Mutually Assured Destruction† (or the Joy of Sync)

Peter Welch (phw@kent.ac.uk),
Jan Bækgaard Pedersen (matt.pedersen@unlv.edu)
Frederick R.M. Barnes (frmb@kent.ac.uk)

CPA 2013 Fringe, Napier University, 25 August, 2013

† plus non-blocking barriers and performance ...
The Joy of Sync

- Process oriented design ...
- Synchronous communications ...
- Synchronous barriers ...
- Mutually assured destruction ...
- Non-blocking barriers ...
- Performance ...
(a) a network of three processes, connected by four internal (hidden) and three external channels.

(b) three processes sharing the writing end of a channel to a server process.

(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.

(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).
(a) a network of three processes, connected by four internal (hidden) and three external channels.

CHAN BYTE a, b, c, d:
PAR
  foo (in?, left!, a?, b!, c!)
  bar (a!, b?, d!)
  merge (c?, d?, right!)
PROC thing (CHAN INT in?, left!, right!)
CHAN BYTE a, b, c, d:
PAR
  foo (in?, left!, a?, b!, c!)
  bar (a!, b?, d!)
  merge (c?, d?, right!)
:
PROC thing (CHAN INT in?, left!, right!)
CHAN BYTE a, b, c, d:
PAR
  foo (in?, left!, a?, b!, c!)
  bar (a!, b?, d!)
  merge (c?, d?, right!)
::
Like *foo*, *bar* and *merge* previously, *thing* is a process that can be used as a component in another network.

Concurrent systems have structure – networks within networks. We must be able to express this! And we can … 😊😊😊
(b) three processes sharing the writing end of a channel to a server process.

```
SHARED ! CHAN SOME SERVICE c:

PAR
  circle (c!)
  triangle (c!)
  square (c!)
  server (c?)
```
\[\text{SHARED CHAN ANOTHER.SERVICE} \ c:\ \text{PAR}\]
\[\begin{align*}
\text{PAR} \\
\text{circle} \ (c!) \\
\text{triangle} \ (c!) \\
\text{square} \ (c!) \\
\text{PAR} \ i = 0 \ \text{FOR} \ 8 \\
\text{s} \ (i, \ c?)
\end{align*}\]

(c) three processes sharing the writing end of a channel to a bank of servers sharing the reading end.
**BARRIER b:**

\[
\text{PAR } i = 0 \text{ FOR } n \text{ ENROLL } b \\
p(i, b)
\]

(d) n processes enrolled on a shared barrier (any process synchronising must wait for all to synchronise).
The Joy of Sync

- Process oriented design ...
- Synchronous communications ...
- Synchronous barriers ...
- Mutually assured destruction ...
- Non-blocking barriers ...
- Performance ...
Synchronised Communication

\[ (A(c) \parallel B(c)) \setminus \{c\} \]

**A** may **write** on \( c \) at any time, but has to wait for a **read**.

**B** may **read** from \( c \) at any time, but has to wait for a **write**.
Synchronised Communication

Only when both $A$ and $B$ are ready can the communication proceed over the channel $c$.

$\{A (c) \parallel B (c)\} \setminus \{c\}$
Synchronised Communication

- **Benefit**
  - Once the writer has written, it *knows* the reader has read

- **Careful**
  - Writer blocks if reader is not ready
  - Lots of deadlock possibilities

OK: plenty of other processes to run and ultra-fast context switch (comparable to a procedure call)

OK: work with (a small set of) synchronisation patterns for which we have proven safety theorems
If there is no discipline on when $A$ and $B$ communicate, then $A$ may commit to output on $c$, followed by $B$ on $d$ … or vice-versa. Either way, neither are listening and both are stuck. Same happens if both commit to input.
Client-Server Pattern

client: makes a request any time, then commits to taking reply.

server: always accepts a request (within some bounded time), then always makes a reply (within some bounded time). It may make requests itself, as a client to other servers.

No deadlock is now possible from this client-server relationship.
**Client-Server Pattern**

A client makes a **request** any time, then commits to taking a **reply**.

A server always accepts a **request** (within some bounded time), then always makes a **reply** (within some bounded time). It may make requests itself, as a **client** to other **servers**.

