CAMP: **Cost-Aware Multiparty Session Protocols**

DAVID CASTRO-PEREZ and NOBUKO YOSHIDA, Imperial College London, United Kingdom

This paper presents CAMP, a new static performance analysis framework for message-passing concurrent and distributed systems, based on the theory of *multiparty session types* (MPST). Understanding the runtime performance of concurrent and distributed systems is of great importance for the identification of bottlenecks and optimisation opportunities. In the message-passing setting, these bottlenecks are generally *communication overheads* and *synchronisation times*. Despite its importance, reasoning about these *intensional* properties of software, such as performance, has received little attention, compared to verifying *extensional* properties, such as correctness. Behavioural protocol specifications based on sessions types capture not only extensional, but also intensional properties of concurrent and distributed systems. CAMP augments MPST with annotations of *communication latency* and *local computation cost*, defined as estimated execution times, that we use to extract cost equations from protocol descriptions. CAMP is also extendable to analyse *asynchronous communication optimisation* built on a recent advance of session type theories. We apply our tool to different existing benchmarks and use cases in the literature with a wide range of communication protocols, implemented in C, MPI-C, Scala, Go, and OCaml. Our benchmarks show that, in most of the cases, we predict an upper-bound on the real execution costs with < 15% error.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Software performance**; **System description languages**;

Additional Key Words and Phrases: parallel programming, session types, cost models, message optimisations

ACM Reference Format:
David Castro-Perez and Nobuko Yoshida. 2020. CAMP: Cost-Aware Multiparty Session Protocols. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 155 (November 2020), 30 pages. https://doi.org/10.1145/3428223

1 INTRODUCTION

Understanding the amount of resources, e.g. time or memory that are required by a computation, is of great importance. Correct but slow-performing software can cause a number of problems, ranging from the unnecessary use of resources, to exploitable security vulnerabilities. Worse still, performance issues are very difficult to detect in runtime because of their non fail-stop nature; and although the root causes of performance bugs can be very diverse, *uncoordinated functions* and *synchronisation issues* are prevalent, i.e. inefficient composition of efficient functions, and unnecessary synchronisation that increases thread competition [Jin et al. 2012]. These inefficient compositions have more impact in a distributed setting, where the *communication overhead* and *synchronisation cost* may become the bottleneck of the whole system.

The development of new static performance analysis tools will reduce the impact of bad performing software, by allowing the identification of their bottlenecks and optimisation. Further, for concurrency and distribution, such a tool must take into account communication and synchronisation overheads. This paper presents a new static performance analysis framework, CAMP

Authors' address: David Castro-Perez, d.castro-perez@imperial.ac.uk; Nobuko Yoshida, n.yoshida@imperial.ac.uk, Imperial College London, Computing, 180 Queen's Gate, London, SW7 2AZ, United Kingdom.

2475-1421/2020/11-ART155
https://doi.org/10.1145/3428223

(*Cost-Aware Multiparty Protocols*), that can identify potential performance bottlenecks in concurrent and distributed systems. Specifically, CAMP addresses the following two main challenges: *non-determinism* and *practicality*. Firstly, the non-deterministic nature of concurrent and distributed systems makes it hard to reason statically about the performance of alternative interleavings of actions in a program trace; and secondly, making the performance analysis practically useful for already existing implementations is not trivial.

CAMP solves the non-determinism issue by building on top of *multiparty session types* (MPST) [Coppo et al. 2015; Honda et al. 2008]. MPST is a well-established theory that describes not only *extensional*, but also *intensional* information about communicating systems. Specifically, MPST captures the communication structure, or *protocol* among distributed peers. Protocols appear not only in distributed networks but also in parallel multicore programming as *patterns* or *topologies* [Goetz et al. 2006; Lea 1997; Rauber and Rünger 2010; Taubenfeld 2006]. MPST uses *global types* for describing such protocols from a global point of view, and can be used to ensure *deadlock-freedom* and *session fidelity*: every send has a matching receive, and every component of the system complies with its part in the global protocol. Built on the MPST theory, CAMP enables the protocol-based performance analysis, giving a precise abstraction as (correct) communication structures of programs. By tying the analysis to a particular protocol specification that is statically enforced on the system, CAMP solves the issue of non-determinism.

On the practical side, since all we require is a global type, CAMP can be readily applied to existing implementations, as long as they are proven to comply with a known global type. We show this by taking existing benchmarks, either implemented using MPST-based tools, or following a known protocol. Different extensions of the core MPST have been already used to implement a wide range of applications written in different programming languages through several transports and architectures e.g. [Castro et al. 2019; Castro-Perez and Yoshida 2020b; Gay and Ravara 2017; Hu and Yoshida 2017; Imai et al. 2020; Ng et al. 2015], and our methodology is easily adaptable to these variants. In addition, not only CAMP is immediately usable for analysis of representative parallel patters [Asanovic et al. 2009; Krommydas et al. 2016; Rauber and Rünger 2010], but also it is applicable to Savina benchmarks [Imam and Sarkar 2014] or multicore algorithms which incur more complex patterns and synchronisations (§8.2).

The key notion in CAMP is that of execution *cost*: the amount of time that it takes a protocol, participant or function to run from beginning to end. To statically compute execution costs for concurrency and distribution, CAMP extends global types with *sizes* for values of messages (encoded in the payload types) and *local computation cost* information. This size and cost information can be obtained via profiling, or further static analysis, such as using *sized-types* [Hughes et al. 1996]. Our cost models take these extended protocols, and compute a set of equations which describe the total cost of each participant. These measurements provide us with fine-grained information to obtain communication overhead and synchronisation cost among participants of a protocol. For recursive protocols, CAMP produces a set of *recurrence equations* that describe the total cost after each iteration of the protocol. For non-terminating protocols (e.g. streaming computation split in multiple stages), CAMP computes the *latency*, or the average cost per iteration.

CAMP enables to quantify the performance gain of *asynchronous communication optimisation*. We evaluate this using non-optimised and optimised benchmarks. The optimisation analysis by CAMP is grounded on *asynchronous session subtyping*, which is one of the most advanced session types theories in the literature, and has been actively studied over a decade using various different formalisms, e.g., *first and higher-order mobile processes* [Chen et al. 2017, 2014; Ghilezan et al. 2021; Mostrous and Yoshida 2009, 2015; Mostrous et al. 2009], *denotational semantics* [Demangeon and Yoshida 2015; Dezani-Ciancaglini et al. 2016] and *automata theories* [Bravetti et al. 2019, 2017, 2018; Lange and Yoshida 2017].

Contributions. We present a compile-time performance analysis framework, CAMP, for concurrent/distributed systems that infers upper execution cost bounds of multiparty session protocols. The cost models in CAMP are parametric, and can combine both static and dynamic (e.g., profiling) information to produce accurate results. We prove that the cost analysis by CAMP is sound with respect to the operational semantics of a given global type instrumented with sizes and execution costs; and extensible to analyse communication optimisations. Our main contributions are:

a) we define the semantics of CAMP, integrating *global* and *local type semantics* with *local computation costs*, that can be used to explore the costs of particular traces (§3);
b) we instrument *global types* with size and local cost information, and use it to statically estimate an upper bound of the execution cost of a protocol, that we prove sound with respect to the operational semantics (§4);
c) we define multiple metrics on the *cost recurrences* associated with recursive protocols, that can be used to effectively analyse the performance behaviour of potentially infinite executions (§5);
d) we extend CAMP to handle *asynchronous message optimisations*, enabling us to statically quantify the potential performance gains when performing such optimisations/reordering (§6);
e) we implement a DSL for specifying global types, from which we can extract cost equations (§7), and we compare our cost model predictions with real benchmarks used in MPST implementations in different languages: C-MPI [Ng et al. 2015], C+pthreads [Castro-Perez and Yoshida 2020b], Go [Castro et al. 2019], OCaml [Imai et al. 2020] and F⋆ [Zhou et al. 2020]. Additionally, we apply CAMP to a subset of the Savina benchmarking suite (Scala) [Imam and Sarkar 2014]. These benchmarks include examples of common, and complex topologies, such as ring, butterfly and a double-buffering protocol (§8).

§9 discusses related work and §10 concludes the paper. See Castro-Perez and Yoshida [2020a] for additional definitions and full proofs. The git repository https://github.com/camp-cost/camp provides a working prototype implementation, described in §7 and the data used in §8, with instructions for replicating our experiments.

2 OVERVIEW

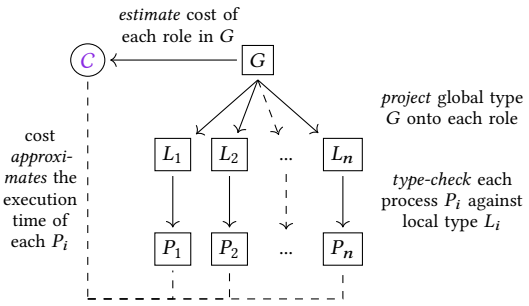


Fig. 1. CAMP framework

MPST ***Basics.*** We first explain how MPST satisfies extensional properties. Fig. 1 depicts the standard top-down methodology of MPST enhanced with cost-analysis, which we illustrate by a simple *scatter-gather* example between two Masters ($\mathtt{m}_1$, $\mathtt{m}_2$) and two Workers ($\mathtt{w}_1$, $\mathtt{w}_2$).

$$G = \mathtt{m}_1 \to \mathtt{w}_1\{\tau_1\}.\mathtt{m}_1 \to \mathtt{w}_2\{\tau_1\}.\\ \mathtt{w}_1 \to \mathtt{m}_2\{\tau_2\}.\mathtt{w}_2 \to \mathtt{m}_2\{\tau_2\}$$

G is a *global type*: a specification of the protocol between participants from a global perspective. G says master $\mathtt{m}_1$ first sends a message with type τ_1 to worker $\mathtt{w}_1$ then to worker $\mathtt{w}_2$, and finally master $\mathtt{m}_2$ collects a message with type τ_2 from each worker. For each participant $\mathtt{p}$, the global type is projected to a *local type*, which describes localised send and receive actions from $\mathtt{p}$ viewpoint:

$$L_1 = \mathtt{w}_1 \,!\, \tau_1.\mathtt{w}_2 \,!\, \tau_1.\mathsf{end} \qquad L_2 = \mathtt{m}_1 \,?\, \tau_1.\mathtt{m}_2 \,!\, \tau_2.\mathsf{end} \qquad L_3 = \mathtt{w}_1 \,?\, \tau_2.\mathtt{w}_2 \,?\, \tau_2.\mathsf{end}$$

L_1 says $\mathtt{m}_1$ should send (!) a τ_1 message to $\mathtt{w}_1$, then to $\mathtt{w}_2$, while L_2 says $\mathtt{w}_1$ receives (?) a τ_1 message from $\mathtt{m}_1$, followed by sending a τ_2 message to $\mathtt{m}_2$ (worker $\mathtt{w}_2$ has the same type L_2). Local types

are used to statically check local programs P_i implementing these types, i.e. the communication structures of each program complies with their local type. A well-typed system of programs is guaranteed free from deadlock and type errors, following the protocol given by G (session fidelity).

Cost-Aware MPST**.** Now we consider the cost to run G: $\mathtt{m}_1 \to \mathtt{w}_1\{\tau_1 \diamond c_1\}.\mathtt{m}_1 \to \mathtt{w}_2\{\tau_1 \diamond c_1\}.\mathtt{w}_1 \to \mathtt{m}_2\{\tau_2\}.\mathtt{w}_2 \to \mathtt{m}_2\{\tau_2\}$ Here c_1 represents the *local computation cost* at the receiver side. In this example, we are assuming the computational cost at $\mathtt{w}_1$ and $\mathtt{w}_2$ is c_1, while such cost at $\mathtt{m}_2$ is 0. Another factor we should take into account is the *communication cost*, which is parameterised by types, i.e. the time required for sending ($\mathtt{c}_\mathtt{O}(\tau)$) and receiving ($\mathtt{c}_\mathtt{I}(\tau)$) a value of type τ.

We assume our transport is *asynchronous*, i.e. sending is non-blocking and the order of messages are preserved, like TCP communications, hence there is no communication ordering between $\mathtt{w}_1$ and $\mathtt{w}_2$. Both workers can process the values independently at two different locations or CPUs. Then, the total execution cost at $\mathtt{m}_1$, $\mathtt{w}_1$ and $\mathtt{w}_2$ are:

$$\mathtt{m}_1 \mapsto 2 \times \mathtt{c}_\mathtt{O}(\tau_1) \quad \mathtt{w}_1 \mapsto \mathtt{c}_\mathtt{O}(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_\mathtt{O}(\tau_2) \quad \mathtt{w}_2 \mapsto 2 \times \mathtt{c}_\mathtt{O}(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_\mathtt{O}(\tau_2)$$

To consider the cost for $\mathtt{m}_2$, we should take care of the dependencies in the protocol. Each $\mathtt{w}_i$ can operate in parallel, and they exhibit almost the same cost. The only difference is that worker $\mathtt{w}_1$ can perform its computation as soon as $\mathtt{m}_1$ sends one message, but worker $\mathtt{w}_2$ can only proceed after $\mathtt{m}_1$ sends the second message. This difference means that $\mathtt{m}_2$ can start gathering one of the messages, while the other worker finishes its actions, which will be delayed by the time it takes $\mathtt{m}_1$ to send one message. Hence the cost of $\mathtt{w}_2$ is:

$$\mathtt{m}_2 \mapsto \max(\mathtt{c}_\mathtt{I}(\tau_2), \mathtt{c}_\mathtt{O}(\tau_1)) + \mathtt{c}_\mathtt{O}(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_\mathtt{O}(\tau_2) + \mathtt{c}_\mathtt{I}(\tau_2)$$

In §4, we shall prove the cost calculated based on local types and global types semantics coincide.

In many scenarios, we do not know how many iterations recursive protocols are going to run, or this number of iterations is large. In such cases, computing the cost of the protocol is not useful or meaningful. In such scenarios, we calculate the average cost per iteration of a protocol (*latency*) from a set of recurrences. From this latency, we calculate other useful metrics, such as the latency divided by the number of messages exchanged per iteration by participant (*latency relative to a particular participant*). The latency relative to a participant is used to estimate how much work can a participant do per iteration of the protocol.

3 COST-AWARE MULTIPARTY SESSION PROTOCOLS

This section introduces cost-aware multiparty session protocols (CAMP) which is an extension of multiparty session types (MPST) [Coppo et al. 2015; Deniélou and Yoshida 2013; Honda et al. 2008] where the payload types (τ) are types that have been extended with *size annotations*, adapted from the literature on *sized types* [Avanzini and Dal Lago 2017; Hughes et al. 1996], and interactions have been extended with *cost annotations* (c) which represent the local execution time at the receiver:

$$\tau ::= \mathsf{int}^i \mid \cdots \mid D^i\ \vec{\tau}$$
$$c ::= i \mid k \mid c + c \mid \max(c, c) \mid k \times c$$

Our types are base types (integer, boolean, …) annotated with a size i, type constructors D applied

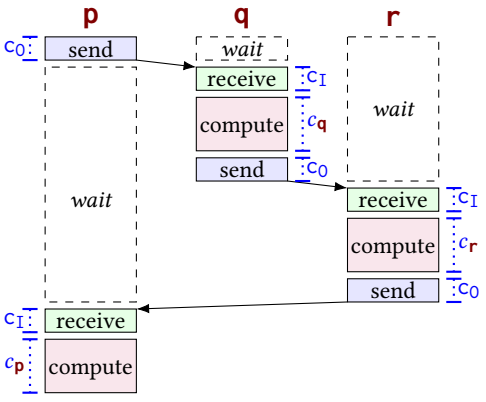


Fig. 2. A ring protocol, and an execution trace.

to a sequence of sized types $\vec{\tau}$, annotated with a
size i. Cost expressions c are either sizes i, constants k, the addition of two costs, the maximum of two costs, or a constant multiplied by a cost. A size i an arithmetic expression that may contain constants (k) or size variables (n, m, ...). Definitions of global and local types are based on the most commonly used MPST in the literature [Coppo et al. 2015]. The syntax of global (G) and local (L) types in MPST is given below:

$$G ::= \mathbf{p} \to \mathbf{q}\{\tau \diamond c\}.G \mid \mathbf{p} \to \mathbf{q} : \{l_i.G_i\}_{i\in I} \mid \mu X.G \mid X \mid \mathsf{end}$$
$$L ::= \mathbf{p}\,!\,\tau.L \mid \mathbf{p} \oplus \{l_i.L_i\}_{i\in I} \mid \mathbf{p}\,?\,\tau \diamond c.L \mid \mathbf{p}\,\&\,\{l_i.L_i\}_{i\in I} \mid \mu X.L \mid X \mid \mathsf{end}$$

We start with a set of *roles*, $\mathbf{p}$, $\mathbf{q}$, ..., and a set of *labels*, l_1, l_2, ... These are considered as natural numbers: roles are *participant* identifiers, e.g. thread or process ids; and labels are tags that differentiate branches in the data/control flow. Global type $\mathbf{p} \to \mathbf{q}\{\tau \diamond c\}.G$ denotes *data* interactions from role $\mathbf{p}$ to role $\mathbf{q}$ with value of type τ and local computation cost c; *Branching* is represented by $\mathbf{p} \to \mathbf{q} : \{l_i.G_i\}_{i\in I}$ with actions l_i from $\mathbf{p}$ to $\mathbf{q}$. end represents a *termination* of the protocol. $\mu X.G$ represents a *recursion*, which is *equivalent* to $[\mu X.\ G/X]G$. We assume recursive types are guarded [Pierce 2002].

Each role in G represents a different participant in a parallel process. *Local types* represent the communication actions performed by each role. The *send* type $\mathbf{p}\,!\,\tau.L$ expresses sending of a value of type τ to role $\mathbf{p}$ followed by interactions specified by L. The *receive* type $\mathbf{p}\,?\,\tau \diamond c.L$ receives a value of type τ from role $\mathbf{p}$ with local computation cost c. The *selection* type represents the transmission to role $\mathbf{p}$ of label l_i chosen in the set of labels ($i \in I$) followed by L_i. The *branching* type is its dual. The rest are the same as G. $\mathsf{roles}(G)/\mathsf{roles}(L)$ denote the set of roles that occur in G/L.

Remark 3.1. Global types which combine label and data messages are also used in the literature. They can be encoded as global types in this paper by using singleton labels (see [Deniélou and Yoshida 2013, p.178]). E.g. $\mathbf{p} \to \mathbf{q} : \{l_i(\tau_i).G_i\}_{i\in I}$ is encoded as $\mathbf{p} \to \mathbf{q} : \{l_i.G'_i\}_{i\in I}$ and $G'_i = \mathbf{p} \to \mathbf{q}\{\tau_i\}.G_i$. It is possible to account for the differences in cost by setting the cost of sending/receiving labels appropriately, e.g. removing the cost of sending labels, and slightly increasing the size of the data messages, to account for the fact that they must be sent alongside a label.

