Evaluating the Performance Limitations of MPMD Communication

Chi-Chao Chang
Dept. of Computer Science
Cornell University

Grzegorz Czajkowski (Cornell)
Thorsten von Eicken (Cornell)
Carl Kesselman (ISI/USC)

Framework

Parallel computing on clusters of workstations

- Hardware communication primitives are message-based
- Programming models: SPMD and MPMD
- SPMD is the predominant model

Why use MPMD ?

- appropriate for distributed, heterogeneous setting: metacomputing
- parallel software as “components”

Why use RPC ?

- right level of abstraction
- message passing requires receiver to know when to expect incoming communication

Systems with similar philosophy: Nexus, Legion

How do RPC-based MPMD systems perform on homogeneous MPPs?

Problem

MPMD systems are an order of magnitude slower than SPMD systems on homogeneous MPPs

1. Implementation:
   - trade-off: existing MPMD systems focus on the general case at expense of performance in the homogeneous case

2. RPC is more complex when the SPMD assumption is dropped.

Approach

MRPC: an MPMD RPC system specialized for MPPs

- best base-line RPC performance at the expense of heterogeneity
- start from simple SPMD RPC: Active Messages
- “minimal” runtime system for MPMD
- integrate with a MPMD parallel language: CC++
- no modifications to front-end translator or back-end compiler

Goal is to introduce only the necessary RPC runtime overheads for MPMD

Evaluate it w.r.t. a highly-tuned SPMD system

- Split-C over Active Messages

MRPC

Implementation

- Library: RPC, basic types marshalling, remote program execution
- about 4K lines of C++ and 2K lines of C
- Implemented on top of Active Messages (SC ‘96)
  - “dispatcher” handler
- Currently runs on the IBM SP2 (AIX 3.2.5)

Integrated into CC++:

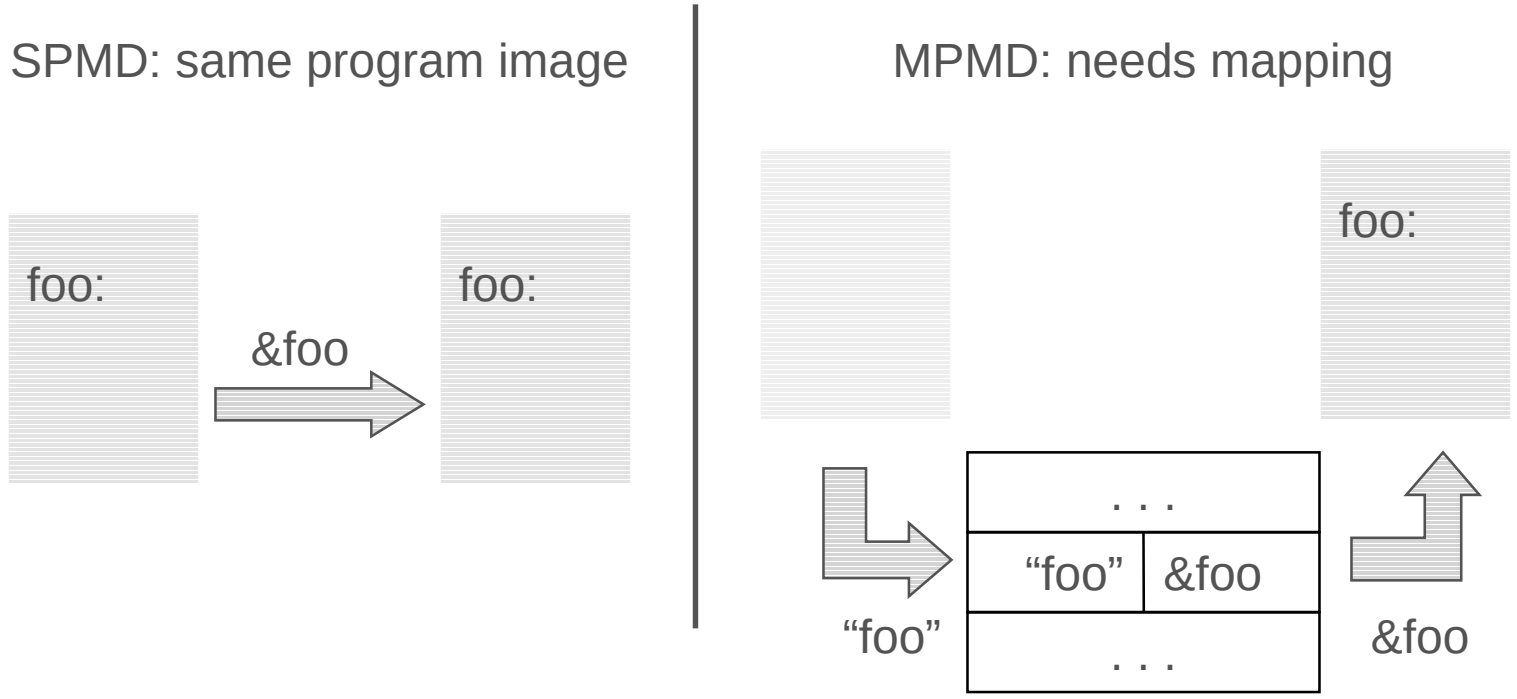
- relies on CC++ global pointers for RPC binding
- borrows RPC stub generation from CC++
- no modification to front-end compiler