**Symbology**: this represents a client-server relation. It points to the server and allows a **2-way** conversation (initiated by the client).
A *server* may have many *clients* …

Only one *client* at a time converses with the *server*. They form an orderly queue. Still no deadlock possible – and no client starvation. No polling on the queue, so no livelock either.
Client-Server Theorem

A *client-server* system that has no cycles in its *client-server* relations is deadlock, livelock and starvation free.
The Joy of Sync

- Process oriented design
- Synchronous communications
- Synchronous barriers
- Mutually assured destruction
- Non-blocking barriers
- Performance
**Barriers**

The `occam-π BARRIER` type corresponds to a multiway CSP `event`, though some higher level design patterns (such as `resignation`) have been built in.

Basic CSP semantics apply. When a process `synchronises` on a barrier, it blocks until all other processes `enrolled` on the barrier have also `synchronised`. Once the barrier has completed (i.e. all `enrolled` processes have `synchronised`), all blocked processes are rescheduled for execution.
Barriers

The \texttt{occam-\pi BARRIER} type corresponds to a multiway \texttt{CSP event}, though some higher level design patterns (such as \texttt{resignation}) have been built in.

The number of processes enrolled on an in-scope barrier is unchanged by a \texttt{non-enrolling PAR} – but \textit{only one} of its components may reference it.

A \texttt{PAR} construct must \textit{explicitly} \texttt{ENROLL} its components on barriers.
Barriers

Processes may synchronise on more than one barrier:

To synchronise on a barrier:

- **BARRIER** b, c:
  
  PAR i = 0 FOR n ENROLL b, c

- worker (i, b, c)

or

- **SYNC** b

- **SYNC** c
Barriers

Barriers are commonly used to synchronise multiple *phases* of computation between a set of processes. Within each phase, other synchronisations (channel/barrier) may take place:

```c
PROC worker (VAL INT id, BARRIER b, c)
  ... local declarations / initialisation
  WHILE running
    SEQ
    SYNC b
    ... observe neighbourhood phase
    SYNC c
    ... change neighbourhood phase
  :
```

All workers do this together – all see the same thing ...

All workers do this together – may need to negotiate ...
Of course, only one barrier is actually needed to synchronise the phases in this example:

```plaintext
PROC worker (VAL INT id, BARRIER a) ...
  ... local declarations / initialisation
  WHILE running
    SEQ
      SYNC a
      ... observe neighbourhood phase
      SYNC a
      ... change neighbourhood phase
    :
```

But it's safer programming for each phase to be synchronised by its own barrier ...

All workers do this together – all see the same thing ...

All workers do this together – may need to negotiate ...
Barriers – Safety

**occam-π** BARRIER synchronisation is *safe* in the sense that enrollment and resignation are automatically managed. A process may synchronise on a BARRIER if and only if it is enrolled.

Try to break this rule … your program won’t compile. There are zero memory and run-time costs to enforce it. 😊
The Joy of Sync

Process oriented design ...
Synchronous communications ...
Synchronous barriers ...
Mutually assured destruction ...
Non-blocking barriers ...
Performance ...
Mutually Assured Destruction

Two processes are given, at the same time, their own task to complete; we are satisfied with the completion of either one of them; whichever process finishes first interrupts the other and reports its completion; the one that is interrupted abandons its task and reports that fact.

Such requirements are common in control systems, robotics, e-commerce, model-checking, …

- Drive rover vehicle forwards target meters.
- Look out for Martians.
- Stop and report when either is achieved.
- Drive rover vehicle forwards target meters.
- Look out for Martians.
- Stop and report when either is achieved.
PROTOCOL REPORT

CASE
  me    -- task completed
  she   -- task abandoned

PROTOCOL KILL

CASE
  kill

INT

sensor

command

report

monitor (mode)

killMe

killYou
PROC monitor (VAL INT mode, CHAN INT command?, sensor?,
  CHAN REPORT report!,
  CHAN KILL killYou!, killMe?)

WHILE TRUE
  PRI ALT
    INT target:
    command ? target              -- service requested
      service (mode, target, sensor?, report!,
        killYou!, killMe?)
    INT x:
    sensor ? x                     -- accept and discard
      SKIP

:
PROC service (VAL INT mode, target, CHAN INT sensor?,
    CHAN REPORT report!,
    CHAN KILL killYou!, killMe?)

  ...  local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running
  PRI ALT
    killMe ? kill
      ...  report 'she' and exit loop
    INT x:
    sensor ? x
      ...  process x
  :
killMe ? kill
SEQ
  report ! she
  running := FALSE
PROC service (VAL INT mode, target, CHAN INT sensor?,
CHAN REPORT report!,
CHAN KILL killYou!, killMe?)