End Point Projection. The local type L of a participant $\mathbf{p}$ in a global type G can be obtained by the *end point projection* (EPP) of G onto $\mathbf{p}$, denoted by G as $G \upharpoonright \mathbf{p}$. The local type gives a local view of a global protocol onto each participant. Our definition of EPP follows the standard projection rules in [Demangeon and Honda 2012; Deniélou and Yoshida 2013]. The projection uses the *full merging* operator [Demangeon and Honda 2012; Deniélou and Yoshida 2013], which allows more well-formed global types than the original projection rules [Honda et al. 2008].

Definition 3.1 (Projection and Merging). The *end point projection* (EPP) of G onto $\mathbf{p}$, denoted by G as $G \upharpoonright \mathbf{p}$, is the partial function defined below, together with the merging of local types L_i:

Projection:

$$(\mathbf{q} \to \mathbf{r}\{\tau \diamond c\}.\ G) \upharpoonright \mathbf{p} = \begin{cases} \mathbf{r}\,!\,\tau.\ (G \upharpoonright \mathbf{p}) & \text{if } \mathbf{p} = \mathbf{q} \neq \mathbf{r} \\ \mathbf{q}\,?\,\tau \diamond c.\ (G \upharpoonright \mathbf{p}) & \text{if } \mathbf{p} = \mathbf{r} \neq \mathbf{q} \\ G \upharpoonright \mathbf{p} & \text{otherwise} \end{cases}$$

$$(\mathbf{q} \to \mathbf{r} : \{l_i.G_i\}_{i\in I}) \upharpoonright \mathbf{p} = \begin{cases} \mathbf{r} \oplus \{l_i.G_i \upharpoonright \mathbf{p}\} & \text{if } \mathbf{p} = \mathbf{q} \neq \mathbf{r} \\ \mathbf{q}\,\&\,\{l_i.G_i \upharpoonright \mathbf{p}\} & \text{if } \mathbf{p} = \mathbf{r} \neq \mathbf{q} \\ \sqcap_{i\in I}(G_i \upharpoonright \mathbf{p}) & \text{otherwise} \end{cases}$$

$$(\mu X.G) \upharpoonright \mathbf{p} = \begin{cases} \mu X.(G \upharpoonright \mathbf{p}) & \text{if } G \upharpoonright \mathbf{p} \neq X',\ \forall X' \\ \mathsf{end} & \text{otherwise} \end{cases} \qquad (X) \upharpoonright \mathbf{p} = X \qquad \mathsf{end} \upharpoonright \mathbf{p} = \mathsf{end}$$

Merging:

$$\mathbf{p} \,\&\, \{l_i.L_i\}_{i\in I} \sqcap \mathbf{p} \,\&\, \{l_j.L'_j\}_{j\in J} = \mathbf{p} \,\&\, \{l_k.L_k \sqcap L'_k\}_{k\in I\cap J} \cup \{l_l.L_l\}_{l\in I\setminus J} \cup \{l_m.L_m\}_{m\in J\setminus I}$$

$$\mu X.L_1 \sqcap \mu X.L_2 = \mu X.(L_1 \sqcap L_2) \qquad L \sqcap L = L$$

The first line of the projection rule defines a case where the sender and receiver are the same [Deniélou et al. 2012]. The global type projection onto a role is not necessarily defined. Particularly, projecting $\mathbf{q} \to \mathbf{r} : \{l_i.G_i\}$ onto $\mathbf{p}$, with $\mathbf{r} \neq \mathbf{p}$ and $\mathbf{q} \neq \mathbf{p}$, is only defined if the projection of all G_i onto $\mathbf{p}$ can be *merged* (Def. 3.1). Two local types can be merged only if they are the same, or if they branch on the same role, and their continuations can be merged. For example, $\mathbf{p}$'s local type of the global type $\mathbf{p} \to \mathbf{q}\{\tau\diamond c\}$.end is $\mathbf{q} \,!\, \tau$.end, while $\mathbf{q}$'s is $\mathbf{p} \,?\, \tau\diamond c$.end. As a more complex example, $\mathbf{r}$'s local type of the branching global type:

$$\mu X.\mathbf{p} \to \mathbf{q} \left\{ l_1.\mathbf{q} \to \mathbf{r} : l_2.\mathbf{p} \to \mathbf{r} : l_3.X,\ l_4.\mathbf{q} \to \mathbf{r} : l_5.\mathbf{p} \to \mathbf{r} : l_6.\mathsf{end} \right\}$$

is $\mu X.\mathbf{q}\ \&\ \{l_2.\mathbf{p}\ \&\ \{l_3.X\}, l_5.\mathbf{p}\ \&\ \{l_6.\mathsf{end}\}\}$.

We say that a global type is *well formed*, if its projection on all its roles is defined. We denote: $\mathsf{Wf}(G) = \forall \mathbf{p} \in \mathsf{roles}(G), \exists L, G \upharpoonright \mathbf{p} = L$.

Definition 3.2 (Label Broadcasting). We define a macro to represent the broadcasting of a label to multiple participants in a choice. We write $\mathbf{p} \to \{\mathbf{q}_j\}_{j\in[1,n]} : \{l_i.G_i\}_{i\in I}$ as a synonym to $\mathbf{p} \to \mathbf{q}_1\{l_i.\mathbf{p} \to \mathbf{q}_2\{l_i.\ldots.\mathbf{p} \to \mathbf{q}_n\{l_i.G_i\}\ldots\}\}_{i\in I}$. Similarly, for local types, $\{\mathbf{q}\}_{j\in[1,n]} \oplus \{l_i.L_i\}_{i\in I}$ expands to $\mathbf{q} \oplus \{l_i.\mathbf{q}_2 \oplus \{l_i.\ldots.\mathbf{q}_n \oplus \{l_i.L_i\}\ldots\}\}_{i\in I}$.

It is straightforward to derive: $(\mathbf{p} \to \{\mathbf{q}_j\}_{j\in J} : \{l_i.G_i\}_{i\in I}) \upharpoonright \mathbf{r} = \{\mathbf{q}\}_{j\in J} \oplus \{l_i.G_i \upharpoonright \mathbf{r}\}_{i\in I}$, if $\mathbf{p} = \mathbf{r}$, and $(\mathbf{p} \to \{\mathbf{q}_j\}_{j\in J} : \{l_i.G_i\}_{i\in I}) \upharpoonright \mathbf{r} = \mathbf{q}_j\ \&\ \{l_i.G_i \upharpoonright \mathbf{r}\}_{i\in I}$, if $\mathbf{r} = \mathbf{q}_j$ for some $j \in J$.

3.1 Labelled Transition System of Global Types

We introduce the labelled transition system (LTS) of global types to associate protocol execution costs with cost annotations. Our semantics is based on the LTS for global and local types in Deniélou and Yoshida [2013] that define their asynchronous operational semantics, and prove their sound and complete correspondence.

We designate the *observables* $(\alpha, \alpha', \ldots)$ to be the send, receive, branch and select actions that trigger a transition, and an *internal* transition at a role, which represents the cost c a role spends performing computation at the receiver (denoted by $\mathbf{p}\diamond c$). The syntax of the observables is:

$$\alpha ::= \mathbf{pq} \,!\, \tau \mid \mathbf{pq} \,?\, \tau \mid \mathbf{pq}\oplus \mid \mathbf{pq}\& \mid \mathbf{p}\diamond c$$

The $\mathbf{p}\diamond c$ does not affect the communication structure of the protocol, similar to the silent actions of common process calculi. We say that the *subject* of an action α is the role in charge of performing it: $\mathbf{p} = \mathsf{subj}(\mathbf{pq} \,!\, \tau) = \mathsf{subj}(\mathbf{qp} \,?\, \tau) = \mathsf{subj}(\mathbf{pq}\oplus) = \mathsf{subj}(\mathbf{qp}\&) = \mathsf{subj}(\mathbf{p}\diamond c)$.

Following [Deniélou and Yoshida 2013], we extend the grammar of G to represent the intermediate steps in the execution with the construct $\mathbf{p} \rightsquigarrow \mathbf{q}\{\tau\diamond c\}.G$ to represent the fact that $\mathbf{p}$ has sent the message of type τ but $\mathbf{q}$ has not received it yet, and $\mathbf{p}\diamond(\tau\diamond c).G$ to represent that $\mathbf{p}$ is performing a computation of type τ and cost c. For the branching we use $\mathbf{p} \rightsquigarrow \mathbf{q}\ j\ \{l_i.\ G_i\}_{i\in I}$ to represent the fact that $\mathbf{p}$ has sent label l_j to $\mathbf{q}$. Then the LTS for global types is defined as below. The main rules different from [Deniélou and Yoshida 2013] are [GR1a,GR2a,GR2b] which consider the execution cost. When we send a message or a label, the type becomes the received mode $\mathbf{p} \rightsquigarrow \mathbf{q}$ (e.g. [GR1a]) and then it asynchronously receives the corresponding message (e.g. [GR2a]). We also observe the actions under the prefix if the participants are unrelated (e.g. [GR4a]).

Definition 3.3 (LTS for Global Types). The relation $G \xrightarrow{\alpha} G'$ is defined as follows:

$$[\text{GR1a}]\ \mathbf{p} \to \mathbf{q}\{\tau\diamond c\}.G \xrightarrow{\mathbf{pq}!\tau} \mathbf{p} \rightsquigarrow \mathbf{q}\{\tau\diamond c\}.G \qquad [\text{GR2a}]\ \mathbf{p} \rightsquigarrow \mathbf{q}\{\tau\diamond c\}.G \xrightarrow{\mathbf{pq}?\tau} \mathbf{q}\diamond(\tau\diamond c).G$$

$$[\text{GR1b}]\ \mathbf{p} \to \mathbf{q}\{l_i.G_i\}_{i\in I} \xrightarrow{\mathbf{pq}\oplus l_j} \mathbf{p} \rightsquigarrow \mathbf{q}\ :j\ \{l_i.G_i\}_{i\in I}\quad (j\in I) \qquad [\text{GR2b}]\ \mathbf{q}\diamond(\tau\diamond c).G \xrightarrow{\mathbf{q}\diamond c} G$$

$$[\text{GR2c}]\ \mathbf{p} \rightsquigarrow \mathbf{q}\ :j\ \{l_i.G_i\}_{i\in I} \xrightarrow{\mathbf{pq}\&l_j} G_j \qquad [\text{GR4a}]\ \frac{G \xrightarrow{\alpha} G'\ \ \mathbf{p},\mathbf{q}\notin \text{subj}(\alpha)}{\mathbf{p} \to \mathbf{q}\{\tau\diamond c\}.G \xrightarrow{\alpha} \mathbf{p} \to \mathbf{q}\{\tau\diamond c\}.G'}$$

$$[\text{GR4b}]\ \frac{\forall i\in I\ \ G_i \xrightarrow{\alpha} G_i'\ \ \mathbf{p},\mathbf{q}\notin \text{subj}(\alpha)}{\mathbf{p} \to \mathbf{q}\{l_i.\ G_i\}_{i\in I} \xrightarrow{\alpha} \mathbf{p} \to \mathbf{q}\{l_i.\ G_i'\}_{i\in I}} \qquad [\text{GR5a}]\ \frac{G \xrightarrow{\alpha} G'\ \ \mathbf{q}\notin \text{subj}(\alpha)}{\mathbf{p} \rightsquigarrow \mathbf{q}\{\tau\diamond c\}.G \xrightarrow{\alpha} \mathbf{p} \rightsquigarrow \mathbf{q}\{\tau\diamond c\}.G'}$$

$$[\text{GR5b}]\ \frac{G \xrightarrow{\alpha} G'\ \ \mathbf{p}\notin \text{subj}(\alpha)}{\mathbf{p}\diamond(\tau\diamond c).G \xrightarrow{\alpha} \mathbf{p}\diamond(\tau\diamond c).G'} \qquad [\text{GR3}]\ \frac{G[\mu X.G/X] \xrightarrow{\alpha} G'}{\mu X.G \xrightarrow{\alpha} G'} \qquad [\text{GR5c}]\ \frac{G_j \xrightarrow{\alpha} G_j'\ \ \mathbf{q}\notin \text{subj}(\alpha)\ \ \forall i\in I\setminus j, G_i = G_i'}{\mathbf{p} \rightsquigarrow \mathbf{q}\ j\ \{l_i.\ G_i\}_{i\in I} \xrightarrow{\alpha} \mathbf{p} \rightsquigarrow \mathbf{q}\ j\ \{l_i.\ G_i'\}_{i\in I}}$$

3.2 Labelled Transition System of Local Types

The labelled transition system (LTS) of local types are given for configurations (C) which map each participant to its local type and a set of FIFO queues (Q) where each represents a queue from sender $\mathbf{p}$ to receiver $\mathbf{q}$. We also extend the syntax $(\tau\diamond c).L$ to represent the intermediate state where the receiver executes a local computation with cost c. In the definition below, [LR2,LR3] formalise the observability of the local computation cost c when receiving the value. Other rules are the standard FIFO enqueue and dequeue rules.

Definition 3.4 (LTS for Local Types). The relation $\langle C, Q\rangle \xrightarrow{\ell} \langle C', Q'\rangle$ where $C = [\mathbf{p} \mapsto L_i]_{i\in I}$ and $Q = [\mathbf{pq} \mapsto w]_{i\in I, j\in I}$ is defined as follows:

$$\begin{array}{lrl}
[\text{LR1}] & \langle C[\mathbf{p}\mapsto \mathbf{q}\,!\,\tau.\,L], Q[\mathbf{pq}\mapsto w]\rangle & \xrightarrow{\mathbf{pq}!\tau} \langle C[\mathbf{p}\mapsto L], Q[\mathbf{pq}\mapsto a\cdot w]\rangle \\
[\text{LR2}] & \langle C[\mathbf{p}\mapsto \mathbf{q}\,?\,\tau\diamond c.\,L], Q[\mathbf{qp}\mapsto w\cdot\tau]\rangle & \xrightarrow{\mathbf{qp}?\tau} \langle C[\mathbf{p}\mapsto (\tau\diamond c).L], Q[\mathbf{qp}\mapsto w]\rangle \\
[\text{LR3}] & \langle C[\mathbf{p}\mapsto (\tau\diamond c).L], Q\rangle & \xrightarrow{\mathbf{p}\diamond c} \langle C[\mathbf{p}\mapsto L], Q\rangle \\
[\text{LR4}] & \langle C[\mathbf{p}\mapsto \mathbf{q}\oplus\{l_i.L_i\}_{i\in I}], Q[\mathbf{pq}\mapsto w]\rangle & \xrightarrow{\mathbf{pq}\oplus l_i} \langle C[\mathbf{p}\mapsto L_i], Q[\mathbf{pq}\mapsto l_i\cdot w]\rangle \\
[\text{LR5}] & \langle C[\mathbf{p}\mapsto \mathbf{q}\,\&\,\{l_i.L_i\}_{i\in I}], Q[\mathbf{qp}\mapsto w\cdot l_i]\rangle & \xrightarrow{\mathbf{qp}\&l_i} \langle C[\mathbf{p}\mapsto L_i], Q[\mathbf{qp}\mapsto w]\rangle
\end{array}$$

The following theorem shows the global type semantics is exactly matched with local asynchronous interactions between participants.

Theorem 3.5. *[Soundness and Completeness] Let G be a global type with $\mathcal{P} = \text{roles}(G)$ and let $C = [\mathbf{p}\mapsto G\upharpoonright \mathbf{p}]_{\mathbf{p}\in\mathcal{P}}$. Then $G \approx (C; [\mathbf{pq}\mapsto\epsilon]_{\mathbf{p},\mathbf{q}\in\mathcal{P}})$.*

Proof. (Sketch) The proof of soundness and completeness is a straightforward adaptation of that in [Deniélou and Yoshida 2013]. The distinction between send/receive and select/branch actions is straightforward, e.g. actions GR1a and GR1b are special cases of rule GR1 in [Deniélou and Yoshida 2013] (see Remark 3.1). The addition of cost actions to local types is does not complicate the proof, since it only happens after communication has taken place, which is ensured by being local to each role, and the local context. □

Definition 3.6 (Deadlock-freedom). We call $\langle C_0, Q_0\rangle$ *deadlock-free* if for all C and Q such that $\langle C_0, Q_0\rangle \xrightarrow{\vec{\ell}} \langle C, Q\rangle$, either (1) $\forall \mathbf{p}\in \text{dom}(C).\ C(\mathbf{p}) = \text{end}$ and $Q = \emptyset$; or (2) $\langle C, Q\rangle \xrightarrow{\ell} \langle C', Q'\rangle$ for some C' and Q'. We call global type G *deadlock-free* if $\langle C_0, \emptyset\rangle$ such that $C_0 = [\mathbf{p}\mapsto G\upharpoonright \mathbf{p}]_{\mathbf{p}\in\mathcal{P}}$ with $\mathcal{P} = \text{roles}(G)$ is deadlock-free.

Note that the definition of deadlock-freedom is not affected by cost annotations. Hence by Remark 3.1, we can directly apply the result in [Deniélou and Yoshida 2013] to obtain:

Theorem 3.7 (Deadlock-freedom). ([Deniélou and Yoshida 2013]) $\text{Wf}(G)$ *is deadlock-free.*

4 COST FOR MULTIPARTY SESSION PROTOCOLS (1): BOUNDED RECURSION

This section presents the cost analysis for protocols with bounded recursions. We first define the cost of a trace, as the total cost accumulated by each participant at the end of the execution of a trace. Next we introduce the cost model using global types and show that it provides an upper bound of the cost of any possible trace for each participant.

4.1 Cost of Local Traces

We first explain several assumptions for giving a calculation of cost. Our first assumption in theory is that participants do not share resources with other participants, i.e. they can run on independent CPUs, with no source of contention such as memory or shared cache.

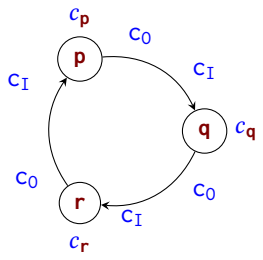


Fig. 3. A cost-annotated ring protocol.