Outline

- Design issues in MRPC
- MRPC and CC++
- Performance results

Method Name Resolution

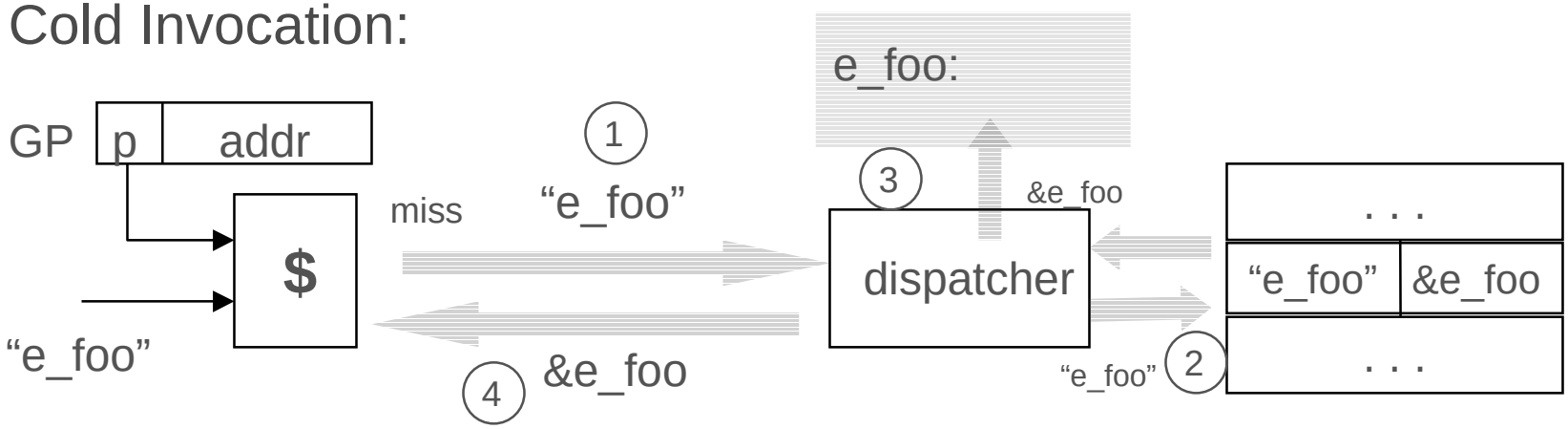
Compiler cannot determine the existence or location of a remote procedure statically



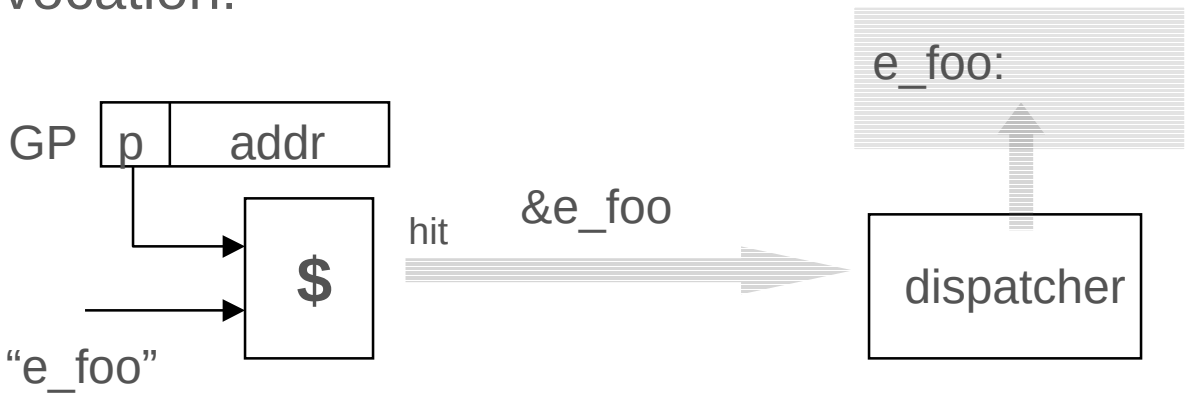
MRPC: sender-side stub address caching

Stub address caching

Cold Invocation:



Hot Invocation:



Argument Marshalling

Arguments of RPC can be arbitrary objects

- must be marshalled and unmarshalled by RPC stubs
- even more expensive in heterogeneous setting

versus…

- AM: up to 4 4-byte arguments, arbitrary buffers (programmer takes care of marshalling)

MRPC: efficient data copying routines for stubs

Data Transfer

Caller stub does not know about the receive buffer

- no caller/callee synchronization

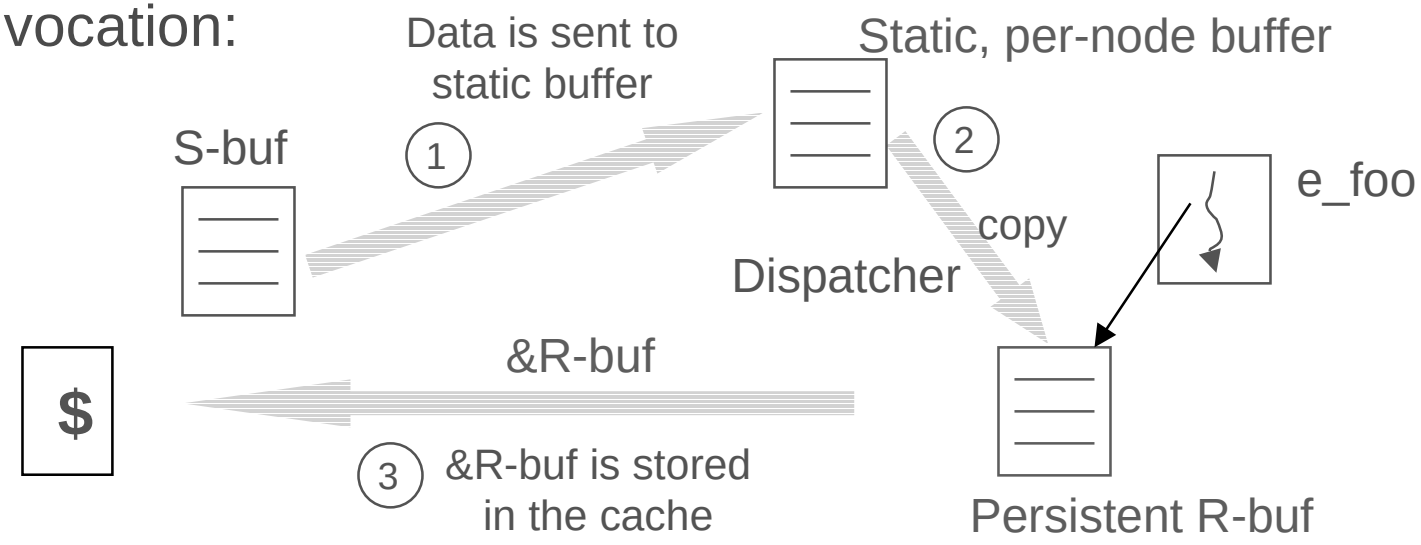
versus…

- AM: caller specifies remote buffer address

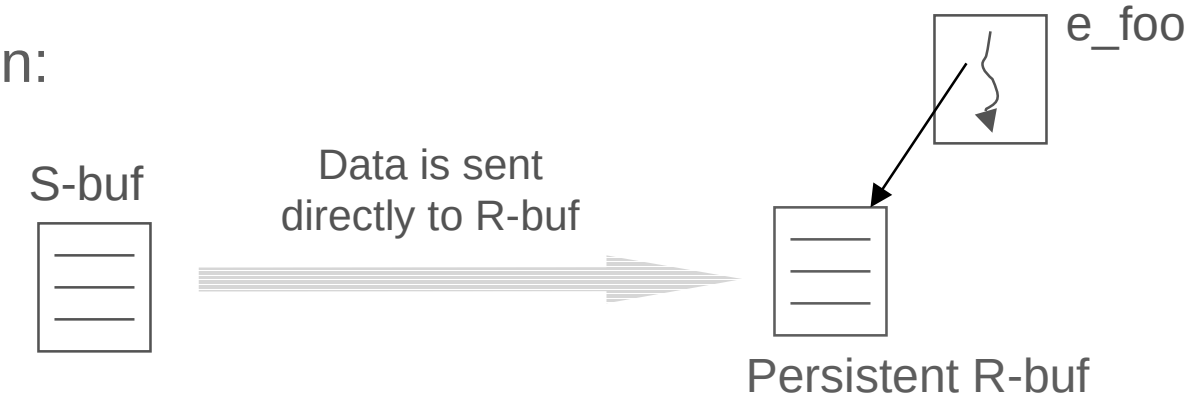
MRPC: Efficient buffer management and persistent receive buffers