... local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running

PRI ALT
  killMe ? kill
  ... report 'she' and exit loop
INT x:
sensor ? x
  ... process x
:
INT x:
sensor ? x
  SEQ
  ...
  update local state with x (depends on mode)
  IF
  ...
  task complete
  SEQ
  killYou ! kill
  report ! me
  running := FALSE
  TRUE
  SKIP
  :
PROC MADsystem (CHAN INT moveCommand?, searchCommand?,
CHAN INT motorSensor?, cameraSensor?,
CHAN REPORT moveReport!, searchReport!)

CHAN KILL a, b:
PAR
  monitor (move, moveCommand?, motorSensor?,
           moveReport!, b!, a?)
  monitor (search, searchCommand?, cameraSensor?,
           searchReport!, a!, b?)

:
Soak Testing

average sensor data interval = 10 ms (randomised)
average sensor inputs per service = 100 (randomised)

MADsystem

- motorSim
- cameraSim
- motorSensor
- cameraSensor
- moveReport
- moveCommand
- searchReport
- searchCommand
- controllerSim

Ran for 30 days (approx. 2.5m trials): P A S S E D
average sensor data interval = 10 ms (varying)
average sensor inputs per service = 100 (varying)

In Service

Ran for 2 years (approx. 64m trials): DEADLOCKED
Should have asked for a model check …

A trace leading to deadlock is provided:

<moveCommand, motorSensor, searchCommand, cameraSensor>
Should have asked for a model check …

A trace leading to deadlock is provided:

\[
<\text{moveCommand, motorSensor, searchCommand, cameraSensor}>
\]
PROC monitor (VAL INT mode, CHAN INT command?, sensor?,
CHAN REPORT report!,
CHAN KILL killYou!, killMe?)

WHILE TRUE
PRI ALT
    INT target:
    command ? target
    service (mode, target, sensor?, report!,
    killYou!, killMe?)

INT x:
    sensor ? x
    -- accept and discard
    SKIP

: 
PROC service (VAL INT mode, target, CHAN INT sensor?, 
CHAN REPORT report!, 
CHAN KILL killYou!, killMe?)

... local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running
PRI ALT
  killMe ? kill
    ... report 'she' and exit loop
INT x:
  sensor ? x
    ... process x
INT x:
  sensor ? x
SEQ
  \[\text{update local state with } x \text{ (depends on mode)}\]
IF
  \[\text{task complete}\]
SEQ
  \[\text{killYou} \land \text{kill}\]
  \[\text{report} \land \text{me}\]
  \[\text{running} := \text{FALSE}\]
TRUE
SKIP
:
If the kill windows of the two monitors overlap, both will try to kill the other – resulting in deadlock.
average sensor data interval = 10 ms (randomised)
average sensor inputs per service = 100 (randomised)

\[ \Rightarrow \]

average service time = 1 second
kill window = 100 nanoseconds (approx.)

\[ \Rightarrow \]

chance of kill window overlap (deadlock) = 1/10,000,000

time before 50% chance of deadlock = 90 days (approx.)

INT x:
sensor ? x
SEQ
  ...  update local state with x (depends on mode)
IF
  ...  task complete
  SEQ
    killYou ! kill
    report ! me
    running := FALSE
  TRUE
  SKIP
  :
chance of kill window overlap (deadlock) = 1/10,000,000

This assumes each monitor runs on its own dedicated core ...
CCSP multicore scheduler dynamically batches processes to cores ...
If monitors are in the same batch, they will not deadlock ...

chance of kill window overlap (deadlock) = 1/100,000,000 (approx.)

⇒
time before 50% chance of deadlock = 2 years (approx.)

Kill Window

```plaintext
INT x:
sensor ? x
SEQ
  ... update local state with x (depends on mode)
IF
  ... task complete
SEQ
  killYou ! kill
  report ! me
  running := FALSE
TRUE
  SKIP
:
```
Mutually Assured Destruction (revised implementation)

Communication between the monitors is mostly a one-way kill (from killer to killed). Deadlock happens when both turn killer – two-way communications.

Idea: make communication between the monitors always two-way – either a kill in both directions (should both tasks complete around the same time) or a kill in one direction followed by an ack in the other (which will be most of the time).

Claim: this eliminates all deadlock (at the cost of an extra ack).
Previously ...