The cost of a trace is the total execution time taken by each participant to run the protocol from start to end. We compute this total execution cost by tracking the dependency from the input to the output (*IO-dependency*), and by associating each action in a trace (Def. 3.3) with an execution cost. Assuming that every participant has access to their own set of resources (including CPU) implies that every action in the trace (Def. 3.3) will be triggered as early as possible, e.g. send actions will not be arbitrarily delayed by other actions. This also implies that any pair of actions will happen in parallel if swapped freely according to the semantics.

We explain these assumptions with simple examples. Consider the following trace:

$$\mathbf{p} \to \mathbf{r}\{\tau \diamond c\}.\mathbf{q} \to \mathbf{r}\{\tau' \diamond c'\} \xrightarrow{\mathbf{pr}!\tau \cdot \mathbf{qr}!\tau'} \mathbf{p} \rightsquigarrow \mathbf{r}\{\tau \diamond c\}.\mathbf{q} \rightsquigarrow \mathbf{r}\{\tau' \diamond c'\}$$

According to Def. 3.3, since $\mathrm{subj}(\mathbf{pr} \,!\, \tau) = \mathbf{p} \neq \mathbf{q} = \mathrm{subj}(\mathbf{qr} \,!\, \tau')$, because the sender is different in each action, this would be another possible trace for the same global type:

$$\mathbf{p} \to \mathbf{r}\{\tau \diamond c\}.\mathbf{q} \to \mathbf{r}\{\tau' \diamond c'\} \xrightarrow{\mathbf{qr}!\tau' \cdot \mathbf{pr}!\tau} \mathbf{p} \rightsquigarrow \mathbf{r}\{\tau \diamond c\}.\mathbf{q} \rightsquigarrow \mathbf{r}\{\tau' \diamond c'\}$$

The intuition is that, since $\mathbf{p}$ and $\mathbf{q}$ are running on different CPUs, and their actions are independent, both $\mathbf{qr} \,!\, \tau'$ and $\mathbf{pr} \,!\, \tau$ can happen in parallel.

We assume that the cost of the message-passing operations depend on the size of the data that is sent and that the costs of sending and receiving messages are known, and that they are functions on the size of the data. Due to the presence of IO-dependencies, we record when the *send* actions have happened: a participant cannot perform a computation until received a value, and it cannot receive a value until at least the time it took for the sender to finish sending the data has passed.

Cost Environments, Queues and Trace Cost. To record the cost, we use *queues* that record when the data becomes available. A cost dependency queue is, similarly to Q in Def. 3.4, a mapping from pairs of participants to queues of execution times, that records when the data in the queue becomes available. We use W for these cost queues.

The cost of a sequence of actions is defined as a mapping from participants to the total execution time accumulated by each participant, defined as *cost environments*. It is computed by adding the cost of each individual action to the cost of the participant that performs it, taking into account the cost dependencies recorded by the queue.

We call mappings from participants to total accumulated costs *cost environments*. If c is an execution time estimation and R is a set of participants, then $T = [\mathbf{r} \mapsto c]_{r \in R}$, with the usual

extension $T[\mathbf{r} \mapsto c]$ and indexing $T(\mathbf{r})$ operations. We define $T(\mathbf{r}) = 0$, if $\mathbf{r} \notin T$ and the following operations:

$$T[c' | \mathbf{r} \hookrightarrow c] = T[\mathbf{r} \mapsto \max(T(\mathbf{r}), c') + c]$$

This operation is used to record cost dependencies. Specifically, $T[c' | \mathbf{r} \hookrightarrow c]$ means that $\mathbf{r}$ incurs additional cost c, after possibly waiting for an action by an external process with total cost c'. If the action that took c time depends on an action that took c' time, then the cost of $\mathbf{r}$ will be updated with the maximum of either the time c', or the total accumulated cost by $\mathbf{r}$. We write:

$$T[\mathbf{r}' | \mathbf{r} \hookrightarrow c] \text{ for } T[T(\mathbf{r}') | \mathbf{r} \hookrightarrow c] \quad \text{and} \quad T[\mathbf{r} \hookrightarrow c] \text{ for } T[0 | \mathbf{r} \hookrightarrow c].$$

For the cost of actions, we define: (1) $\mathsf{c}_\mathsf{I}(\tau)$ is the time required for *receiving* a value of type τ; (2) $\mathsf{c}_0(\tau)$ is the time required for *sending* a value of type τ; (3) c^τ is cost associated to type τ; and (4) the cost of labels l is calculated as unit type 1.

Definition 4.1 (Cost of a Trace and Action Cost). The cost of a trace $\vec{\alpha}$ takes as an input an initial cost T, an input dependency queue W, and produces a pair of a final cost T' and queue W'.

$$C(\epsilon)(T, W) = (T, W) \qquad C(\alpha \cdot \vec{\alpha}) = C(\vec{\alpha})(C(\alpha)(T, W))$$

The initial cost is $C(\vec{\alpha}) = C(\vec{\alpha})([\mathbf{r} \mapsto 0]_{\mathbf{r} \in \vec{\alpha}}, [\mathbf{pq} \mapsto \epsilon]_{\mathbf{pq} \in \vec{\alpha}})$ with empty initial queues and zero costs. The cost of individual actions is defined below:

$$\begin{aligned}
C(\mathbf{pq}\,?\,\tau)(T, W[\mathbf{pq} \mapsto c \cdot w]) &= (T[c | \mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(\tau)], W[\mathbf{pq} \mapsto w]) \\
C(\mathbf{pq}\,\&\,l_k)(T, W[\mathbf{pq} \mapsto c \cdot w]) &= (T[c | \mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(1)], W[\mathbf{pq} \mapsto w]) \\
C(\mathbf{pq}\,!\,\tau)(T, W[\mathbf{pq} \mapsto w])) &= (T[\mathbf{p} \hookrightarrow \mathsf{c}_0(\tau)], W[\mathbf{pq} \mapsto w \cdot (T(\mathbf{p}) + \mathsf{c}_0(\tau))]) \\
C(\mathbf{pq} \oplus l_k)(T, W[\mathbf{pq} \mapsto w])) &= (T[\mathbf{p} \hookrightarrow \mathsf{c}_0(1)], W[\mathbf{pq} \mapsto w \cdot (T(\mathbf{p}) + \mathsf{c}_0(1))]) \\
C(\mathbf{p} \diamond c)(T, W) &= (T[\mathbf{p} \hookrightarrow c], W)
\end{aligned}$$

Example of Trace Cost. We show an example of calculating the cost of a trace. Consider the following global type:

$$G = \mathbf{p} \to \mathbf{q}\{\mathrm{str}^n \diamond n \times 3\mathrm{ms}\}.\mathbf{q} \to \mathbf{p}\{\mathrm{int}^i \diamond 6\mathrm{ms}\}.\mathrm{end}$$

In this protocol, there are two participants, $\mathbf{p}$ and $\mathbf{q}$. First, $\mathbf{p}$ sends a string of size n to $\mathbf{q}$, that requires $n \times 3$ms of local computation time. Then, $\mathbf{q}$ replies with an integer of size i (i.e. smaller than i) to $\mathbf{p}$, that takes a constant computation time of 6ms. We represent this scenario as a trace of actions:

$$tr = \mathbf{pq}\,!\,\mathrm{str}^n \cdot \mathbf{pq}\,?\,\mathrm{str}^n \cdot \mathbf{q} \diamond (n \times 3\mathrm{ms}) \cdot \mathbf{qp}\,!\,\mathrm{int}^i \cdot \mathbf{qp}\,?\,\mathrm{int}^i \cdot \mathbf{p} \diamond (6\mathrm{ms})$$

To compute the cost, we traverse the trace, record at which time each event happened in the message queue, and add the cost of each action to the total execution time accumulated by the subject of the action. For example, $C(\mathbf{pq}\,!\,\mathrm{str}^n)([], []) = ([\mathbf{p} \mapsto \mathsf{c}_0(\mathrm{str}^n)], [\mathbf{pq} \mapsto \mathsf{c}_0(\mathrm{str}^n)])$, i.e. the cost of sending a string of size n is added to the cost for $\mathbf{p}$, and the message queue now records that this message was sent after $\mathsf{c}_0(\mathrm{str}^n)$ time. Then, $C(\mathbf{pq}\,?\,\mathrm{str}^n)([\mathbf{p} \mapsto \mathsf{c}_0(\mathrm{str}^n)], [\mathbf{pq} \mapsto \mathsf{c}_0(\mathrm{str}^n)]) = ([\mathbf{p} \mapsto \mathsf{c}_0(\mathrm{str}^n);\ \mathbf{q} \mapsto \mathsf{c}_0(\mathrm{str}^n) + \mathsf{c}_\mathsf{I}(\mathrm{str}^n)], [])$. That means that the cost of receiving a string of size n is added to the cost of $\mathbf{q}$, after the time recorded in the queue $\mathbf{pq}$, in this case the cost of sending a string of size n, and the message queue would now be empty. By following the cost rules with the remaining actions, we produce the following cost equation:

$$C(\mathrm{tr}) = \begin{bmatrix} \mathbf{p} \mapsto \mathsf{c}_0(\mathrm{str}^n) + \mathsf{c}_\mathsf{I}(\mathrm{str}^n) + n \times 3\mathrm{ms} + \mathsf{c}_0(\mathrm{int}^i) + \mathsf{c}_\mathsf{I}(\mathrm{int}^i) + 6\mathrm{ms} \\ \mathbf{q} \mapsto \mathsf{c}_0(\mathrm{str}^n) + \mathsf{c}_\mathsf{I}(\mathrm{str}^n) + n \times 3\mathrm{ms} + \mathsf{c}_0(\mathrm{int}^i) \end{bmatrix}$$

By instantiating the sizes of the messages and the send/receive costs with e.g. profiling information, we can now estimate how much time it will take the protocol to complete.

Example 4.2 (Scatter/Gather). This global type represents a scatter/gather protocol, where $\mathbf{p}$ distributes tasks to $\mathbf{q}$ and $\mathbf{r}$, and $\mathbf{s}$ collects the results. We omit the cost on the receiving end of $\mathbf{s}$ to represent that $\mathbf{s}$ simply gathers the results, and that has computation cost 0.

$$G = \mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{p} \to \mathbf{r}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{s}\{\tau_2\}.\mathbf{r} \to \mathbf{s}\{\tau_2\}.\text{end}$$

We show below two examples of the possible traces:

$$tr_1 = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qs}\,!\,\tau_2 \cdot \mathbf{qs}\,?\,\tau_2 \cdot \mathbf{s} \diamond 0 \cdot \mathbf{pr}\,!\,\tau_1 \cdot \mathbf{pr}\,?\,\tau_1 \cdot \mathbf{r} \diamond c_1 \cdot \mathbf{rs}\,!\,\tau_2 \cdot \mathbf{rs}\,?\,\tau_2 \cdot \mathbf{s} \diamond 0$$
$$tr_2 = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pr}\,!\,\tau_1 \cdot \mathbf{pr}\,?\,\tau_1 \cdot \mathbf{r} \diamond c_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qs}\,!\,\tau_2 \cdot \mathbf{rs}\,!\,\tau_2 \cdot \mathbf{qs}\,?\,\tau_2 \cdot \mathbf{s} \diamond 0 \cdot \mathbf{rs}\,?\,\tau_2 \cdot \mathbf{s} \diamond 0$$

Since we assume that each participant can run in parallel to the remaining of the participants, the cost of both traces yield the same result:

$$\begin{aligned} C(tr_1)([],[]) &= C(\mathbf{pq}\,?\,\tau_1 \cdots)([\mathbf{p} \mapsto \mathtt{c}_0(\tau_1)], [\mathbf{pq} \mapsto \mathtt{c}_0(\tau_1)]) \\ &= C(\mathbf{q} \diamond c_1 \cdots)([\mathbf{p} \mapsto \mathtt{c}_0(\tau_1)]; \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1), []) \\ &= \left[\begin{array}{l} \mathbf{p} \mapsto 2 \times \mathtt{c}_0(\tau_1);\ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2);\quad \mathbf{r} \mapsto 2 \times \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2); \\ \mathbf{s} \mapsto \max(\mathtt{c}_\mathtt{I}(\tau_2), \mathtt{c}_0(\tau_1)) + \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2) + \mathtt{c}_\mathtt{I}(\tau_2) \end{array}\right] \end{aligned}$$

Example 4.3 (Parallel Pipeline). We now show the cost of a fragment of the trace that corresponds with $G = \mu X.\mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{r}\{\tau_2 \diamond c_2\}.X$. Two possible traces for two iterations of this protocol are as follows:

$$tr_1 = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qr}\,!\,\tau_2 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qr}\,!\,\tau_2 \cdot \mathbf{qr}\,?\,\tau_2 \cdot \mathbf{r} \diamond c_2 \cdot \mathbf{qr}\,?\,\tau_2 \cdot \mathbf{r} \diamond c_2$$
$$tr_2 = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qr}\,!\,\tau_2 \cdot \mathbf{qr}\,?\,\tau_2 \cdot \mathbf{r} \diamond c_2 \cdot \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qr}\,!\,\tau_2 \cdot \mathbf{qr}\,!\,\tau_2 \cdot \mathbf{r} \diamond c_2$$

In the first trace, $\mathbf{p}$ sends first two messages to $\mathbf{q}$. Then, $\mathbf{q}$ receives, computes them and sends the results to $\mathbf{r}$. Finally, $\mathbf{r}$ receives the results, and performs their computation with cost c_2. The second trace, instead, is the repetition of two single iterations of the protocol, where $\mathbf{p}$ sends one message, $\mathbf{q}$ receives, processes and sends the result to $\mathbf{r}$, and $\mathbf{r}$ performs its local computation. Note, however, that since the cost models assume that each participant runs at a different CPU, the costs of both traces is the same. To help readability, we name $T_\mathbf{p} = \mathtt{c}_0(\tau_1)$, $T_\mathbf{q} = \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2)$ and $T_\mathbf{q} = \mathtt{c}_\mathtt{I}(\tau_2) + c_2$. The trace cost is:

$$\begin{aligned} C(tr_1)([],[]) &= C(\mathbf{pq}\,?\,\tau_1 \cdots)([\mathbf{p} \mapsto \mathtt{c}_0(\tau_1)], [\mathbf{pq} \mapsto \mathtt{c}_0(\tau_1)]) = C(\mathbf{q} \diamond c_1 \cdots)([\mathbf{p} \mapsto \mathtt{c}_0(\tau_1)]; \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1), []) \\ &= \left[\mathbf{p} \mapsto 2 \times T_\mathbf{p};\quad \mathbf{q} \mapsto T_\mathbf{p} + T_\mathbf{q} + \max(T_\mathbf{p}, T_\mathbf{q})\quad \mathbf{r} \mapsto T_\mathbf{p} + T_\mathbf{q} + T_\mathbf{r} + \max(T_\mathbf{p}, T_\mathbf{q}, T_\mathbf{r})\right] \end{aligned}$$

We can see that the cost is the expected one, where the cost includes the initialisation and finalisation of the protocol, where the costs are added, and a pipeline steady state, where the cost is the maximum of the costs of each participant.

Example 4.4 (Dependency Cycle). We change slightly the pipeline example, to illustrate what happens to the trace cost when we introduce a dependency cycle in the protocol. The protocol that we show below is a recursive ping-pong, where $\mathbf{p}$ sends to $\mathbf{q}$, and then $\mathbf{q}$ replies to $\mathbf{p}$: $\mu X.\mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{p}\{\tau_2 \diamond c_2\}.X$. There is only one possible trace for such protocol, due to the input/output dependencies between $\mathbf{q}$ and $\mathbf{p}$ (see conditions $\mathbf{p}, \mathbf{q} \notin \text{subj}(\alpha)$ in Def. 3.3, e.g. [GR4a]). The trace and cost in this instance is:

$$tr = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qp}\,!\,\tau_2 \cdot \mathbf{qp}\,?\,\tau_2 \cdot \mathbf{p} \diamond c_2 \cdot \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q} \diamond c_1 \cdot \mathbf{qp}\,!\,\tau_2 \cdot \mathbf{qp}\,?\,\tau_2 \cdot \mathbf{p} \diamond c_2$$
$$C(tr)([],[]) = \left[\mathbf{p} \mapsto 2 \times (T_\mathbf{p} + T_\mathbf{q})\quad \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + T_\mathbf{p} + 2 \times T_\mathbf{q}\right]$$

Here, $T_\mathbf{p} = \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_2) + c_2$ and $T_\mathbf{q} = \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2)$. Participant $\mathbf{p}$ needs to send τ_1, then wait for $\mathbf{q}$ to complete its part of the protocol, and then receive τ_2 and process it. Therefore, the cost per iteration is $T_\mathbf{p} + T_\mathbf{q}$ in all cases. For participant $\mathbf{q}$, the situation is slightly different. A single iteration of $\mathbf{q}$ only requires it to wait until $\mathbf{p}$ sends τ_1, and then perform its part of the protocol. Hence, the cost is $\mathtt{c}_0(\tau_1) + T_\mathbf{q}$. However, on the next iteration, $\mathbf{q}$ needs to wait until $\mathbf{p}$ finishes with its actions for the previous iteration. This implies that the cost of a single iteration for $\mathbf{q}$ ($\mathtt{c}_0(\tau_1) + T_\mathbf{q}$) is *less* than the average cost per iteration ($T_\mathbf{p} + T_\mathbf{q}$).

4.2 Cost of Global Protocols

We have introduced a way to compute the cost of *one* trace. This cost is useful to statically analyse the potential execution times of particular executions of a protocol. However, it is in general not feasible to produce all possible traces to analyse the cost of a concurrent/distributed system. Our *global type cost* addresses this issue, by providing a syntactic method to estimate an upper bound of the execution cost.

The global type cost produces, just like Def. 4.1, a *cost environment*, with a per-participant estimation. The protocol will complete when all the participants have finished their tasks, and so the overall cost is the maximum of the cost per participant. The global type cost is a function from a global type, an estimation of the number of iterations for the recursive protocols, and an initial cost environment. For proving completeness, we use a dependency queue as an input to the global type cost, that will only be used at intermediate stages of the execution.