Persistent Receive Buffers

Cold Invocation:



Hot Invocation:



Threads

Each RPC requires a new (logical) thread at the receiving end

- No restrictions on operations performed in remote procedures
- Runtime system must be thread safe

versus…

- Split-C: single thread of control per node

MRPC: custom, non-preemptive threads package

Message Reception

Message reception is not receiver-initiated

- Software interrupts: very expensive

versus…

- MPI: several different ways to receive a message (poll, post, etc)
- SPMD: user typically identifies comm phases into which cheap polling can be introduced easily

MRPC: Polling thread

CC++ over MRPC

CC++: caller

```
gpA->foo(p,i);
```

compiler

C++ caller stub

```
(endpt.InitRPC(gpA, “entry_foo”),
 endpt << p, endpt << i,
 endpt.SendRPC(),
 endpt >> retval,
 endpt.Reset());
```

CC++: callee

```
global class A {
   . . . };
double A::foo(int
   p, int i) {
   . . .}
```

compiler

C++ callee stub

```
A::entry_foo(. . .) {
   . . .
   endpt.RecvRPC(inbuf, . . . );
   endpt >> arg1; endpt >> arg2;
   double retval = foo(arg1, arg2);
   endpt << retval;
   endpt.ReplyRPC();
   . . .
}
```

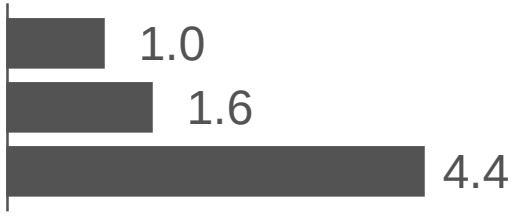
MRPC Interface

- InitRPC
- SendRPC
- RecvRPC
- ReplyRPC
- Reset

Micro-benchmarks

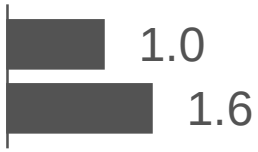
Null RPC:

AM:	55 us
CC++/MRPC:	87 us
Nexus/MPL:	240 µs (DCE: ~50 µs)



Global pointer read/write (8 bytes)

Split-C/AM:	57 µs
CC++/MRPC:	92 µs



Bulk read (160 bytes)