PROTOCOL KILL
CASE

::

INT

sensor

monitor (mode)

killMe

killYou

INT

command

report

PROTOCOL REPORT
CASE

me -- task completed

she -- task abandoned

:
monitor' (mode)

INT

sensor

report

command

PROTOCOL REPORT'
CASE
me -- task completed
she -- task abandoned
both -- both completed

PROTOCOL KILL'
CASE
kill
ack

INT

killMe

killYou
PROC monitor' (VAL INT mode, CHAN INT command?, sensor?, CHAN REPORT' report!, CHAN KILL' killYou!, killMe?)

WHILE TRUE
  PRI ALT
  INT target:
  command ? target              -- service requested
  service' (mode, target, sensor?, report!, killYou!, killMe?)

  INT x:
  sensor ? x                    -- accept and discard
  SKIP
  :
PROC service (VAL INT mode, target, CHAN INT sensor?,
   CHAN REPORT report!,
   CHAN KILL killYou!, killMe?)

... local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running
   PRI ALT
   killMe ? kill
   ... report ‘she’ and exit loop
   INT x:
   sensor ? x
   ... process x

::
PROC service’ (VAL INT mode, target, CHAN INT sensor?,
   CHAN REPORT’ report!,
   CHAN KILL’ killYou!, killMe?)

... local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running
PRI ALT
   killMe ? kill
      ... 'ack' the kill, report 'she' and exit loop
   INT x:
      sensor ? x
      ... process x
Previously ...

\[
\text{killMe} \ ? \ \text{kill} \\
\text{SEQ} \\
\quad \text{report} \ ! \ \text{she} \\
\quad \text{running} \ := \ \text{FALSE}
\]
```
killMe ? kill
SEQ
  killYou ! ack
  report ! she
running := FALSE
```
PROC service’ (VAL INT mode, target, CHAN INT sensor?,
    CHAN REPORT’ report!,
    CHAN KILL’ killYou!, killMe?)

... local state and initialisation
INITIAL BOOL running IS TRUE:
WHILE running

PRI ALT
killMe ? kill
    ... 'ack' the kill, report 'she' and exit loop
INT x:
sensor ? x
    ... process x
:
INT x:
sensor ? x
SEQ
  ... update local state with x (depends on mode)
IF
  ... task complete
  SEQ
  killYou ! kill
  report ! me
  running := FALSE
TRUE
  SKIP
::
Each process knows what happened in the other – potentially a very useful side benefit.

```plaintext
INT x:
sensor ? x
SEQ
  ... update local state with x (depends on mode)
IF
  ... task complete
  SEQ
  PAR
  killYou ! kill               -- send and
  killMe ? CASE                -- receive in parallel
  ack
  report ! me
  kill
  report ! both
  running := FALSE
TRUE
  SKIP
:```

Copyright P.H.Welch, (2013)
Key state information could easily be piggy-backed on the kill and ack signals – i.e. each process would know what the other found.

```
INT x:
sensor ? x
SEQ
  ...  update local state with x (depends on mode)
IF
  ...  task complete
  SEQ
  PAR
    killYou ! kill       -- send and
    killMe ? CASE       -- receive in parallel
    ack
    report ! me
    kill
    report ! both
    running := FALSE
  TRUE
  SKIP
:```
Better ask for a model check ...

VERIFY DEADLOCK.FREE MADsystem'
Soak Testing

**MADsystem'**

- motorSim
- cameraSim
- motorSensor
- cameraSensor
- moveReport
- moveCommand
- searchReport
- searchCommand
- controllerSim

Only for confidence boosting – it will not deadlock (assuming compiler, run-time kernel, microprocessor are OK)
In Service

MADsystem

motorSensor

moveReport

moveCommand

searchReport

searchCommand

cameraSensor

This will not deadlock
(assuming compiler, run-time kernel, microprocessor are OK)
Mutually Assured Destruction (asynchronous channels?)

If the channels were finitely buffered (with capacity greater than zero), the deadlock found with synchronous (i.e. zero-buffered) channels would not happen – both monitors would complete their kills, reports and service routines.
Mutually Assured Destruction (asynchronous channels?)

If the channels were finitely buffered, deadlock is still possible – but less likely (exponentially) with increasing buffer size. Infinitely expandable buffer capacity would be needed to eliminate deadlock from the basic algorithm. For practical purposes, I would feel safe with a capacity of 3.
Mutually Assured Destruction (asynchronous channels?)