Definition 4.5 (Global Type Cost). Let the maximum operation that combines two cost environments compute a per participant maximum $\max(T, T') = [\mathbf{p} \mapsto \max(T(\mathbf{p}), T'(\mathbf{p}))]_{\mathbf{p} \in \mathrm{dom}(T) \cup \mathrm{dom}(T')}$[1]. We define the function unfold, that unrolls the recursive protocol $\mu X.G$ k times:

$$\mathsf{unfold}^{k+1}(X, G) = [\mathsf{unfold}^{k}(X, G)/X]G \qquad \mathsf{unfold}^{0}(X, G) = \mathsf{end}$$

Then the *global type cost* is defined recursively on the structure of global types:

- $C(\mathbf{p} \to \mathbf{q}\{\tau \diamond c\}.G, \vec{k})(T, W) = C(G, \vec{k})(T[\mathbf{p} \hookrightarrow \mathsf{c}_0(\tau)][\mathbf{p}|\mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(\tau) + c], W)$
- $C(\mathbf{p} \to \mathbf{q}\{l_i.G_i\}_{i \in I}, \vec{k})(T, W) = \max\{C(G_i, \vec{k})(T[\mathbf{p} \hookrightarrow \mathsf{c}_0(1)][\mathbf{p}|\mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(1)], W)\}_{i \in I}$
- $C(\mu X.G, k \cdot \vec{k})(T, W) = C(\mathsf{unfold}^k(X, G), \vec{k})(T, W)$
- $C(\mathsf{end})(T, W) = (T, W)$

For completeness, and for the proofs, we define the cost rules for the extended global types used in the semantics.

- $C(\mathbf{p} \rightsquigarrow \mathbf{q}\{\tau \diamond c\}.G, \vec{k})(T, W[\mathbf{pq} \mapsto c_\mathbf{p} \cdot w]) = C(G, \vec{k})(T[c_\mathbf{p}|\mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(\tau) + c], W[\mathbf{pq} \mapsto w])$
- $C(\mathbf{p} \rightsquigarrow \mathbf{q}\ j\ \{l_i.G_i\}_{i \in I}, \vec{k})(T, W[\mathbf{pq} \mapsto c_\mathbf{p} \cdot w]) = C(G_j, \vec{k})(T[c_\mathbf{p}|\mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(1)], W[\mathbf{pq} \mapsto w])$
- $C(\mathbf{p} \diamond (\tau \diamond c).G, \vec{k})(T, W) = C(G, \vec{k})(T[\mathbf{p} \hookrightarrow c], W)$

We write $C(G, \vec{k})$ to represent $C(G, \vec{k})([], [])$. Since the dependency queue is only used in the definitions for the intermediate stages of the execution ($\rightsquigarrow$), we can write $C(G, \vec{k})(T)$. When we compare the output of the cost functions, we refer to the per-participant cost, ignoring the dependency queue: $(T, W) \leq (T', W'), \forall \mathbf{p} \in T, T(\mathbf{p}) \leq T(\mathbf{p}')$.

The first rule in Def. 4.5 explains the cost of an interaction from $\mathbf{p}$ to $\mathbf{q}$. Participant $\mathbf{p}$ needs to send a message, and this is what the cost $T[\mathbf{p} \hookrightarrow \mathsf{c}_0(\tau)]$ reflects. Participant $\mathbf{q}$ will receive a value from $\mathbf{p}$, and then take c time performing a computation. Since $\mathbf{q}$ needs to wait until $\mathbf{p}$ finishes, we add this dependency to the cost: $[\mathbf{p}|\mathbf{q} \hookrightarrow \mathsf{c}_\mathsf{I}(\tau) + c]$. The cost of a choice is computed similarly, but to produce an upper bound of the cost of all branches, we compute the maximum cost per-participant. The cost of the intermediate stages of the execution of the protocol ($\mathbf{p} \rightsquigarrow \mathbf{q}$) requires accessing the information in W, and retrieving when $\mathbf{p}$ completed the send operation. The cost of a computation ($\mathbf{p} \diamond (\tau \diamond c)$) is added to accumulated cost of participant $\mathbf{p}$. The cost of a recursive protocol uses parameter k to first unroll the recursion, and then compute the cost. We go back to the Examples 4.2, 4.3 and 4.4 and show the computed cost by their global type.

Example 4.6 (Scatter/Gather). We illustrate the global type cost using the scatter/gather protocol: $G = \mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{p} \to \mathbf{r}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{s}\{\tau_2\}.\mathbf{r} \to \mathbf{s}\{\tau_2\}.\mathsf{end}$.

[1] The maximum operation is defined even if $\mathbf{p}$ is not in one of the environments: recall that $T(\mathbf{p}) = 0$ if $\mathbf{p} \notin \mathrm{dom}(T)$

$$
\begin{aligned}
&C(\mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\} \ldots)([]) = C(\mathbf{p} \to \mathbf{r}\{\tau_1 \diamond c_1\} \ldots)\left(\left[\mathbf{p} \mapsto \mathtt{c}_0(\tau_1);\ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1\right]\right) \\
&= C(\mathbf{q} \to \mathbf{s}\{\tau_2\} \ldots)\left(\left[\mathbf{p} \mapsto 2 \times \mathtt{c}_0(\tau_1);\ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1; \mathbf{r} \mapsto 2 \times \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1\right]\right) \\
&= C(\mathbf{r} \to \mathbf{s}\{\tau_2\} \ldots)\left(\begin{bmatrix}\mathbf{p} \mapsto 2 \times \mathtt{c}_0(\tau_1);\ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2); \mathbf{r} \mapsto 2 \times \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1; \\ \mathbf{s} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2) + \mathtt{c}_\mathtt{I}(\tau_2);\end{bmatrix}\right) \\
&= \begin{bmatrix}\mathbf{p} \mapsto 2 \times \mathtt{c}_0(\tau_1);\ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2); \mathbf{r} \mapsto 2 \times \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2); \\ \mathbf{s} \mapsto \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2) + \max(\mathtt{c}_\mathtt{I}(\tau_2), \mathtt{c}_0(\tau_1)) + \mathtt{c}_\mathtt{I}(\tau_2);\end{bmatrix}
\end{aligned}
$$

The final cost produced by the global type predicts the same as the one taking any possible trace.

Example 4.7 (Parallel Pipeline). $G = \mu X.\mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{r}\{\tau_2 \diamond c_2\}.X$. Applying the cost models with $\vec{k} = 2$, $C(G, 2)([])$, produces the same cost as the trace cost. Particularly, for any arbitrary k, $C(G, k)([])$ produces:

$$\left[\mathbf{p} \mapsto k \times T_\mathbf{p};\quad \mathbf{q} \mapsto T_\mathbf{p} + T_\mathbf{q} + (k-1) \times \max(T_\mathbf{p}, T_\mathbf{q}) \mathbf{r} \mapsto T_\mathbf{p} + T_\mathbf{q} + T_\mathbf{r} + (k-1) \times \max(T_\mathbf{p}, T_\mathbf{q}, T_\mathbf{r})\right]$$

Example 4.8 (Dependency Cycle). $G = \mu X.\mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\mathbf{q} \to \mathbf{p}\{\tau_2 \diamond c_2\}.X$. For any arbitrary $k > 1$, $C(G, k)([])$ produces the following cost, which corresponds to the trace cost:

$$\left[\mathbf{p} \mapsto k \times (T_\mathbf{p} + T_\mathbf{q})\ \ \mathbf{q} \mapsto \mathtt{c}_0(\tau_1) + T_\mathbf{q} + (k-1) \times (T_\mathbf{p} + \times T_\mathbf{q})\right]$$

We showed in the previous examples that function $C(G)$ accurately predicts an upper bound of the cost obtained from any trace of the protocol. We formalise this statement below in Theorem 4.12, and provide a full proof in Castro-Perez and Yoshida [2020a]. In the formalisation, we use $\mathsf{unfold}(G, \vec{k})$ to unroll all recursion variables, using the parameters $\vec{k}$, i.e. $\mathsf{unfold}(G, \vec{k})$ is defined recursively on G, with the only interesting case $\mathsf{unfold}(\mu X.G, k \cdot \vec{k}) = \mathsf{unfold}^k(X, (\mathsf{unfold}(G, \vec{k})))$. Function $\mathsf{unfold}(G, \vec{k})$ is only defined if there are enough the size of $\vec{k}$ is that of the amount of recursion variables in G.

Definition 4.9 (Well-Formedness of Dependency Queues). A dependency queue W is well formed with respect to a global type G if it *only* contains the values required to compute the cost of G.

$$
\begin{aligned}
\mathsf{Wf}(G, W[\mathbf{pq} \mapsto w]) &\implies \mathsf{Wf}(\mathbf{p} \rightsquigarrow \mathbf{q}\{\tau \diamond c\}.G, W[\mathbf{pq} \mapsto c_\mathbf{p} \cdot w]) \\
\mathsf{Wf}(G, W[\mathbf{pq} \mapsto \epsilon]) &\implies \mathsf{Wf}(\mathbf{p} \to \mathbf{q}\{\tau \diamond c\}.G, W[\mathbf{pq} \mapsto \epsilon]) \\
\mathsf{Wf}(G, W) &\implies \mathsf{Wf}(\mathbf{p} \diamond \{\tau \diamond c\}.G, W) \\
j \in I \wedge \mathsf{Wf}(G_j, W[\mathbf{pq} \mapsto w]) &\implies \mathsf{Wf}(\mathbf{p} \rightsquigarrow \mathbf{q}\ j\ \{\iota_i.G_i\}_{i \in I}, W[\mathbf{pq} \mapsto c \cdot w]) \\
\forall (i \in I), \mathsf{Wf}(G_i, W[\mathbf{pq} \mapsto \epsilon]) &\implies \mathsf{Wf}(\mathbf{p} \to \mathbf{q}\ \{\iota_i.G_i\}_{i \in I}, W[\mathbf{pq} \mapsto \epsilon]) \\
&\quad\ \mathsf{Wf}(\mathsf{end}, [])
\end{aligned}
$$

We generally write $\mathsf{Wf}(G, (T, W)) = \mathsf{Wf}(G, W)$.

Lemma 4.10 (Preservation of Wf). *If* $\mathsf{Wf}(G, (T, W))$ *and* $G \xrightarrow{\ell} G'$, *then* $\mathsf{Wf}(G', C(\ell)(T, W))$.

Proof. By induction on the structure of $G \xrightarrow{\ell} G'$. See Castro-Perez and Yoshida [2020a]. □

This lemma states that if W is well-formed with respect to G, and G' results from taking a step in G, then the queue that results from $C(\ell)(T, W)$ is well formed with respect to G'.

Lemma 4.11 (Cost Preservation). *If* $G \xrightarrow{\ell} G'$, *then* $C(G')(C(\ell)(T, W)) \leq C(G)(T, W)$.

Proof. By induction on the structure of the derivation of $G \xrightarrow{\ell} G'$. □

This is the main lemma, that states that if G transitions to G' with action ℓ, then, given an initial cost/queue (T, W), the cost of G' on an initial cost after running ℓ on (T, W) is less or equal than the cost of G with initial cost (T, W). The reason why this cost is less or equal, rather than equal, is that a branching may take a lower cost path in the protocol.

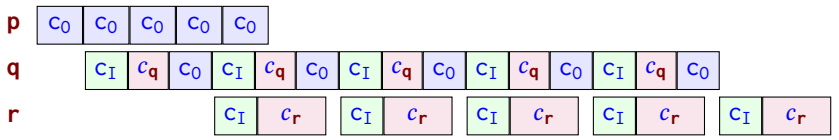


Fig. 4. Latency of a parallel pipeline.

Theorem 4.12 (Bounded-Cost Soundness). *If* $\mathsf{unfold}(G, \vec{k}) \xrightarrow{\vec{\alpha}} \mathsf{end}$, *then* $C(\vec{\alpha}) \leq C(G, \vec{k})$.

Proof. We prove the following generalised statement. If $G \xrightarrow{\vec{\ell}} \mathsf{end}$ and $\mathrm{Wf}(G, (T, W))$ then $C(\vec{\ell}) \leq C(G)$. To recover the original statement, we need to specialise this statement with $G = \mathsf{unfold}(G', \vec{k})$, and $W = []$. By induction on the length of $\vec{\ell}$:

Case $\vec{\ell} = \epsilon$: $G \xrightarrow{\epsilon} G'$ implies that $G = G'$, therefore $G = \mathsf{end}$. $C(\epsilon)(T, W) = C(\mathsf{end})(T, W) = (T, W)$.

Case $\vec{\ell} = \ell_1 \cdot \vec{\ell'}$: $G \xrightarrow{\ell_1 \cdot \vec{\ell'}} G'$ implies that there is a G'' s.t. $G \xrightarrow{\ell_1} G'' \xrightarrow{\vec{\ell'}} G'$. By Lemma 4.10, we know that $\mathrm{Wf}(G'', (C(\ell_1)(T, W)))$. Therefore, by the IH, $C(\vec{\ell'})(C(\ell_1)(T, W)) \leq C(G'', C(\ell_1)(T, W))$. The proof is completed by Lemma 4.11, that allows us to derive that $C(G'', C(\ell_1)(T, W)) \leq C(G)(T, W)$. □

5 COST FOR MULTIPARTY SESSION PROTOCOLS (2): LATENCY OF RECURSION

Previous section presented cost models for multiparty session protocols with bounded recursion. In this section, we extend the cost models for multiparty session protocols with two notions:

(1) The *average cost per iteration* of a protocol, which we call ***latency*** ($C^{\omega}(G)$).
(2) The ***latency relative to*** $\mathtt{p}$, denoted by $C^{\omega}_{\mathtt{p}}(G)$, as the latency of a global type, divided by the number of messages exchanged by participant $\mathtt{p}$ per iteration.

These cost models are useful for scenarios where we do not know how many iterations the protocol is going to run, or this number of iterations is large. For example, consider the protocol for a parallel program following a master-worker pattern, where the master ($\mathtt{m}_1$) distributes a stream of tasks to a series of workers ($\mathtt{w}_i$), and then collects the results ($\mathtt{m}_2$):

$$mw(n) = \mu X.\mathtt{m}_1 \to \{\mathtt{m}_2, \mathtt{w}_1, \ldots, \mathtt{w}_n\} \begin{cases} \iota_1.\ \mathtt{m}_1 \to \mathtt{w}_1\{\tau \diamond c\}. \ldots \mathtt{m}_1 \to \mathtt{w}_n\{\tau \diamond c\}.\mathtt{w}_n \to \mathtt{m}_2\{\tau \diamond c\}. \ldots \mathtt{w}_1 \to \mathtt{m}_2\{\tau \diamond c\}.X \\ \iota_2.\ \mathsf{end} \end{cases}$$

When such protocols are run in practice, they are aimed at speeding up some computation on a large, potentially unbounded, stream of tasks. Therefore, computing $C(mw(n), k)$ can be computationally very expensive, or impossible if k is unknown. These scenarios are where the average cost per iteration, or latency ($C^{\omega}(G)$) is more useful. The key property of $C^{\omega}(G)$ is that approximates to $C(G, k)/k$ as k grows. In the protocol above, it is clear that $C^{\omega}(mw(n)) > C^{\omega}(mw(m))$ if $n > m$, due to the greater number of interactions that involve $\mathtt{m}_1$ and $\mathtt{m}_2$ per iteration. However, unless the cost of the extra interactions outweigh the cost of a computation performed by $\mathtt{w}_i$, it is preferable to use $mw(n)$ than $mw(m)$, subject to the available resources. This is where the latency relative to a participant is useful, since it provides a better measurement about how fast is $mw(n)$ processing tasks.

We explain the intuition behind $C^{\omega}(G)$ and $C^{\omega}_{\mathtt{p}}(G)$ using the master-worker protocol, and $C(mw(n), k)$ from §4. For simplicity, we omit branching:

$$mw(n) = \mu X.\ \ \mathtt{m}_1 \to \mathtt{w}_1\{\tau \diamond c\}. \ldots \mathtt{m}_1 \to \mathtt{w}_n\{\tau \diamond c\}.\mathtt{w}_n \to \mathtt{m}_2\{\tau \diamond c\}. \ldots \mathtt{w}_1 \to \mathtt{m}_2\{\tau \diamond c\}.X$$

To simplify the calculations, we also assume that all $\mathtt{w}_i$ only do their actions after $\mathtt{m}_1$ has finished sending all tasks, although the cost models in §4 will predict a lower cost for $\mathtt{w_i}$ than $\mathtt{w_j}$ if $i < j$, since they only need to wait until their required data has been sent. We start with $mw(2)$ and $k = 1$:

$$C(mw(2),1) \leq \left[\mathtt{m_1} \mapsto 2 \times \overbrace{\mathtt{c_O}(\tau_1)}^{T_1};\mathtt{w_i} \mapsto 2 \times T_1 + \overbrace{\mathtt{c_I}(\tau_1) + c_1 + \mathtt{c_O}(\tau_2)}^{T};\mathtt{m_2} \mapsto 2 \times T_1 + T + 2 \times \underbrace{(\mathtt{c_I}(\tau_2) + c_2)}_{T_2}\right]$$

We named the relevant parts as T_1, T and T_2 for readability's sake. For any arbitrary $k > 1$, we compute the cost $C(mw(2),k)$ as:

$$C(mw(2),k) \leq \left[\begin{array}{l}\mathtt{m_1} \mapsto 2 \times T_1 + (k-1) \times 2 \times T_1;\mathtt{w_i} \mapsto 2 \times T_1 + T + (k-1) \times \max(2 \times T_1, T);\\ \mathtt{m_2} \mapsto 2 \times T_1 + T + 2 \times T_2 + (k-1) \times \max(2 \times T_1, T, 2 \times T_2)\end{array}\right]$$

There are two parts in this cost that can be distinguished, the *fixed* cost that corresponds to the initial and final stages of the protocol: $[\mathtt{m_1} \mapsto 2 \times T_1;\ \mathtt{w_i} \mapsto 2 \times T_1 + T;\ \mathtt{m_2} \mapsto 2 \times T_1 + T + 2 \times T_2]$, and the *latency*, which is the cost that increases the more iterations we take. In general, for an arbitrary n, the latency is:

$$C^{\omega}(mw(n)) = \quad [\mathtt{m_1} \mapsto n \times T_1;\ \mathtt{w_i} \mapsto \max(n \times T_1, T);\mathtt{m_2} \mapsto \max(n \times T_1, T, n \times T_2)]$$

If we keep increasing the number of workers, the latency will indicate a greater cost. However, in this particular protocol what matters is the cost *per message interaction of* $\mathtt{m}_i$, which are the workers that respectively distribute tasks and collect the results. We use $C^{\omega}_{\mathtt{m_2}}(mw(n)) = C^{\omega}(mw(n))/i$, where i is the number of message exchanged by $\mathtt{m_2}$ per iteration:

$$C^{\omega}_{\mathtt{m_2}}(mw(n)) = \left[\mathtt{m_1} \mapsto T_1;\ \mathtt{w_i} \mapsto \max(T_1, T/n);\mathtt{m_2} \mapsto \max(T_1, T/n, T_2)\right]$$

Since $C^{\omega}_{\mathtt{m_2}}(mw(n))$ is less than $C^{\omega}_{\mathtt{m_2}}(mw(m))$ if $m < n$, then the latency relative to $\mathtt{m_2}$ is a better measurement to compare how *fast* a protocol processes tasks. In the remainder of this section, we define C^{ω} and $C^{\omega}_{\mathtt{p}}$; and prove that they approximate $C(G,k)$ for a sufficiently large k.