Split-C/AM:	74 µs
CC++/MRPC:	154 µs

IBM MPI-F and MPL (AIX 3.2.5): 88 us

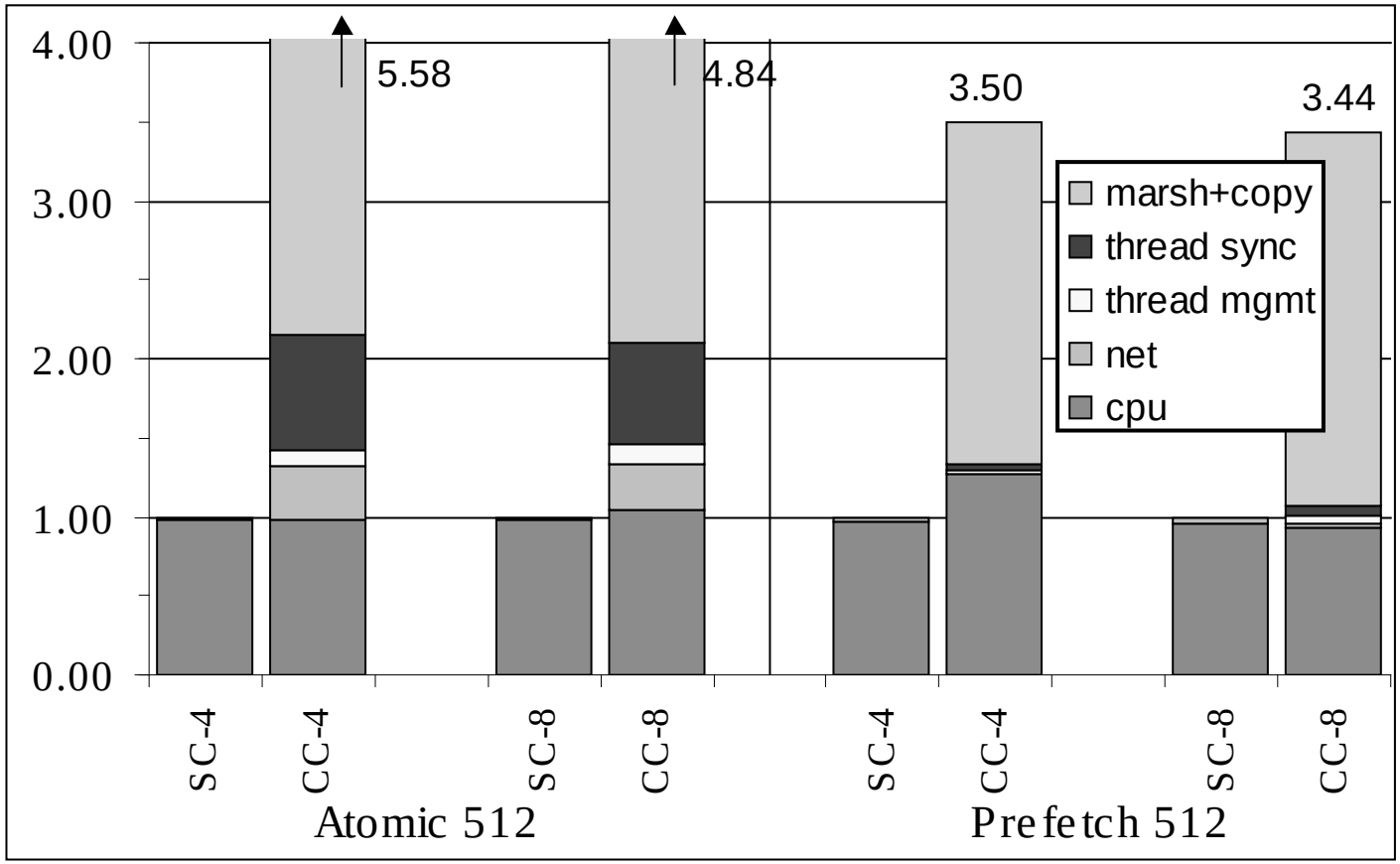
Basic comm costs in CC++/MRPC are within 2x with Split-C/AM and other messaging layers

Applications

- 3 versions of EM3D, 2 versions of Water, LU and FFT
- CC++ versions based on original Split-C code
- Runs taken for 4 and 8 processors on IBM SP-2

App	*Split-C/AM*	*CC++/Nexus*	*CC++/MRPC*
em3d-ghost 800	6.9 s	464 s (67.2x)	16.9 s (2.4x)
water-pref 512 mol	0.75 s	12.3 s (16.4x)	2.6 s (3.4x)
FFT 1M	0.78 s	23.1 s (29.6x)	2.8 s (3.6x)
LU 512	0.81 s	15.5 s (19.1x)	2.9 s (3.6x)

Water


4.00
3.00
2.00
1.00
0.00
5.58
4.84
3.50
3.44
marsh+copy
thread sync
thread mgmt
net
cpu
SC-4
CC-4
SC-8
CC-8
SC-4
CC-4
SC-8
CC-8
Atomic 512
Prefetch 512

Discussion

CC++ applications perform within a factor of 2 to 6 of Split-C

- order of magnitude improvement over previous impl

Method name resolution

- constant cost, almost negligible in apps

Threads

- accounts for ~25-50% of the gap, including:
  - synchronization (~15-35% of the gap) due to thread safety
  - thread management (~10-15% of the gap), 75% context switches

Argument Marshalling and Data Copy

- large fraction of the remaining gap (~50-75%)
- opportunity for compiler-level optimizations

Related Work

Lightweight RPC

- LRPC: RPC specialization for local case

High-Performance RPC in MPPs

- Concert, pC++, ABCL

Integrating threads with communication

- Optimistic Active Messages
- Nexus

Compiling techniques

- Specialized frame mgmt and calling conventions, lazy threads, etc. (Taura's PLDI '97)

Conclusion

Possible to implement an RPC-based MPMD system that is competitive with SPMD systems on homogeneous MPPs

- same order of magnitude performance
- trade-off between generality and performance

Questions remaining:

- scalability for larger number of nodes
- integration with heterogeneous runtime infrastructure

Slides: http://www.cs.cornell.edu/home/chichao

MRPC, CC++ apps source code: chichao@cs.cornell.edu