However, there is a nasty problem. If both monitors send a **kill**, neither is taken and they remain lurking in the buffered channels. Some time in the next service cycle, both will strike and the services will be erroneously aborted.
Mutually Assured Destruction (asynchronous channels?)

This could be overcome by counting cycles and sequence numbering the kill signals: just ignore any kill with a number less than the current count. This adds complexity and run-time overhead.
Mutually Assured Destruction (asynchronous channels?)

Further, this only works if the processes engaged in MAD are in lock-step (which they are in this scenario, but not in general).

This could be overcome by counting cycles and sequence numbering the kill signals: just ignore any kill with a number less than the current count.
Mutually Assured Destruction (asynchronous channels?)

Alternatively, the mess could be sorted out by the Controller process. When/if it gets two me reports from the monitors, it tells each monitor (as part of its next command) to read and discard an incoming kill. Again, this adds complexity – we shouldn’t have a mess to clean up!
Mutually Assured Destruction (asynchronous channels?)

Alternatively, the mess could be sorted out by the Controller process. When/if it gets two **me** reports from the monitors, it tells each monitor (as part of its next command) to read and discard an incoming **kill**. Further, this assumes a Controller, which processes engaged in MAD may not have.
The Joy of Sync

- Process oriented design ...
- Synchronous communications …
- Synchronous barriers …
- Mutually assured destruction …
- Non-blocking barriers …
- Performance …
Non-Blocking Barriers

Recently (2012) introduced to MPI, non-blocking barrier synchronisation seems, at first glance, a contradiction of terms … the whole point of a barrier is to block until all parties are there!

When we have completed our work before a barrier, we normally synchronise on it – thereby notifying that we are there and waiting for the others.

Recall …
Barriers

Processes may synchronise on more than one barrier:

```
worker (0)  worker (1)  ...  worker (n-1)
```

To synchronise on a barrier:

```
BARRIER b, c:
PAR i = 0 FOR n ENROLL b, c
worker (i, b, c)
```

```
SYNC b  or  SYNC c
```
Barriers

Barriers are commonly used to synchronise multiple phases of computation between a set of processes. Within each phase, other synchronisations (channel/barrier) may take place:

```prolog
PROC worker (VAL INT id, BARRIER b, c)
  ... local declarations / initialisation
  WHILE running
  SEQ
    SYNC b
    ... observe neighbourhood phase
    SYNC c
    ... change neighbourhood phase
```

All workers do this together – all see the same thing ...

All workers do this together – may need to negotiate ...
Non-Blocking Barriers

Recently (2012) introduced to MPI, *non-blocking barrier synchronisation* seems, at first glance, a contradiction of terms … the whole point of a *barrier* is to *block* until all parties are there!

When we have completed our work before a *barrier*, we normally synchronise on it – thereby notifying that we are there and waiting for the others.

However, if there is something we can be getting on with that does not disturb our fellow *synchronisers*, (e.g. preparatory work for the phase following the *barrier*), it would be good to be able to do so. Only when we need stuff that depends on the other *synchronisers*, should we have to wait for them.
Blocking Barrier Sync (MPI)

... phase 0 computation
MPI_Barrier (b, ...); // wait for everyone ...
... preparatory work for next phase
... phase 1 computation

Blocking Barrier Sync (occam-π)

SEQ
... phase 0 computation
SYNC b -- wait for everyone ...
... preparatory work for next phase
... phase 1 computation
Non-Blocking Barrier Sync (MPI)

... phase 0 computation
MPI_IBarrier (b, ...); // hey, I’m done ...
... preparatory work for next phase
MPI_WBarrier (b, ...); // I’m waiting now ...
... phase 1 computation

Non-Blocking Barrier Sync (\(0-\pi\))

SEQ
... phase 0 computation
PAR
SYNC b -- hey, I’m done ...
... preparatory work for next phase
... phase 1 computation
Non-Blocking Barrier Sync ($0-\pi$)

SEQ

... phase 0 computation

PAR

SYNC b -- hey, I'm done ...