5.1 Latency of Nested Recursive Protocols

The master-worker protocol above contains only one recursion variable. In general, recursive protocols can have multiple nested recursive sub-protocols. Intuitively, to compute C^{ω} of $\mu X.G$, we need to estimate the total execution time of a single iteration of G. If G contains recursive sub-protocols, this implies that we need to know how many iterations they will run, before recursive variable X is found. We illustrate this with the recursive global type below:

$$G = \mu X.\ \mathtt{p} \rightarrow \mathtt{q}\{\tau \diamond c_1\}.\mu Y.\ \mathtt{q} \rightarrow \mathtt{p}\ \{l_1.\ Y;\ l_2.\ X\}$$

To compute $C^{\omega}(G)$, we need to know how many times the branch that ends in recursion variable Y will be taken. Since this depends on the particular implementation of the protocol, we parameterise such recursion variables with some k, and defer its instantiation. To produce an equation to estimate the latency that is parametric in this k, we split the protocol G into two sub-protocols:

$$G_Y = \mu Y.\ \mathtt{q} \rightarrow \mathtt{p}\ \{l_1.\ Y;\ l_2.\ Z\} \quad G_X = \mu X.\ \mathtt{p} \rightarrow \mathtt{q}\{\tau \diamond c_1\}.[X/Z]G_Y$$

If $C^{\omega}(G_Y)$ contains another recursion variable, then we keep splitting it until we have a set of global types, each of which defined using at most one bound recursion variable. To compute $C^{\omega}(G_X)$ we require a parameter k, and we will use $k \times C^{\omega}([\mathtt{end}/Z]G_Y)$ for the cost of any participant in the inner sub-protocol. In the remainder of this section, we focus on recursive protocols with at most one recursion variable.

5.2 Cost Recurrences

Computing C^ω is done in two steps. First, we build a system of recurrence equations, $T(n)$, that capture the execution costs after n iterations of the recursive protocol. Then, we build the difference equations $\Delta(n)$, where $\Delta(n)(\mathbf{p}) = T(n+1)(\mathbf{p}) - T(n)(\mathbf{p})$, and estimate the value of $\Delta(n)$, as n grows. We observe that, for the recurrences that we generate, with $n \geq 2$, $\Delta(n)$ stabilises.

Definition 5.1 (Cost Recurrences). We use $C(G)$ from Def. 4.5. Given a recursive global type $\mu X.G$, we define its cost recurrence, $T(n)$, as follows: $T(n+1) = C([\mathsf{end}/X]G, T(n))$, $T(0) = []$.

Consider the following parallel pipeline:

$$G = \mu Y.\ \mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\ \mathbf{q} \to \mathbf{r}\{\tau_2 \diamond c_2\}.\ Y$$

We show below an example of the system of recurrence equations that we generate. We take the resulting cost environment, and we produce a different equation $T_\mathbf{p}$, $T_\mathbf{q}$, and $T_\mathbf{r}$ for every participant in the protocol:

$$T_\mathbf{p}(n+1) = T_\mathbf{p}(n) + \mathsf{c}_0(\tau_1) \quad T_\mathbf{q}(n+1) = \max(T_\mathbf{p}(n+1), T_\mathbf{q}(n)) + \mathsf{c}_\mathsf{I}(\tau_1) + c_1 + \mathsf{c}_0(\tau_2)$$
$$T_\mathbf{r}(n+1) = \max(T_\mathbf{q}(n+1), T_\mathbf{r}(n)) + \mathsf{c}_\mathsf{I}(\tau_2) + c_2$$

Definition 5.2 (Cost Difference Equations). Given a recursive global type $\mu X.G$, with cost recurrence T, we define its cost difference equation Δ, as $\Delta(n) = T(n+1) - T(n)$.

The cost difference equation provides an estimate on how much the cost increases for each participant after running the protocol one additional iteration.

Definition 5.3 (Latency per Iteration). The latency of a recursive protocol $\mu X.G$ with cost difference $\Delta(n)$ is defined as the cost expression c that is the least upper bound of the difference equation $\Delta(n)$, for a sufficiently large n:

$$C^\omega(\mu X.G) = c \quad \text{s.t. } \exists k, \forall n \geq k,\ c \geq \Delta(n)$$

Suppose that we want to compute the latency of the previous parallel pipeline. On average, excluding the initialisation of the protocol, the latency for $\mathbf{r}$ must be the maximum of the times for $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$, as usual in parallel pipelines. This is because the actions of $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ are independent *across iterations*. The solution of $\Delta(0)$ shows that the cost is the addition of all individual costs. However, by solving $\Delta(1)$, we obtain the expected result, where $T_\mathbf{p} = \mathsf{c}_0(\tau_1)$, $T_\mathbf{q} = \max(T_\mathbf{p}, \mathsf{c}_\mathsf{I}(\tau_1) + c_1 + \mathsf{c}_0(\tau_2))$, and $T_\mathbf{r} = \max(T_\mathbf{q}, \mathsf{c}_\mathsf{I}(\tau_2) + c_2)$.

When the actions of a recursive protocol are not independent across iterations, i.e. the send/receive dependency graph forms a cycle, then all participants will need to synchronise. An example of this is the protocol:

$$G = \mu Y.\ \mathbf{p} \to \mathbf{q}\{\tau_1 \diamond c_1\}.\ \mathbf{q} \to \mathbf{p}\{\tau_2 \diamond c_2\}.\ Y$$

In the first iteration, we will have that $\mathbf{p}$ sends τ_1 to $\mathbf{q}$, which needs to wait for the message, and then takes c_1 time. At this point, we have that $\mathbf{p}$ spent $\mathsf{c}_0(\tau_1)$, and $\mathbf{q}$ took $\mathsf{c}_0(\tau_1) + \mathsf{c}_\mathsf{I}(\tau_1) + c_1$. Next, $\mathbf{q}$ sends τ_2 to $\mathbf{p}$, which is completed after $\mathsf{c}_0(\tau_1) + T_\mathbf{q}$, where $T_\mathbf{q} = \mathsf{c}_\mathsf{I}(\tau_1) + c_1 + \mathsf{c}_0(\tau_2)$. Then, $\mathbf{p}$ needs to receive τ_2 and take c_2 of local computation time. Since the accumulated time by $\mathbf{p}$ is $\mathsf{c}_0(\tau_1) < \mathsf{c}_0(\tau_1) + T_\mathbf{q}$, we increase the total time spent by $\mathbf{p}$: $T_\mathbf{q} + T_\mathbf{p}$, where $T_\mathbf{p} = \mathsf{c}_0(\tau_1) + \mathsf{c}_\mathsf{I}(\tau_2) + c_2$. In the next iteration, we have that $\mathbf{p}$ takes $T_\mathbf{p} + T_\mathbf{q} + \mathsf{c}_0(\tau_1)$. Next, $\mathbf{q}$ takes $\max(\mathsf{c}_0(\tau_1) + T_\mathbf{q}, T_\mathbf{p} + T_\mathbf{q} + \mathsf{c}_0(\tau_1)) + T_\mathbf{q} = T_\mathbf{p} + T_\mathbf{q} + \mathsf{c}_0(\tau_1) + T_\mathbf{q}$, and finally $\mathbf{p}$ will take $2 \times (T_\mathbf{p} + T_\mathbf{q})$. After k iterations, the cost for $\mathbf{p}$ is $k \times (T_\mathbf{p} + T_\mathbf{q})$, while the cost for $\mathbf{q}$ is $(k-1) \times (T_\mathbf{p} + T_\mathbf{q}) + \mathsf{c}_0(\tau_1) + T_\mathbf{q}$, which approximates $T_\mathbf{p} + T_\mathbf{q}$.

Definition 5.4 (Latency with respect to $\mathbf{p}$). We define $C^\omega_\mathbf{p}(\mu X.G) = C^\omega(\mu X.G)(\mathbf{p})/\mathsf{count}(\mathbf{p}, G)$, where $\mathsf{count}(\mathbf{p}, G)$ is the number of interactions of G in which $\mathbf{p}$ occurs.

5.3 Correctness

We guarantee that the latency correctly approximates the bounded cost of a protocol. Moreover, given an arbitrary trace that is the result of a k-unrolling of a recursive global type G, $k \times C^\omega(G)$ will approximate the cost of the full trace.

Theorem 5.5 (Cost Latency Correspondence). *Given a sufficiently large k_2, for all $k_1 > k_2$, $C(G, k_1) - C(G, k_2) \leq (k_1 - k_2) \times C^\omega(G)$.*

This result follows directly from our definition of C^ω, since the latency approximates $\Delta(n)$ (with $\Delta(n) = T(n+1) - T(n)$) for a sufficiently large n, and that $T(n+1)$ is the recurrence that approximates $C(G, n)$. We need to show that $k \times \Delta(n) = T(n+k) - T(n)$, and then take $k_2 = n$ and $k_1 = n + k$.

Proposition 5.6. *Given $\mu X.G$, let $T(n{+}1) = C([\mathsf{end}/X]G, T(n))$ and $T(0) = []$. Then, $C(\mu X.G, n) = T(n)$.*

Proof. By induction on n, the base case is straightforward: $T(0) = C(\mu X.G, 0) = C(\mathsf{end}) = []$. If $n = m{+}1$, then $T(m{+}1) = C([\mathsf{end}/X]G, T(m)) = C([\mathsf{end}/X]G, C(\mu X.G, m)) = C(\mu X.G, m{+}1)$. □

Proposition 5.6 states that given a recursive protocol, instantiating its recurrence with some number n yields the same cost as unrolling the protocol n times and computing its cost. We use Proposition 5.6 in combination with Definition 5.2 to derive the following. Assume Δ is the difference equation for recursive protocol G. Then, the following equality holds:

$$\Delta(n) = C(G, n+1) - C(G, n) \tag{1}$$

Theorem 5.7 (Latency Soundness). *There exists k' such that for all k, if $\mathsf{unfold}^k(G) \xrightarrow{\vec{\alpha}} \mathsf{end}$, then $C(\vec{\alpha}) \leq k \times C^\omega(G) + k'$.*

Proof. By Definition 5.2, we know that there exists some k_0 such that for all $n \geq k_0$,

$$C^\omega(G) \geq \Delta(n). \tag{2}$$

We show that k' is $C(G, k_0)$. By Theorem 4.12, we know that $C(\vec{\alpha}) \leq C(G, k)$. Therefore, it is sufficient to show that for all k, $C(G, k) \leq k \times C^\omega(G) + C(G, k_0)$. We proceed by case analysis:
Case $k \leq k_0$ straightforward, since $C(G, k) \leq C(G, k_0)$ if $k \leq k_0$.
Case $k > k_0$: By induction on k. All cases $\leq k_0$ are straightforwardly true.

- Case $k = k_0 + 1$ follows from $C(G, k_0) \leq C(G, k_0 + 1)$:

$$\begin{aligned}
C(G, k_0) &\leq C(G, k_0 + 1) && \{\text{multiply } k_0\}\\
k_0 \times C(G, k_0) &\leq k_0 \times C(G, k_0 + 1) && \{\text{add } C(G, k_0 + 1)\}\\
C(G, k_0 + 1) + k_0 \times C(G, k_0) &\leq C(G, k_0 + 1) + k_0 \times C(G, k_0 + 1) && \{\text{sub } k_0 \times C(G, k_0)\}\\
C(G, k_0 + 1) &\leq (k_0 + 1) \times C(G, k_0 + 1) - k_0 \times C(G, k_0) && \{\text{cancel } C(G, k_0)\}\\
C(G, k_0 + 1) &\leq (k_0 + 1) \times (C(G, k_0 + 1) - C(G, k_0)) + C(G, k_0) && \{\text{by (1) and (2)}\}\\
C(G, k_0 + 1) &\leq (k_0 + 1) \times C^\omega(G) + C(G, k_0)
\end{aligned}$$

- $k = k_2 + 1$, with $k_2 > k_0$. Assume the induction hypothesis $C(G, k_2) \leq k_2 \times \Delta(k_2) + C(G, k_0)$:

$$\begin{aligned}
C(G, k_2) &\leq k_2 \times \Delta(k_2) + C(G, k_0) && \{\text{by (1)}\}\\
C(G, k_2) &\leq k_2 \times C(G, k_2 + 1) - k_2 \times C(G, k_2) + C(G, k_0) && \{\text{add } C(G, k_2 + 1)\}\\
C(G, k_2 + 1) + C(G, k_2) &\leq (k_2 + 1) \times C(G, k_2 + 1) - k_2 \times C(G, k_2) + C(G, k_0) && \{\text{sub } C(G, k_2)\}\\
C(G, k_2 + 1) &\leq (k_2 + 1) \times (C(G, k_2 + 1) - C(G, k_2)) + C(G, k_0) && \{\text{by (1) and (2)}\}\\
C(G, k_2 + 1) &\leq (k_2 + 1) \times C^\omega(G) + C(G, k_0)
\end{aligned}$$

□

This implies that the latency approximates the cost of a trace of a k-unrolling of a recursive protocol, and it follows from Theorems 5.5 and 4.12. To illustrate this, consider the average cost per recursion iteration, $C(\vec{\alpha})/k$. By Theorem 5.7, we know that $C(\vec{\alpha})/k \leq C^{\omega}(G) + k'/k$. Since k' does not depend on k, for a sufficiently large k, the term k'/k will become smaller, and the upper bound of $C(\vec{\alpha})/k$ will be approximately $C^{\omega}(G)$.

6 ASYNCHRONOUS MESSAGE OPTIMISATION

This section illustrates one of the key features of CAMP, the formulation and its soundness of *asynchronous message optimisations*. We extend the cost equations in §4 and §5 to tackle protocols in which certain actions have been permuted for optimisation purposes. Parallel programs often make use of parallel pipelines to overlap computation and communication, as far as the overlapping does not interfere with data dependencies. The overlapping can reduce stall time due to blocking wait in the asynchronous communication model. Under the CAMP theory, optimisation should preserve the deadlock-freedom and produce the same outcome, while ensuring less cost for the overall calculation.

Fig.5 shows a safe and efficient ring protocol, in which stage i shares data with stage $(i + 1) \mod 3$, and then proceed to do some local computation. This protocol behaves similarly to that of Fig. 2 in §3, but the output actions have been permuted so that they are performed first, thus reducing the amount of synchronisation required. The optimised version, however, is more difficult to check against a standard global type, because of the permuted actions. This can be illustrated by comparing the optimised and un-optimised local types of $\mathbf{q}$:

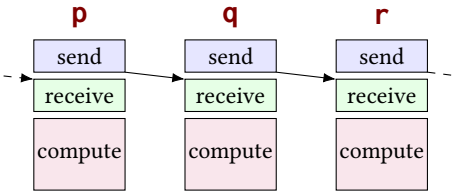


Fig. 5. Optimised ring trace.

$$L_{\mathbf{q}} = \mu X.\mathbf{p}\,?\,\tau{\diamond}c.\mathbf{r}\,!\,\tau.X \qquad L'_{\mathbf{q}} = \mu X.\mathbf{r}\,!\,\tau.\mathbf{p}\,?\,\tau{\diamond}c.X$$

$L_{\mathbf{q}}$ is the unoptimised local type, and $L'_{\mathbf{q}}$ is the optimised version. Both local types represent a similar communication pattern. However, in the left version $L_{\mathbf{q}}$, the send action only happens after receiving, and computing (with cost c), while the right version first sends a value of type τ, and then performs the receive and local computation. This removes unnecessary synchronisation, and allows $\mathbf{r}$ to continue with its interactions before $\mathbf{q}$ finishes its own local computation.

Only certain message permutations are valid. For example, if instead of swapping the send and receive actions for $L_{\mathbf{q}}$, we permute the actions for participant $\mathbf{p}$, then we end up in the following (incorrect) situation:

$$L'_{\mathbf{p}} = \mu X.\mathbf{r}\,?\,\tau{\diamond}c.\mathbf{q}\,!\,\tau.X \qquad L'_{\mathbf{q}} = \mu X.\mathbf{p}\,?\,\tau{\diamond}c.\mathbf{r}\,!\,\tau.X \qquad L'_{\mathbf{r}} = \mu X.\mathbf{p}\,?\,\tau{\diamond}c.\mathbf{p}\,!\,\tau.X$$

This is a clear deadlock situation, since all participants are waiting for a message from each other. To avoid such situations, we define the *Asynchronous Message Optimisation* for global types, and show its soundness:

Definition 6.1 (Asynchronous Message Optimisation). We first extend the syntax of global types to include send (!) and receive (?) actions as: $G ::= \mathbf{pq}\,!\,\{\tau\}.G \mid \mathbf{pq}\,?\,\{\tau\}.G \mid \ldots$ The *asynchronous optimisation* relation, $G_1 \preceq G_2$ (read: G_1 is more optimal than G_2), with $\mathbf{p}_1 \neq \mathbf{p}_2$ or $\mathbf{q}_1 \neq \mathbf{q}_2$ is the transitive closure of the rules below:

$$
\begin{array}{ll}
\text{[Init]} & \mathbf{pq}\,!\,\{\tau\}.\mathbf{qp}\,?\,\{\tau\}.G \preceq \mathbf{p} \rightarrow \mathbf{q}\{\tau\}.G \\
\text{[Out]} & \mathbf{p_1q_1}\,!\,\{\tau_1\}.\mathbf{p_2q_2}\,!\,\{\tau_2\}.G \preceq \mathbf{p_2q_2}\,!\,\{\tau_2\}.\mathbf{p_1q_1}\,!\,\{\tau_1\}.G \\
\text{[In]} & \mathbf{p_1q_1}\,?\,\{\tau_1\}.\mathbf{p_2q_2}\,?\,\{\tau_2\}.G \preceq \mathbf{p_2q_2}\,?\,\{\tau_2\}.\mathbf{p_1q_1}\,?\,\{\tau_1\}.G \\
\text{[Opt]} & \mathbf{p_1q_1}\,!\,\{\tau_1\}.\mathbf{q_2p_2}\,?\,\{\tau_2\}.G \preceq \mathbf{q_2p_2}\,?\,\{\tau_2\}.\mathbf{p_1q_1}\,!\,\{\tau_1\}.G \\
\text{[OBra]} & \mathbf{p_2q_2}\,!\,\{\tau_2\}.\mathbf{p_1} \rightarrow \mathbf{q_1}\{l_i.G_i\}_{i\in I} \preceq \mathbf{p_1} \rightarrow \mathbf{q_1}\{l_i.\mathbf{p_2q_2}\,!\,\{\tau_2\}.G_i\}_{i\in I} \\
\text{[IBra]} & \mathbf{p_1} \rightarrow \mathbf{q_1}\{l_i.\mathbf{q_2p_2}\,?\,\{\tau_2\}.G_i\}_{i\in I} \preceq \mathbf{q_2p_2}\,?\,\{\tau_2\}.\mathbf{p_1} \rightarrow \mathbf{q_1}\{l_i.G_i\}_{i\in I} \\
\text{[Cong]} & G \preceq G' \Rightarrow \quad E[G] \preceq E[G']
\end{array}
$$

where $E ::= [\,] \mid \mathbf{p} \rightarrow \mathbf{q}\{\tau\}.E \mid \mathbf{p} \rightarrow \mathbf{q}(\{l.E\} \cup \{l_k.G_k\}_{k\in K}) \mid \mu X.E.$

The optimisation starts first splitting, by [Init], the message to a sending and receiving operation; [Out] permute two outputs to two different participants; [In] is its dual; [OBra] permutes a send and a branch; [IBra] is dual; and [Cong] is a congruence rule. The key rules are [Opt], [OBra] and [IBra], that perform permutations that allow communication and computation to overlap. This is because the rules permute send actions to the left, and receive actions to the right. We prove that whenever G_2 is deadlock-free, then $G_1 \preceq G_2$ must also be deadlock free. Moreover, we show that $G_1 \preceq G_2$ is decidable. Notice that: (1) our definition is different from the literature asynchronous subtyping for session types, motivated from more practical use cases; (2) our cost models can be applied even whenever we do not have that $G_1 \preceq G_2$, in which case, safety can be guaranteed by using any method from the literature. See §9.

