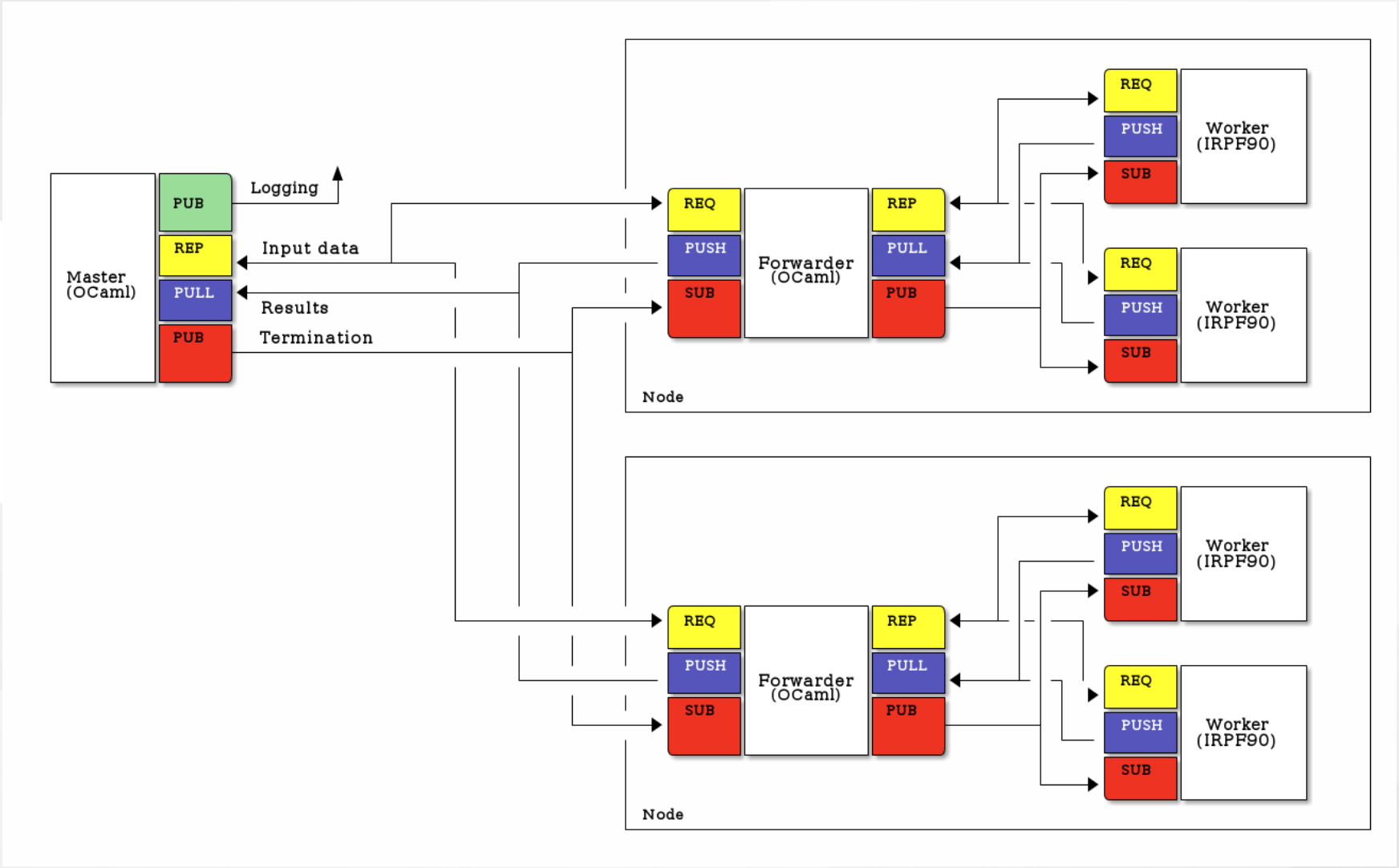
PUSH/PULL

One-sided asynchronous send / receive.



Real Example : QMC=Chem

- Quantum Monte Carlo code developed at LCPQ
- 98.4 % parallel efficiency on 16 000 cores (3 hours run)
- Fully asynchronous
- Fully resilient
- Elastic resources
- Combined run on Desktop computers, HPC cluster (CALMIP) and Cloud infrastructure (France Grilles)


Master
(OCaml)
PUB
REP
PULL
PUB
Logging
Input data
Results
Termination
Node
REQ
PUSH
SUB
Forwarder
(OCaml)
REP
PULL
PUB
Worker
(IRPF90)

Other Use Cases

- Visualization : use PUB/SUB to send data to a real time 3D renderer for molecular dynamics (DL_POLY_4 PRACE summer project)
- Asynchronous I/O on compressed files with a proxy on each node
- Debugging / measuring performance of MPI : Each process writes a log in a PUB socket on the ethernet network.

Project at the LCPQ : Parallel tempering Path-Integral Monte Carlo algorithm

- When a new client connects, a new temperature is given
- Each temperature is an MPI run with ~16-64 replicas (tightly coupled nodes)
- Each replica uses multi-threaded MKL (24 cores/node)
- Each MPI run regularly sends in a PUB socket its energy to a master server
- The master server randomly chooses which temperatures to exchange according to the running energies
- The exchange is sent by the server in a PUB socket with the new temperatures

- The points of the trajectories are sent using PUSH sockets to I/O servers for compressed storage, or on-the-fly evaluation of properties

Links

- ZeroMQ : http://zeromq.org
- ZeroMQ Guide : http://zguide.zeromq.org/
- Fortran Interface : https://github.com/scemama/f77_zmq (for ZeroMQ 4.0)

Other libraries worth looking at:

- Nanomsg : a fork of ZeroMQ in C (no more C++)
- GASPI-GPI : A fault tolerant asynchronous low-latency communication library for HPC