... preparatory work for next phase

... phase 1 computation

The **SYNC** registers that we have arrived at the barrier and lets all move forward when the rest arrive. In parallel with the above, we get on with our **preparatory work**. ☺

When our **preparatory work** is complete, if all the others have reached the barrier, the **SYNC** will have completed – so the **PAR** completes and we immediately move on to **phase 1**. And we have not delayed the others. ☻

When our **preparatory work** is complete, if not all the others have reached the barrier, the **SYNC** will not have completed. We wait for the others at the **SYNC** before moving on to **phase 1** – as we must! ☻
Non-Blocking Barrier Sync (0-π)

SEQ
... phase 0 computation
PAR
SYNC b
-- hey, I’m done ...  
... preparatory work for next phase
... phase 1 computation

The **SYNC** registers that we have arrived at the barrier and lets all move forward when the rest arrive. In parallel with the above, we get on our **preparatory work**. 😊

When our **preparatory work** is complete, if all the others have reached the barrier, the **SYNC** will have completed so the **PAR** completes and we immediately move on to **phase 1**. And we have not delayed the others. 😊

When our **preparatory work** is complete, if not all the others have reached the barrier, the **SYNC** will not have completed. We wait for the others at the **SYNC** before moving on to **phase 1** – as we must! 😊
The Joy of Sync

Process oriented design ...
Synchronous communications ...
Synchronous barriers ...
Mutually assured destruction ...
Non-blocking barriers ...
Performance ...
Take a ring of \( n \) processes …

Each process connects to all …

In parallel, each process sends and receives \( m \) messages (e.g. its \textit{id number}) to all, including itself …

That’s \( mn^2 \) messages …

How long per message?
Andrew’s “Say Hello to Everyone” Benchmark (v1) *

```scala
[n][n]CHAN INT c:

PAR i = 0 FOR n
  PAR
    PAR j = 0 FOR n
      SEQ k = 0 FOR m
        c[i][j] ! i
      PAR j = 0 FOR n
      SEQ k = 0 FOR m
        INT x:
        SEQ
          c[j][i] ? x
        ASSERT (x = j) -- sanity check
```

---

2n^2 processes

mn^2 messages
Performance

Andrew’s “Say Hello to Everyone” Benchmark (v2) *

[n][n]CHAN INT c:

PAR i = 0 FOR n
PAR
SEQ j = 0 FOR n
SEQ k = 0 FOR m
    c[i][j] ! i
SEQ j = 0 FOR n
SEQ k = 0 FOR m
    INT x:
    SEQ
    c[j][i] ? x
    ASSERT (x = j) -- sanity check

2n processes
mn^2 messages

Take a ring of $N$ processes …

Each process connects to all processes in parallel, each process sends and receives $m$ messages (e.g. its ID) to all, including itself …

That’s $mn^2$ messages …

How long per message?

The following observations were made using KRoC 1.5.0-pre5, Ubuntu 11.04 (natty) on an Intel i7 quad-core processor with hyperthreading (i.e. 8 virtual cores), running at 2 MHz.

The benchmark timings were averaged from 10 separate runs, with negligible variance.
Take a ring of \( N \) processes:

- Each process connects to all processes.
- In parallel, each process sends and receives \( m \) messages (e.g., its ID) to all, including itself.
- That's \( m^2 \) messages.

**Performance**

\[
P(0) = P(1)P(n-1) = P(2)P(3)
\]

1 core / 20,000 nodes / v2

- CPU History:
  - CPU1: 11.9%
  - CPU2: 0.0%
  - CPU3: 44.4%
  - CPU4: 57.4%
  - CPU5: 4.0%
  - CPU6: 5.0%
  - CPU7: 4.8%
  - CPU8: 1.0%

- Memory and Swap History:
  - Memory: 6.2 GB (80.9%) of 7.7 GB
  - Swap: 0 bytes (0.0%) of 7.9 GB

- Network History:
  - Receiving: 179 bytes/s
  - Sending: 0 bytes/s
  - Total Received: 42.9 MiB
  - Total Sent: 10.8 MiB
In parallel, each process sends and receives messages (e.g. its id) to all, including itself.

Take a ring of \( N \) processes.

Each process connects to all of its neighbors, \( P(0), \ldots, P(n-1) \).

How long per message?

Performance:

\[
P(0) = \text{2 cores / 20,000 nodes / v2}
\]

\[
P(1) = \text{2 cores / 20,000 nodes / v2}
\]

\[
P(n) = \text{2 cores / 20,000 nodes / v2}
\]
Take a ring of $N$ processes.

Each process connects to all processes in parallel, each process sends and receives $m$ messages (e.g. its id) to all, including itself. That's $mN$ messages.

How long per message?