Theorem 6.2 (Asynchronous Message Optimisation).

(1) (Soundness) *Suppose G_2 is a deadlock-free global type and $G_1 \preceq G_2$. Then G_1 is deadlock-free.*
(2) (Decidability) *Given G_1 and G_2, it is decidable whether $G_1 \preceq G_2$ or not.*

Proof. (1) By induction on the derivation of $G_1 \preceq G_2$. Assume $G_1 \preceq G_2$ and G_2 is deadlock-free and $C_i = [\mathbf{p} \mapsto G_i \upharpoonright \mathbf{p}]_{\mathbf{p}\in\mathcal{P}}$ with $i = 1, 2$. We prove if $\langle C_2, Q\rangle$ is deadlock-free, then $\langle C_1, Q\rangle$ is deadlock-free. To do this proof, we extend the projection for global types as follows. $\mathbf{pq}!\{\tau\}.G \upharpoonright \mathbf{r} = \mathbf{q}!\{\tau\}.(G \upharpoonright \mathbf{r})$ if $\mathbf{p} = \mathbf{r}$, and $G \upharpoonright \mathbf{r}$ otherwise. $\mathbf{pq}\,?\,\{\tau\}.G \upharpoonright \mathbf{r} = \mathbf{q}\,?\,\{\tau\}.(G \upharpoonright \mathbf{r})$ if $\mathbf{p} = \mathbf{r}$, and $G \upharpoonright \mathbf{r}$ otherwise. All cases except [Opt] is obvious. The [Opt] states: $\mathbf{p_1q_1}\,!\,\{\tau_1\}.\mathbf{q_2p_2}\,?\,\{\tau_2\}.G \preceq \mathbf{q_2p_2}\,?\,\{\tau_2\}.\mathbf{p_1q_1}\,!\,\{\tau_1\}.G$. We know that $\langle C_2, Q\rangle$ is deadlock free. Note that $\mathbf{p}_i \neq \mathbf{q}_i$, otherwise G_2 cannot be proven deadlock free (it is either ill-formed, or the optimisation of an ill-formed global type). There are two cases, considering the side conditions for the rules: a) if $\mathbf{p}_1 \neq \mathbf{q}_2$, straightforward since these subject of both interactions are different; b) if $\mathbf{p}_1 = \mathbf{q}_2 = \mathbf{p}$, then we have $C_1 \setminus \mathbf{p} = C_2 \setminus \mathbf{p}$, $C_2(\mathbf{p}) = \mathbf{p}_2\,?\,\{\tau_2\}.\mathbf{q}_1\,!\,\{\tau_1\}.L$, and $C_1(\mathbf{p}) = \mathbf{q}_1\,!\,\{\tau_1\}.\mathbf{p}_2\,?\,\{\tau_2\}.L$. Since $\langle C_2, Q\rangle$ is deadlock free, then $Q(\mathbf{p}_2\mathbf{p}) = w \cdot \tau_2$. Therefore, $\langle C_1, Q\rangle \xrightarrow{*} \langle C_1', Q'\rangle$, $\langle C_2, Q\rangle \xrightarrow{*} \langle C_2', Q'\rangle$, with $C_1'(\mathbf{p}) = C_2'(\mathbf{p}) = L$, and $Q'(\mathbf{p}_2\mathbf{p}) = w$. Since $\langle C_2, Q\rangle$ is deadlock free, then $\langle C_2', Q'\rangle$ must also be deadlock free, and $\langle C_1', Q'\rangle$ as well.

(2) We consider a normal form which is derived applying [Out,In] with the side condition $\mathbf{p}_1 < \mathbf{p}_2$; and all other rules except [Init] as much as possible until no rule is applicable, and finally applying [Init] to all pairs of send/receive. Then if $G_1 \preceq G_2$, there exists a unique global type G such that $G_i \preceq G$ derivable applying the above rules finitely. This means interpreting $G_1 \preceq G_2$ as a term rewriting system. The term rewriting system is terminating because a) the terms are finite, since we do not unroll recursion; and b) the only potential rewrite cycle appears in rules [Out,In], which is prevented by the additional side condition that $\mathbf{p}_1 < \mathbf{p}_2$ The repeated application of these rules permute the send and receive actions to their rightmost and leftmost position respectively. By the side conditions of the rules, it is straightforward to show that any critical pairs can be unified, since no rule can prevent another rule from being applied. □

Finally, we prove that, ignoring sending costs, if $G_1 \preceq G_2$, then the cost of G_1 is less than the cost of G_2. The reason why we need to ignore sending costs for this proof is that permuting two output actions may introduce delays in a later computation stage. Note that this property is a statement about the *synchronisation* costs, not an algorithm for optimising a protocol. To illustrate this case, consider the following global types:

$$G_1 = \mathbf{pq_1}\,!\,\tau.\mathbf{pq_2}\,!\,\tau.\mathbf{q_1p}\,?\,\tau.\mathbf{q_2p}\,?\,\tau.\mathbf{q_1}\diamond c_1.\mathbf{q_2}\diamond c_2.\mathbf{q_2r}\,!\,\tau'.\mathbf{rq_2}\,?\,\tau'.\text{end}$$
$$G_2 = \mathbf{pq_2}\,!\,\tau.\mathbf{pq_1}\,!\,\tau.\mathbf{q_1p}\,?\,\tau.\mathbf{q_2p}\,?\,\tau.\mathbf{q_1}\diamond c_1.\mathbf{q_2}\diamond c_2.\mathbf{q_2r}\,!\,\tau'.\mathbf{rq_2}\,?\,\tau'.\text{end}$$

It is clear that $G_1 \preceq G_2$, by [Out]. However, whenever $c_2 \geq c_1$, then $C(G_1) \geq C(G_2)$, since $\mathbf{q}_2$ must wait longer in G_1 than in G_2 before receiving the message of type τ. Note that, even if $c_1 = c_2$, the cost of the global protocol will be greater in G_1, since $\mathbf{r}$ is the participant that takes longer in the protocol, and needs to wait for $\mathbf{q}_2$. The implications of this result are twofold: (a) we know that whenever $G_1 \preceq G_2$, G_1 contains *less* overhead due to synchronisation; and (b) for a given G_2, choosing an optimal $G_1 \preceq G_2$ is not straightforward, and depends on actual local computation costs and communication latencies.

Theorem 6.3 (Optimisation Cost). *Suppose G_2 is a well-formed global type and $G_1 \preceq G_2$. If the sending cost is 0, then $C(G_1) \leq C(G_2)$.*

Proof. By induction on the derivation of $G_1 \preceq G_2$. Most cases are permutations of independent interactions, and all independent interactions can be permuted with no effect on the cost. Since we assume zero send costs, the cost of sending two actions is the same, independently of the order. The reasoning is similar for receiving interactions. The only rules that we need to consider are [Opt], [OBra] and [IBra]. Notice that in all the cases, the left hand side contains a sending (or choice) at an earlier position than the right hand side. We show the proof for case [Opt], but all cases follow a similar structure. The cost of $\mathbf{p_1q_1}\,!\,\{\tau_1\}.\mathbf{p_2q_2}\,?\,\{\tau_2\}.G$ is the cost of G, where the message queue for $\mathbf{p_1q_1}$ contains the current execution time for $\mathbf{p}_1$. If $\mathbf{p}_1 \neq \mathbf{p}_2$, then the cost will be the same in both cases. But if $\mathbf{p}_1 = \mathbf{p}_2$, then the cost in the right hand side will contain the accumulated cost for $\mathbf{p}_1$, plus the cost of receiving from $\mathbf{q}_2$. Since the costs recorded at the message queue are greater, then the cost of the continuation must also be greater. □

We illustrate how this optimisation reduces synchronisation time with one iteration of a ring protocol of size 2: $\mathbf{p} \to \mathbf{q}\{\tau_1\diamond c_1\}.\mathbf{q} \to \mathbf{p}\{\tau_2\diamond c_2\}.\text{end}$. The only possible trace for running such protocol is: $\mathbf{pq}\,!\,\tau_1 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q}\diamond c_1 \cdot \mathbf{qp}\,!\,\tau_2 \cdot \mathbf{qp}\,?\,\tau_2 \cdot \mathbf{p}\diamond c_2$. This trace and the derived cost imply that computation costs c_1 and c_2 cannot happen in parallel: $T_{\mathbf{p}} = \mathtt{c}_0(\tau_1) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1 + \mathtt{c}_0(\tau_2) + \mathtt{c}_\mathtt{I}(\tau_2) + c_2$.
In cases where such interactions are independent, we can permute the send/receive actions of $\mathbf{q}$ to remove the synchronisation cost from $\mathbf{p}$, and allow *any trace that is an interleaving of the following sub-traces*, where the send operations happen before the matching receive:

$$tr_{\mathbf{p}} = \mathbf{pq}\,!\,\tau_1 \cdot \mathbf{qp}\,?\,\tau_2 \cdot \mathbf{p}\diamond c_2 \qquad tr_{\mathbf{q}} = \mathbf{qp}\,!\,\tau_2 \cdot \mathbf{pq}\,?\,\tau_1 \cdot \mathbf{q}\diamond c_1$$

Such optimisations is represented by the following type: $\mathbf{pq}\,!\,\{\tau_1\}.\mathbf{qp}\,!\,\{\tau_2\}.\mathbf{qp}\,?\,\{\tau_1\diamond c_1\}.\mathbf{pq}\,?\,\{\tau_2\diamond c_2\}.\text{end}$.
This scenario will have the cost that we show below, which is smaller than the original cost.

$$T_{\mathbf{p}} = \max(\mathtt{c}_0(\tau_1), \mathtt{c}_0(\tau_2)) + \mathtt{c}_\mathtt{I}(\tau_2) + c_2 \quad T_{\mathbf{q}} = \max(\mathtt{c}_0(\tau_1), \mathtt{c}_0(\tau_2)) + \mathtt{c}_\mathtt{I}(\tau_1) + c_1$$

7 IMPLEMENTATION

We implemented a library in Haskell for describing global types augmented with size and cost information, from which we can derive cost equations for protocols.

7.1 Resource Contention

CAMP addresses the issue that multiple participants may need to share computational resources. We model the cases in which the participants of a protocol are mapped to distinct *nodes* of a distributed system, where each node may contain multiple *cores*. This requires: a) a *target hardware* specification, and b) a *mapping* from participants to nodes. The target hardware specification describes the amount of nodes available, the cores per-node, and the communication latencies between nodes. The mapping from participants to nodes assigns each participant of the distributed system to a different node. Our assumptions are: a) there is no mechanism for process migration; b) processes can be pinned to specific nodes, but not to specific cores; and c) an optimistic scheduling scenario, in which participants will run as soon as possible, whenever a core becomes available.

Definition 7.1 (Target Hardware Specification). The target hardware is specified as an indexed set of node descriptions, and the communication latencies between nodes: $\{C_n\}_{n\in N}$ and $\{L_{n_1n_2}\}_{n_1,n_2\in N}$. Here, N is the set of *node identifiers*, C_n is a natural number that describes the number of available cores for node n, and $L_{n_1n_2}$ is a function from a size to the amount of time it takes to transmit a value from n_1 to n_2.

Definition 7.2 (Participant Mapping). The participant mapping associates each participant with a specific node. We say that participants are *pinned* to nodes $M : \mathcal{P} \to N$.

For example, consider the master-worker example, where we have 1 master and 5 workers:

$$\mu X.\ \mathtt{m} \to \mathtt{w_1}\{\tau_1\}.\ldots.\mathtt{m} \to \mathtt{w_5}\{\tau_1\}.\mathtt{w_1} \to \mathtt{m}\{\tau_2\}.\ldots.\mathtt{w_5} \to \mathtt{m}\{\tau_2\}.X$$

First, we need to know which is the target hardware. In our case, this is a distributed system with two nodes, n_1 and n_2, with 1 and 4 cores respectively. That is: with $C_{n_1} = 1$ and $C_{n_2} = 4$. Suppose that the communication latency between n_1 and n_2 is a known function on the size of the messages, l. Then, $L_{n_1n_2} = L_{n_2n_1} = l$. Our hardware description is completed by $\{C_n\}_{n\in\{n_1,n_2\}}$, and $\{L_{nn'}\}_{n,n'\in\{n_1,n_2\}}$. Finally, we require to map our participants to the different nodes in the architecture. In our example, we may want to run $\mathtt{m}$ in n_1, and $\mathtt{w}_i$ in n_2: $M(\mathtt{m}) = n_1$ and $M(\mathtt{w}_i) = n_2$.

To compute the cost in this specific scenario, we use the *resource bounded* cost equations. The key difference is that, as well as keeping track of the accumulated time per-role, we keep the accumulated time per node, using a *core-availability* time, which is the earliest time at which a core becomes available. The resource-bounded cost equations are obtained using $C(G)(T, S, W)$, where S accumulates the cost at each core and each node of the system. We assume a hardware specification and mapping. The rules are now modified in the following way:

$$C(\mathtt{p} \to \mathtt{q}\{\tau\}.G, \vec{k})(T, S, W) = C(G, \vec{k})(T[\mathtt{p} \mapsto S_1(M(\mathtt{p})_c), \mathtt{q} \mapsto S_2(M(\mathtt{q})_c)], S_2, W$$

where $S_1 = S[M(\mathtt{p})_{c_1} \hookrightarrow \mathtt{c_O}(\tau)]$, $S_2 = S_1[M(\mathtt{q})_{c_2} \hookrightarrow \mathtt{c_I}(\tau) + L_{M(\mathtt{p})M(\mathtt{q})}]$, $\forall c, S[M(\mathtt{p})_{c_1}] \leq S[M(\mathtt{p})_c]$ and $\forall c, S_1[M(\mathtt{q})_{c_2}] \leq S[M(\mathtt{q})_c]$. In this definition, we update the accumulated cost of $\mathtt{p}$ and $\mathtt{q}$ to the total accumulated cost of the lowest cost core of the node to which they are mapped. The definition of $S[n_c \hookrightarrow c]$ is the same as in §4.

7.2 A Monadic Interface for Global Types

We develop a deep embedding of the global types of §3 in Haskell, and provide a monadic interface on top as a simpler interface for representing protocols. We call this monadic interface GTM, for *Global Type Monad*. In GTM, there is an implicit end at the end of each sequence of interactions. An interaction is specified using function message, and participants are created using mkRole. Function gclose runs the code in the GTM monad, and produces the resulting global type (CGT). We show below the Haskell code that generates an n-stage pipeline, and a recursive 2-stage pipeline generated using the following code:

```
pipe :: [(SType, Cost)] → Role → GTM () → GTM ()
pipe []          p k = k
pipe ((t, c) : r) p k = mkRole >>= λq → message p q t (cost t) >> pipe r q k

rpipe2 :: CGT
rpipe2 = gclose $ mkRole >>= λr → grec $ λx → pipe [(t1, c1), (t2, c2)] r x
```

The code for rpipe2 produces the following global type: $\mu X.\mathbf{p} \rightarrow \mathbf{q}\{\texttt{t1}\diamond\texttt{c1}\}.\mathbf{q} \rightarrow \mathbf{r}\{\texttt{t2}\diamond\texttt{c2}\}.X$. Notice that embedding a global type language in Haskell allows us to compute topologies based on any input parameters, such as the number of stages of a pipeline, that would otherwise require the use of extensions to MPST, e.g. parameterised roles [Castro et al. 2019; Deniélou et al. 2012]. However, to check well-formedness, we need to instantiate the parameters.

We provide functions cost and latency, both of type CGT → Time, to compute the set of equations that describe the cost (latency) of an input global type. To obtain a particular prediction, the user needs to provide an instantiation of all free size and cost variables in the equations, including the transmission costs between participants.

8 EVALUATION

This section presents a number of benchmarks used to evaluate the predictive power of CAMP. Our benchmarks are taken from multiple different sources, mostly MPST-based tools [Castro et al. 2019; Castro-Perez and Yoshida 2020b; Imai et al. 2020; Ng et al. 2015; Zhou et al. 2020], but also a subset of the Savina actor benchmarking suite [Imam and Sarkar 2014]. We categorise our benchmarks following the structure of the Savina benchmarking suite: (i) microbenchmarks, (ii) concurrency benchmarks, and (iii) parallel algorithms. Microbenchmarks focus on different structures and protocols, and are aimed at testing and evaluating the different features of CAMP. Concurrency benchmarks are aimed at evaluating the impact on communication and synchronisation. This can be useful to, e.g. estimate server response times, and set the appropriate timeouts in larger systems. In the context of parallel algorithms, the main use of the cost models is to predict the parallel speedups achieved by a particular parallelisation, without needing to run or profile the application.

8.1 Methodology

We follow a series of steps in order to make our results as consistent as possible. We will detail now these steps, highlighting which part is automated, and which needs to be provided by the developer. Our methodology is divided in two parts: (1) characterising the target architecture; and (2) benchmark cost analysis.

Characterising the target architecture. To tailor a cost analysis to a specific target architecture, we need to characterise the costs of sending/receiving data between nodes. This requires three steps: (1) specifying the amount of nodes, and the amount of processors/cores per node; (2) estimate message latencies between nodes; and, (3) profiling send/receive operations in the required languages/frameworks with inputs of different sizes.

We require the results of these steps to be stored in a `.hs` file, as an architecture description, that will be imported and used by CAMP's cost models. These steps must be performed only once per architecture and programming language.

Additionally to our theory, the implementation allows programmers to specify an overhead for running multiple participants in a single node. This is to account for all factors that CAMP is currently not considering for deriving cost equations. See §10 for a discussion.

Benchmark cost analysis. This is the main part of the cost analysis. This part *does not require that the target application is implemented using an* MPST*-based framework*. Assume that we start with a target application, already implemented. The steps of our methodology are the following:

(1) **Write its global type.** Since most of our benchmarks are derived from implementations in other MPST-based tools, this step is straightforward. For non MPST-based implementations, the developer needs to analyse the communication protocol and write it as a global type.
(2) **Extract the sequential parts.** The sequential parts must be extracted as self-contained implementations, that can be run independently of the whole distributed system.
(3) **Run the profiler on the sequential parts.** Our profiler requires multiple input sizes, measures the execution costs of the sequential parts on these input sizes, and performs cubic spline interpolation on the gathered data. Note that the sequential cost is only valid for inputs of sizes that are within the measured range. This part can be omitted when using a static cost analysis, or the cost equations are known and provided manually.
(4) **Annotate the global type** and extract cost equations.
(5) **Instantiate the cost** by feeding the profiling information for both the target architecture and the sequential parts.

8.2 Benchmark Structure

We list and provide a brief explanation of all the benchmarks that we used for the cost models. We used two different hardware configurations for the evaluation. We name them `Arch1` and `Arch2`: `Arch1` is a 4-core Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with hyperthreading, and `Arch2` comprises 2 NUMA nodes, 12 cores per node and 62GB of memory, using Intel Xeon CPU E5-2650 v4 @ 2.20GHz chips. `Arch2` is an HPC cluster that uses PBS queuing mechanism. We made sure that we consistently selected the same hardware for every execution. In the remainder of this section, we will specify whether the benchmarks were run on `Arch1` or `Arch2`.

We used the benchmarks as defined in the different sources from where we took the source code. Overall, we used averages of > 50 repetitions for benchmarks with large computation costs, and linear regression (95% CI) for smaller (micro-benchmarks such as ping-pong, all-to-all, etc).

Microbenchmarks. Recursive Ping-Pong is the recursive ping-pong example. We run both the Scala benchmark (**`pp-akka`**) from the Savina benchmarking suite [Imam and Sarkar 2014], and the OCaml version taken from Imai et al. [2020] on `Arch1`, on three different transports (**`pp-ev`**, **`pp-lwt`**, **`pp-ipc-`***n*). Since the cost of sending in the **`ipc`** transport depends on the input size, we use *n* to differentiate different runs of this benchmark with different input sizes. We introduced an arbitrary computation to the Scala version to increase the local computation costs. ***Thread Ring*** (**`ring`**) is the Scala version from Imam and Sarkar [2014], both with and without asynchronous message optimisations, on `Arch1`. ***Counting Actor*** (**`count`**) is a benchmark with two actors, one of which counts the number of messages received from the other. This is the Savina microbenchmark Imai et al. [2020] on `Arch1`. ***One-to-All***, ***All-to-One*** and ***All-to-All***: we use the Go one-to-all, all-to-one and all-to-all Go implementations (**`1a`**, **`a1`** and **`aa`**) in [Castro et al. 2019], all run on `Arch1`.

Concurrency Benchmarks. Two-Buyer Protocol (**`twobuy`**). We use an F⋆ implementation taken from [Zhou et al. 2020], and extracted into OCaml. ***Sleeping Barber***x, ***Dining Philosophers*** and ***Cigarette Smoker*** (**`barb`**, **`dphil`**, **`csmok`**). These are the Savina Akka benchmarks in [Imam and Sarkar 2014], run on `Arch1`. ***K-Nucleotide***, ***Spectral-Norm*** and ***Regex-DNA*** (**`kn`**, **`sn`**, **`dna`**). These benchmarks are Go implementations taken from [Castro et al. 2019], based on the Computer Language Benchmarks Game, and use different combinations of scatter, gather, choices and recursion.

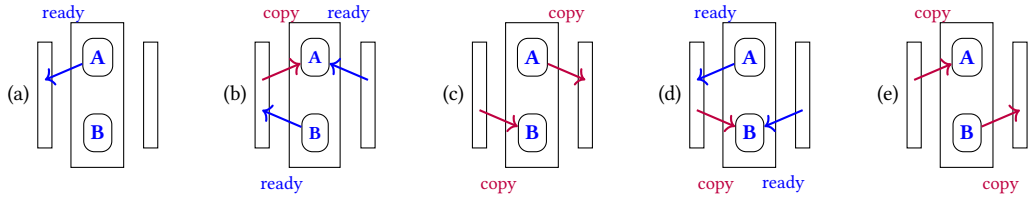


Fig. 6. Double-Buffering

Parallel Algorithms. All these benchmarks were run on `Arch2`, and they were taken from two sources: Ng et al. [2015](NBody, linear equation solver, wordcount and adpredictor) and Castro-Perez and Yoshida [2020b](dot product, fast fourier transform and mergesort).

Ng et al. [2015]. The work by Ng et al. [2015] has implemented representative parallel benchmarks from [Asanovic et al. 2009]. ***NBody*** (`nb`) is a 2D NBody simulation in C+MPI which is implemented as a thread ring with asynchronous communication optimisations. ***Linear equation solver*** (`ls`) is parallelised using a wraparound mesh. Similarly to the NBody example, we required the extension with asynchronous communication optimisation. ***WordCount*** (`wc`) and ***AdPredictor*** (`ap`) are parallelised using map-reduce.

Double-Buffering Algorithm. (`dbuff`) [USENIX 2020] is a well-known technique for increasing the throughput of a device that has two buffers. To accurately represent a double-buffering protocol, we use CAMP's extension with asynchronous message optimisations. We show the protocol below, using participants `p` for *source*, `q` for *sink* and `r` for the *service*:

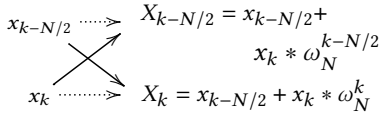
$$\texttt{rp}!\{r_1\}.\texttt{rp}!\{r_2\}.\mu X.\ \texttt{pr}?\{r_1\}.\texttt{p} \to \texttt{r}\{s_1\}.\texttt{q} \to \texttt{r}\{t_1\}.\texttt{r} \to \texttt{q}\{u_1\}.\texttt{rp}!\{r_1\}.\texttt{pr}?\{r_2\}.\ \texttt{p} \to \texttt{r}\{s_2\}.\texttt{q} \to \texttt{r}\{t_2\}.\texttt{r} \to \texttt{q}\{u_2\}.\texttt{rp}!\{r_2\}.\ X$$

Fig. 6 illustrates this protocol. Suppose a streaming service with two buffers (`A` and `B`), a source (left) and a sink (right). First **(a)**, buffer `A` is ready to copy an element (message r_1), and so it notifies the source. Then **(b)**, an element is copied into `A` (message s_1). Meanwhile, both the sink and `B` can notify the service and the source (respectively) that they are ready to copy (messages r_2 and t_1). This implies that, next **(c)**, both the service and the sink can copy an element *in parallel* (messages s_1 and u_1). Note that using a single buffer, this would not be possible, since we would risk overwriting the buffer before the sink copied it. In the next iteration **(d)**, we can swap the buffers, and repeat the process. By swapping the buffers, both the service (buffer `A`) and the sink can notify that they are ready, even if data is still being copied to buffer `B` (messages r_2 and t_2). Finally **(e)**, buffer `A` and the sink can copy the respective next elements, again in parallel (messages s_2 and u_2).

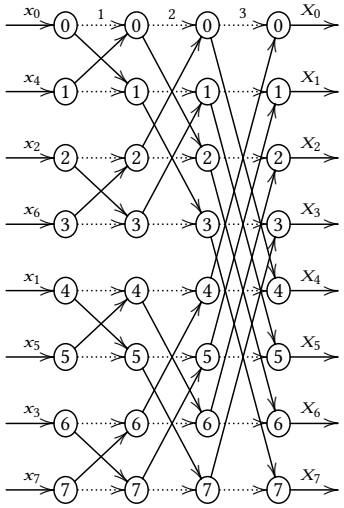
Remark 8.1 (Double-buffering). **(1)** Definition 6.1 does not directly check the asynchronous subtyping of local types from the above global type as our rules do not include unrolling recursive global types (to obtain the decidability result). However we can apply any (sound) asynchronous subtyping relation from the literature since the cost calculation does not related to well-formedness of global types. For example, local types that behave as the projections of this global type are known as deadlock-free [Mostrous et al. 2009; Yoshida et al. 2008]. **(2)** The syntax of the global type in [Mostrous et al. 2009] uses the explicit channels in global types. We translated them to corresponding labels, which does not affect the cost calculation, and our end-point implementation is essentially as identical as one in [Yoshida et al. 2008].

Castro-Perez and Yoshida [2020b]. ***Mergesort*** (`ms`) follows a divide-and-conquer protocol. ***Fast Fourier Transform*** (`fft`) is the Cooley-Tukey fast-fourier transform algorithm, implemented

(a) Butterfly pattern



(b) FFT diagram



(c) Global type

$$
\begin{aligned}
&\Pi n.\text{foreach}(i < 2^n)\{\\
&\ \text{foreach}(l < n)\{\\
&\ \ \text{foreach}(i < 2^l)\{\\
&\ \ \ \text{foreach}(j < 2^{n-l-1})\{\\
&\ \ \ \ \text{foreach}(k < 2)\{\\
&\ \ \ \ \ \text{foreach}(k' < 2)\{\\
&\ \ \ \ \ \ \mathsf{p}_{i\times 2^{n-l}+k\times 2^{n-l-1}+j}\\
&\ \ \ \ \ \ \ \rightarrow \mathsf{p}_{i\times 2^{n-l}+k'\times 2^{n-l-1}+j}\}\}\}\}\}
\end{aligned}
$$

(d) Programs

$\mathsf{P}_0(v)$ ::=	$\mathsf{P}_1(v)$ ::=
$x \leftarrow \texttt{fft}(v);$	$x \leftarrow \texttt{fft'}(v);$
send P_1 x;	send P_0 x;
$y \leftarrow$ recv P_1;	$y \leftarrow$ recv P_0;
$x \leftarrow \texttt{zip}_+(x, y);$	$x \leftarrow \texttt{zip}_-(x, y);$
send P_2 x;	send P_3 x;
$y \leftarrow$ recv P_2;	$y \leftarrow$ recv P_3;
$x \leftarrow \texttt{zip}_+(x, y);$	$x \leftarrow \texttt{zip}_+(x, y);$
send P_4 x;	send P_5 x;
$y \leftarrow$ recv P_4;	$y \leftarrow$ recv P_5;
$x \leftarrow \texttt{zip}_+(x, y);$	$x \leftarrow \texttt{zip}_+(x, y);$
return(x)	return(x);

Fig. 7. Butterfly Network Topology for Fast Fourier Transform

in C using pthreads, parallelised using a butterfly topology as illustrated in Fig. 7. It uses a divide-and-conquer strategy based on the following equation (we use $\omega_N^{2k} = \omega_{N/2}^k$):

$$X_k = \sum_{j=0}^{N-1} x_j\, \omega_N^{jk} = \sum_{j=0}^{N/2-1} x_{2j}\, \omega_{N/2}^{jk} + \omega_N^k \sum_{j=0}^{N/2-1} x_{2j+1}\, \omega_{N/2}^{jk}$$

Each of the two separate sums are DFT of half of the original vector members, separated into even and odd. Recursive calls can then divide the input set further based on the value of the next binary bits. Fig. 7(a) illustrates this recursive principle, called *butterfly*, where two different intermediary values can be computed in constant time from the results of the same two recursive calls. The complete algorithm for a size-8 is illustrated by the diagram from Fig.7(b). The global type in Fig.7(c) shows the resulting global type, in terms of the indices of the participants that need to communicate. We use keyword foreach to represent that the body must be expanded for all natural numbers that satisfy the condition (similarly to parameterised MPST [Deniélou et al. 2012]). CAMP uses a recursive definition that expands into a butterfly of the required size. Fig.7(d) shows the (abstract) code of our implementation for participants 0 and 1. We show the high-level structure, in terms of send and receive. Suppose that participants receive as initial value v, the deinterleaving of the input vector. Then, they all start applying a sequential fft, and communicate the result to the appropriate participants. Then, they apply the necessary addition and subtraction to compute their part of the result, and communicate it accordingly.

8.3 Discussion of Predicted Execution Times

Fig. 8 shows a comparison, for each benchmark, of the execution times compared with the predictions by our cost models. For most of our examples, we get predictions with $< 15\%$ of error.

Benchmark	*Protocol*	*Cost (s)*	Real (s)	Diff (%)
OCaml [Imai et al. 2020]				
pp-ev	PP	6.39e-6	6.29e-6	2.07
pp-lwt	PP	4.20e-7	4.07e-7	3.29
pp-ipc-0	PP	6.27e-6	5.95e-6	5.40
pp-ipc-1	PP	6.28e-6	6.12e-6	2.66
pp-ipc-2	PP	6.42e-6	6.19e-6	3.67
pp-ipc-3	PP	7.96e-6	7.80e-6	2.08
pp-ipc-4	PP	2.54e-5	2.09e-5	21.9
pp-ipc-5	PP	2.20e-4	2.19e-4	0.62
Go [Castro et al. 2019]				
aa-2	AA	2.42e-6	2.13e-6	14
aa-4	AA	4.85e-6	4.45e-6	8.8
1a-2	S	6.28e-6	4.46e-6	0.41
1a-3	S	8.12e-6	7.64e-6	6.42
1a-4	S	9.98e-6	9.85e-6	1.34
a1-2	G	2.8e-6	2.14e-6	30.66
a1-3	G	3.27e-6	2.86e-6	14.09
a1-4	G	3.74e-6	3.30e-6	13.22
sn-1	SG, CR	11.62	11.58	0.37
sn-2	SG, CR	5.87	5.81	1.05
sn-3	SG, CR	3.98	3.95	0.79
sn-4	SG, CR	3.06	3.05	0.08
kn-1	SG	10.88	10.65	2.16
kn-2	SG	11.93	11.13	7.15
kn-3	SG	14.01	13.01	7.69
kn-4	SG	17.28	17.17	0.66
dna-1	SG	3.00	2.93	2.38
dna-2	SG	3.34	3.39	1.38
dna-3	SG	3.68	3.66	0.48
dna-4	SG	4.02	4.01	0.24
Savina [Imam and Sarkar 2014]				
pp-akka	PP	4.4e-5	3.99e-5	10.28
ring	Ring	7.09-3	5.04e-3	40.67
ring-opt	Ring	5.24e-4	5.4e-4	2.8
count	CR	1.98e-4	1.53e-4	29.41
barb	CR	3.5e-4	3.36e-4	4.16
dphil	CR	2.03e-4	1.92e-4	5.75
csmok	CR	1.05e-4	1.03e-4	1.6

Benchmark	*Protocol*	*Cost (s)*	Real (s)	Diff (%)
C-MPI [Ng et al. 2015]				
nb-1	Ring	177.91	177.91	2e-6
nb-4	Ring	45.17	44.71	1.02
nb-16	Ring	12.07	11.10	8.79
nb-32	Ring	6.69	7.84	15
nb-64	Ring	4.29	4.28	0.086
ls-1	Mesh	10.98	10.58	3.78
ls-4	Mesh	4.34	4.44	2.23
ls-16	Mesh	1.88	1.72	9.67
ls-32	Mesh	1.19	1.30	8.79
ls-64	Mesh	0.87	0.72	0.20
wc-1	MR	57	57	1e-5
wc-2	MR	31.8	27.5	17
wc-8	MR	17	16	6.26
wc-24	MR	17.5	19.5	10
wc-64	MR	20.6	23.0	10
ap-1	MR	657	656	7e-2
ap-2	MR	330	284	16
ap-8	MR	67	65	3.4
ap-24	MR	51	45	13
ap-64	MR	74	64	17
C-pthreads [Castro-Perez and Yoshida 2020b]				
fft	Btfly	143.1	143.0	5.8e-2
fft-2	Btfly	74.3	74.1	1.7e-1
fft-4	Btfly	40.5	40.8	7.2e-1
fft-8	Btfly	24.3	21.8	12
fft-32	Btfly	13.6	12.4	9.3
ms-2	d&c	53.6	53.2	7.3-1
ms-4	d&c	31.39	31.33	1.3-1
ms-8	d&c	20.1	18.1	11.3
ms-16	d&c	14.6	14.2	2.5
OCaml [Zhou et al. 2020]				
twobuy	CR	4.0133	4.0035	0.24
C [Yoshida et al. 2008]				
dbuff	Double Buffer	2.54e-1	2.12e-1	19.7

Fig. 8. Predicted vs real execution times: PP = Ping-Pong, AA = All-to-All, S = Scatter, G = Gather, SG = Scatter-Gather, CR = choice with recursion, MR = MapReduce, D&C = parallel divide and conquer.

Examples include **pp-ipc-4**, **a1-2**, **ring**, **count**, **nb-32**, **wc-2**, **ap-2**, **ap-64**, and **dbuff**. We observe that the worst predictions are those of the microbenchmarks, with very small execution times. Here, communication costs dominate, and are repeated a large number of times. With such small costs, a small error is amplified after a large enough number of iterations. An example of this is **ring**, that is a recursive ring protocol that is run for 10^5 iterations.

When we consider examples with larger local computation costs, most of the predictions are with less than 10% error. There are a small number of examples above than 10% where errors in the prediction are due to factors that CAMP's cost models do not take into account, such as scheduler costs, cost of thread creation, or resource contention such as shared caches. These details that the cost models do not take into account can also explain why, in some cases, the cost models do not predict an upper bound of the cost, since the real executions include slowdowns due to these factors. Note, however, that CAMP offers a quick and static first assessment of the performance behaviour of concurrent and distributed systems which use different transports and topologies, without the need to deploy or profile the application.

Asynchronous Communication Optimisations. Algorithms `fft`, `dbuff`, `nb` and `ring` all rely on asynchronous communication optimisations. Both `fft` and `dbuff` require to be specified using this extension. For `ring`, we take measurements to compare the optimised and unoptimised global types. We can observe a speedup in the execution of the protocol that is predicted by the cost models, which is consistent with Theorem 6.3.