Performance

$P(1)$

$P(n-1)$

$P(0)$

$P(2)$

$P(3)$

4 cores / 20,000 nodes / v2
Take a ring of $N$ processes. Each process connects to all and receives $m$ messages (e.g., its id) to all, including itself. That's $mn^2$ messages. How long per message?

**Performance**

$P(0)$

$P(1)$

$P(n-1)$

8 cores / 20,000 nodes / v2

- **CPU History**
  - CPU1 100.0%
  - CPU2 100.0%
  - CPU3 100.0%
  - CPU4 100.0%
  - CPU5 100.0%
  - CPU6 100.0%
  - CPU7 99.0%
  - CPU8 100.0%

- **Memory and Swap History**
  - Memory: 6.2 GiB (80.7%) of 7.7 GiB
  - Swap: 0 bytes (0.0%) of 7.9 GiB

- **Network History**
  - Receiving: Total Received: 42.6 MiB
  - Sending: Total Sent: 9.5 MiB

25-Aug-2013

Copyright P.H.Welch, (2013)
So … how long per message?

6,000 nodes …
(V2) 12,000 processes …
30 messages node-to-node …
1.08 billion messages …
8 cores …

7 nanoseconds
So … how long per message?

10,000 nodes …
(V2) 20,000 processes …
30 messages node-to-node …
3 billion messages …
8 cores …

7 nanoseconds
So … how long per message?

- 20,000 nodes …
- (V2) 40,000 processes …
- 30 messages node-to-node …
- 12 billion messages …
- 8 cores …

8 nanoseconds
So … how long per message?

6,000 nodes …
(V2) 12,000 processes …
30 messages node-to-node …
1.08 billion messages …
8 cores …

7 nanoseconds
So ... how long per message?

6,000 nodes ...

(V1) 72 million processes* ...

30 messages node-to-node ...

1.08 billion messages ...

8 cores ...

14 nanoseconds

* can't get many more than this (only 4 Gbytes currently addressable).
So ... how long per message?

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>per message (nanoseconds)</td>
<td>34.5</td>
<td>17.6</td>
<td>10.2</td>
<td>7.0</td>
</tr>
<tr>
<td>speed up</td>
<td>1.0</td>
<td>2.0</td>
<td>3.4</td>
<td>4.9</td>
</tr>
</tbody>
</table>
Performance

So ... how long per message?

10,000 nodes ...
(V2) 20,000 processes ...
30 messages node-to-node ...
3 billion messages ...

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>per message (nanoseconds)</td>
<td>37.5</td>
<td>17.4</td>
<td>10.3</td>
<td>7.2</td>
</tr>
<tr>
<td>speed up</td>
<td>1.0</td>
<td>2.2</td>
<td>3.6</td>
<td>5.2</td>
</tr>
</tbody>
</table>
So ... how long per message?

20,000 nodes ...
(V2) 40,000 processes ...
30 messages node-to-node ...
12 billion messages ...

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>per message (nanoseconds)</td>
<td>61.8</td>
<td>20.7</td>
<td>11.3</td>
<td>7.7</td>
</tr>
<tr>
<td>speed up</td>
<td>1.0</td>
<td>3.0</td>
<td>5.5</td>
<td>8.0</td>
</tr>
</tbody>
</table>
So ... how long per message?

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>per message (nanoseconds)</td>
<td>34.5</td>
<td>17.6</td>
<td>10.2</td>
<td>7.0</td>
</tr>
<tr>
<td>speed up</td>
<td>1.0</td>
<td>2.0</td>
<td>3.4</td>
<td>4.9</td>
</tr>
</tbody>
</table>

6,000 nodes ...
(V2) 12,000 processes ...
30 messages node-to-node ...
1.08 billion messages ...

Performance

\[ P(n) = P(n-1) \]

Repeat.
6,000 nodes …
(V1) 72 million processes* …
30 messages node-to-node …
1.08 billion messages …

So … how long per message?

<table>
<thead>
<tr>
<th># cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>per message (nanoseconds)</td>
<td>120.0</td>
<td>47.1</td>
<td>26.5</td>
<td>14.0</td>
</tr>
<tr>
<td>speed up</td>
<td>1.0</td>
<td>2.5</td>
<td>4.5</td>
<td>8.6</td>
</tr>
</tbody>
</table>

* can’t get many more than this
(only 4 Gbytes currently addressable).
The Joy of Sync

Process oriented design ...
Synchronous communications ...
Synchronous barriers ...
Mutually assured destruction ...
Non-blocking barriers ...
Performance ...

Any questions?