9 RELATED WORK

Resource Analysis and Session Types. Das et al. [2018b] combine session types with amortised resource analysis in a linear type system, to reason about resource usage of message-passing processes, but their work focuses on binary sessions in a linear type system, while we focus on multiparty session types, and the global execution times of the protocol. Das et al. [2018a] extend a system of binary session types in a Curry-Howard correspondence with intuitionistic linear logic [Caires and Pfenning 2010; Caires et al. 2016] with temporal modalities *next*, *always*, and *eventually*, to prescribe the timing of the communication. A fundamental difference with our work is that Das et al. [2018a] require the introduction of delays into the processes, to match the specified cost. In our case, the processes are left unmodified, and the cost is computed from the protocol descriptions. Finally, their work are limited to theory, while our work are readily applied to real use cases.

Asynchronous Communication Optimisation. The first idea of asynchronous communication optimisation was found in Scribble [scribble authors 2008] where a multiparty financial protocol with message ordering permutations is informally described. Later this idea was formalised as *asynchronous session subtyping* for the π-calculus [Chen et al. 2017, 2014; Mostrous and Yoshida 2009, 2015; Mostrous et al. 2009] and its denotational properties were studied in [Demangeon and Yoshida 2015; Dezani-Ciancaglini et al. 2016]. Concurrently, because of the need of asynchronous optimisation in multiparty protocols, several applications inspired by asynchronous subtyping have been developed in Java [Hu 2017], C [Yoshida et al. 2008] and MPI-C [Ng et al. 2015, 2012], but without any formal theories. Recently, this subtyping relation was found undecidable for *binary* session types [Bravetti et al. 2017, 2018; Lange and Yoshida 2017] and its sound algorithm for binary session communicating automata was proposed in [Bravetti et al. 2019]. We have implemented a different and more practical decidable optimisation relation based on asynchronous subtyping for multiparty session types, recently proposed in [Ghilezan et al. 2021]. None of the above work has (1) developed a formal cost theory which can justify the optimisation; and (2) measured and compared the cost of optimised/unoptimised applications with a formal justification. CAMP is the first framework which (1) proposes a formal cost theory with asynchronous optimisation (Theorems 6.2 and 6.3) and (2) justifies the optimisation cost against real benchmarks using (1).

Timed Session Types. The notion of *time* has been introduced to session types [Bartoletti et al. 2017; Bocchi et al. 2015, 2019, 2014], to account for protocols that require time specifications, originated from communicating timed automata (CTA) [Krčál and yi 2006]. Session types and the π-calculus processes have been related in terms of static typing [Bocchi et al. 2019, 2014], or timed session types are linked with compliments relations [Bartoletti et al. 2017] or CTA [Bocchi et al. 2015]. Among them, [Bartoletti et al. 2017; Bocchi et al. 2019] are limited to binary or server-client session types. All of the above works are theoretical only, while the work in [Bocchi et al. 2014] was applied to the runtime monitoring in Python [Neykova et al. 2017]. The main difference is that the above timed session types focus on ensuring that deadlines or time constraints are satisfied. In contrast, our work does not enforce any time constraints, since we are interested on the static estimation of execution costs, but not on enforcing that timeouts and deadlines are respected.

Type-Based and Amortised Cost Analysis. Handley et al. [2019] use refinement types to reason about efficiency, cost, of Haskell programs, but they do not consider concurrency or parallelism. Sized types [Hughes et al. 1996] are one of the successful techniques for cost analysis of programs

[Avanzini and Dal Lago 2017; Portillo et al. 2002; Vasconcelos 2008]. Most of the uses of sized types do not deal with concurrency and distribution. Exceptions are [Gimenez and Moser 2016], that address space and space-time complexity of parallel reductions of interaction-net programs using sized and scheduled types, but they do not address message-passing and distributed environments. The work [Hoffmann and Shao 2015] extends earlier amortised cost analyses [Hoffmann et al. 2012] to parallel reductions. Their work focuses on parallel functional programs with explicit parallel composition, but does not address message-passing. To our best knowledge, none of the work above addresses the cost of message-passing constructs or distributed environments.

10 CONCLUSIONS AND FUTURE WORK

We have presented CAMP, a framework for statically predicting the *cost*, execution times, of concurrent and distributed systems. CAMP augments global types from the theory of multiparty session types with *local computation costs*, and its trace semantics is extended with local computation observable actions. We have developed a way to extract cost equations from these instrumented protocol descriptions, that we can use for estimating upper-bounds of the execution times required by the participants of a protocol. CAMP can be used to predict the *latency*, i.e. the execution times that the participants of a protocol will require, on average, per iteration of the protocol. Furthermore, we extended CAMP to address *asynchronous communication optimisation*. CAMP's cost analysis on top of multiparty session types gives us several benefits. Firstly, we can use global types to reason about both *correctness* and *performance* of concurrent and distributed systems. Secondly, the cost analysis can be readily applied and integrated into any MPST framework. Thirdly, it can be used in non-session-based concurrency benchmarks by simply providing MPST protocols. It suffices to describe the global type instrumented with cost, and instantiate the derived cost equations with measured or estimated communication latencies, and local computation costs. And, fourthly our prototype accounts for CPU/CORE availability of the target hardware.

CAMP addresses two main concerns when estimating execution costs of concurrent and distributed systems: communication overheads, and synchronisation. Although these factors are a main source of inefficiency, there are more that we still do not take into account, such as the cost of starting new threads, the cost of context switching/scheduling, or the cost of resource contention such as shared caches [Lea 1997]. We plan to study how to extend CAMP to take such factors into account as future work. CAMP considers distributed systems comprised of multiple nodes, each of which with a number of CPUs/cores. We plan to extend CAMP's hardware descriptions to consider heterogeneity, e.g. CPU clusters, FPGAs, etc. CAMP's cost models take the maximum cost of the different possible branches in a protocol. This is sufficient to compute a worst-case execution time of non-recursive protocols. However, we can extend our costs to take into account the *weight* of different branches, so that our cost models would compute an average cost based on the probability to take the different branches. Moreover, since communication latencies may not be very predictable, we plan to study the extension of CAMP to use probabilistic cost estimations. Finally we plan to study the development of a performance analysis tool for existing code, based on the *inference* or *extraction* of the communication protocol followed by non-session-typed implementations such as [Gabet and Yoshida 2020; Lange et al. 2018; Ng and Yoshida 2016].

ACKNOWLEDGMENTS

We thank the OOPSLA reviewers for their careful reviews and suggestions; and Lorenzo Gheri and Fangyi Zhou for their comments. Francisco Ferreira and Keigo Imai helped testing our artifact submission. The work is supported by EPSRC EP/T006544/1, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, EP/T006544/1, EP/T014709/1 and EP/V000462/1, and NCSS/EPSRC VeTSS.

REFERENCES

Krste Asanovic, Rastislav Bodík, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. 2009. A view of the parallel computing landscape. *Commun. ACM* 52, 10 (2009), 56–67. https://doi.org/10.1145/1562764.1562783

Martin Avanzini and Ugo Dal Lago. 2017. Automating sized-type inference for complexity analysis. *PACMPL* 1, ICFP (2017), 43:1–43:29. https://doi.org/10.1145/3110287

Massimo Bartoletti, Tiziana Cimoli, and Maurizio Murgia. 2017. Timed Session Types. *Logical Methods in Computer Science* 13, 4 (2017). https://doi.org/10.23638/LMCS-13(4:25)2017

Laura Bocchi, Julien Lange, and Nobuko Yoshida. 2015. Meeting Deadlines Together. In *26th International Conference on Concurrency Theory (LIPIcs)*, Vol. 42. Schloss Dagstuhl, 283–296.

Laura Bocchi, Maurizio Murgia, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2019. Asynchronous Timed Session Types - From Duality to Time-Sensitive Processes. In *28th European Symposium on Programming, ESOP 2019 (LNCS)*, Luís Caires (Ed.), Vol. 11423. Springer, 583–610. https://doi.org/10.1007/978-3-030-17184-1_21

Laura Bocchi, Weizhen Yang, and Nobuko Yoshida. 2014. Timed Multiparty Session Types. In *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings (LNCS)*, Paolo Baldan and Daniele Gorla (Eds.), Vol. 8704. Springer, 419–434. https://doi.org/10.1007/978-3-662-44584-6_29

Mario Bravetti, Marco Carbone, Julien Lange, Nobuko Yoshida, and Gianluigi Zavattaro. 2019. A Sound Algorithm for Asynchronous Session Subtyping. In *30th International Conference on Concurrency Theory (LIPIcs)*, Vol. 140. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2017. Undecidability of asynchronous session subtyping. *Inf. Comput.* 256 (2017), 300–320.

Mario Bravetti, Marco Carbone, and Gianluigi Zavattaro. 2018. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.* 722 (2018), 19–51. https://doi.org/10.1016/j.tcs.2018.02.010

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings (LNCS)*, Paul Gastin and François Laroussinie (Eds.), Vol. 6269. Springer, 222–236. https://doi.org/10.1007/978-3-642-15375-4_16

Luís Caires, Frank Pfenning, and Bernardo Toninho. 2016. Linear logic propositions as session types. *Mathematical Structures in Computer Science* 26, 3 (2016), 367–423. https://doi.org/10.1017/S0960129514000218

David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. 2019. Distributed Programming using Role-Parametric Session Types in Go *(POPL'19)*. ACM, New York, NY, USA, 12.

David Castro-Perez and Nobuko Yoshida. 2020a. CAMP: Cost-Aware Multiparty Session Protocols. arXiv:2010.04449 [cs.PL].

David Castro-Perez and Nobuko Yoshida. 2020b. Compiling First-Order Functions to Session-Typed Parallel Code. In *Proc. of the 29th Int. Conf. on Compiler Construction (CC2020) (CC 2020)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/3377555.3377889

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. 2017. On the Preciseness of Subtyping in Session Types. *LMCS* 13 (2017), 1–62. Issue 2.

Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2014. On the Preciseness of Subtyping in Session Types. In *PPDP*. ACM Press, 135–146.

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2015. A Gentle Introduction to Multiparty Asynchronous Session Types. In *15th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Multicore Programming (LNCS)*, Vol. 9104. Springer, 146–178.

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018a. Parallel complexity analysis with temporal session types. *PACMPL* 2, ICFP (2018), 91:1–91:30. https://doi.org/10.1145/3236786

Ankush Das, Jan Hoffmann, and Frank Pfenning. 2018b. Work Analysis with Resource-Aware Session Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 305–314. https://doi.org/10.1145/3209108.3209146

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 – Concurrency Theory*, Maciej Koutny and Irek Ulidowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 272–286.

Romain Demangeon and Nobuko Yoshida. 2015. On the Expressiveness of Multiparty Sessions. In *FSTTCS 2015 (LIPIcs)*, Prahladh Harsha and G. Ramalingam (Eds.), Vol. 45. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 560–574. https://doi.org/10.4230/LIPIcs.FSTTCS.2015.560

Pierre-Malo Deniélou and Nobuko Yoshida. 2013. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II (LNCS)*, Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg (Eds.), Vol. 7966. Springer, 174–186. https://doi.org/10.1007/978-3-642-39212-2_18

Pierre-Malo Deniélou, Nobuko Yoshida, Andi Bejleri, and Raymond Hu. 2012. Parameterised Multiparty Session Types. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:6)2012

Mariangiola Dezani-Ciancaglini, Silvia Ghilezan, Svetlana Jaksic, Jovanka Pantovic, and Nobuko Yoshida. 2016. Denotational and Operational Preciseness of Subtyping: A Roadmap. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*. 155–172. https://doi.org/10.1007/978-3-319-30734-3_12

Julia Gabet and Nobuko Yoshida. 2020. Static Race Detection and Mutex Safety and Liveness for Go Programs *(LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. To appear in ECOOP'20.

Simon Gay and Antonio Ravara (Eds.). 2017. *Behavioural Types: from Theory to Tools*. River Publishers.

Silvia Ghilezan, Jovanka Pantovic, Ivan Prokic, Alceste Scalas, and Nobuko Yoshida. 2021. Precise Subtyping for Asynchronous Multiparty Sessions. *Proc. ACM Program. Lang.* POPL (2021). To appear in POPL'21.

Stéphane Gimenez and Georg Moser. 2016. The complexity of interaction. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 243–255. https://doi.org/10.1145/2837614.2837646

Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.

Martin A. T. Handley, Niki Vazou, and Graham Hutton. 2019. Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. *Proc. ACM Program. Lang.* 4, POPL, Article Article 24 (Dec. 2019), 27 pages. https://doi.org/10.1145/3371092

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.* 34, 3 (2012), 14:1–14:62. https://doi.org/10.1145/2362389.2362393

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *24th European Symposium on Programming, ESOP 2015 (LNCS)*, Jan Vitek (Ed.), Vol. 9032. Springer, 132–157. https://doi.org/10.1007/978-3-662-46669-8_6

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proc. of 35th Symp. on Princ. of Prog. Lang. (POPL '08)*. ACM, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

Raymond Hu. 2017. Distributed Programming Using Java APIs Generated from Session Types. *Behavioural Types: from Theory to Tools* (2017), 287–308.

Raymond Hu and Nobuko Yoshida. 2017. Explicit Connection Actions in Multiparty Session Types. In *20th Int. Conf. on Fundamental Approaches to Software Engineering, FASE 2017 (LNCS)*, Marieke Huisman and Julia Rubin (Eds.), Vol. 10202. Springer, 116–133. https://doi.org/10.1007/978-3-662-54494-5_7

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 410–423. https://doi.org/10.1145/237721.240882

Keigo Imai, Rumyana Neykova, Nobuko Yoshida, and Shoji Yuen. 2020. Multiparty Session Programming with Global Protocol Combinators. https://github.com/keigoi/ocaml-mpst *(LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. To appear in ECOOP'20.

Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proc. of the 4th Int. Workshop on Programming Based on Actors Agents & Decentralized Control (AGERE! '14)*. Association for Computing Machinery, New York, NY, USA, 67–80.

Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 77–88. https://doi.org/10.1145/2254064.2254075

K. Krommydas et al. 2016. OpenDwarfs: Characterization of Dwarf-Based Benchmarks on Fixed and Reconfigurable Architectures. *J Sign Process Syst* 85 (2016), 373—-392.

Pavel Krčál and Wang yi. 2006. Communicating Timed Automata: The More Synchronous, the More Difficult to Verify. 249–262. https://doi.org/10.1007/11817963_24

Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. 2018. A Static Verification Framework for Message Passing in Go using Behavioural Types. In *40th International Conference on Software Engineering*. ACM, 1137–1148.

Julien Lange and Nobuko Yoshida. 2017. On the Undecidability of Asynchronous Session Subtyping. In *20th International Conference on Foundations of Software Science and Computation Structures (LNCS)*, Vol. 10203. Springer, 441–457.

Doug Lea. 1997. *Concurrent programming in Java - design principles and patterns*. Addison-Wesley-Longman.

Dimitris Mostrous and Nobuko Yoshida. 2009. Session-Based Communication Optimisation for Higher-Order Mobile Processes. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings (Lecture Notes in Computer Science)*, Pierre-Louis Curien (Ed.), Vol. 5608. Springer, 203–218. https://doi.org/10.1007/978-3-642-02273-9_16

Dimitris Mostrous and Nobuko Yoshida. 2015. Session Typing and Asynchronous Subtying for Higher-Order π-Calculus. *Info.& Comp.* 241 (2015), 227–263.

Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. 2009. Global Principal Typing in Partially Commutative Asynchronous Sessions. In *ESOP (LNCS)*, Vol. 5502. Springer, 316–332.

Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. 2017. Timed Runtime Monitoring for Multiparty Conversations. *FAOC* (2017), 1–34.

Nicholas Ng, José Gabriel de Figueiredo Coutinho, and Nobuko Yoshida. 2015. Protocols by Default - Safe MPI Code Generation Based on Session Types. In *24th Int. Conf. on Compiler Construction, CC 2015 (LNCS)*, Björn Franke (Ed.), Vol. 9031. Springer, 212–232. https://doi.org/10.1007/978-3-662-46663-6_11

Nicholas Ng and Nobuko Yoshida. 2016. Static deadlock detection for concurrent Go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, Ayal Zaks and Manuel V. Hermenegildo (Eds.). ACM, 174–184. https://doi.org/10.1145/2892208.2892232

Nicholas Ng, Nobuko Yoshida, and Kohei Honda. 2012. Multiparty Session C: Safe Parallel Programming with Message Optimisation. In *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings (Lecture Notes in Computer Science)*, Carlo A. Furia and Sebastian Nanz (Eds.), Vol. 7304. Springer, 202–218. https://doi.org/10.1007/978-3-642-30561-0_15

Benjamin C Pierce. 2002. *Types and programming languages*. The MIT Press.

Álvaro J. Rebón Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro B. Vasconcelos. 2002. Cost Analysis Using Automatic Size and Time Inference. In *Implementation of Functional Languages, 14th International Workshop, IFL 2002, Madrid, Spain, September 16-18, 2002, Revised Selected Papers (LNCS)*, Ricardo Pena and Thomas Arts (Eds.), Vol. 2670. Springer, 232–248. https://doi.org/10.1007/3-540-44854-3_15

Thomas Rauber and Gudula Rünger. 2010. *Parallel Programming - for Multicore and Cluster Systems*. Springer. https://doi.org/10.1007/978-3-642-04818-0

The scribble authors. 2008. Scribble homepage. https://www.scribble.com.

Gadi Taubenfeld. 2006. *Synchronization algorithms and concurrent programming*. Pearson Education.

USENIX. 2020. Double-Buffering Algorithm (web). https://www.usenix.org/legacy/publications/library/proceedings/usenix02/full_papers/huang/huang_html/node8.html.

Pedro B. Vasconcelos. 2008. *Space cost analysis using sized types*. Ph.D. Dissertation. University of St Andrews, UK. http://hdl.handle.net/10023/564

Nobuko Yoshida, Vasco Thudichum Vasconcelos, Hervé Paulino, and Kohei Honda. 2008. Session-Based Compilation Framework for Multicore Programming. In *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures (Lecture Notes in Computer Science)*, Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine (Eds.), Vol. 5751. Springer, 226–246. https://doi.org/10.1007/978-3-642-04167-9_12

Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. 2020. Statically Verified Refinements for Multiparty Protocols. (2020). Conditionally Accepted by OOPSLA '20, Preprint on https://www.doc.ic.ac.uk/~fz315/oopsla20-preprint.pdf.