UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO DE CIÊNCIAS, TECNOLOGIA E SAÚDE - CAMPUS ARARANGUÁ
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Felipe Bonato

Concorrência em Sistemas Embarcados: Uma Revisão de Escopo

Araranguá
2023

Felipe Bonato

Concorrência em Sistemas Embarcados: Uma Revisão de Escopo

Trabalho de Conclusão de Curso do Curso de Graduação em Engenharia de Computação do Centro de Ciências, Tecnologia e Saúde - Campus Araranguá da Universidade Federal de Santa Catarina para a obtenção do título de Bacharel em Engenharia de Computação.
Orientador: Prof. Alison R. Panisson, Dr.

Araranguá
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Bonato, Felipe
Concorrência em Sistemas Embarcados: Uma Revisão de Escopo / Felipe Bonato ; orientador, Alison R. Panisson, 2023.
22 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Campus Araranguá, Graduação em Engenharia de Computação, Araranguá, 2023.

Inclui referências.

1. Engenharia de Computação. 2. Sistemas Embarcados. 3. Concorrência. 4. Linguagens de Programação. 5. Revisão de Escopo. I. Panisson, Alison R. . II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Computação. III. Título.

Felipe Bonato

Concorrência em Sistemas Embarcados: Uma Revisão de Escopo

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Engenharia de Computação" e aprovado em sua forma final pelo Curso de Graduação em Engenharia de Computação.

Araranguá, 21 de Junho de 2023.

Documento assinado digitalmente
Jim Lau
Data: 27/06/2023 15:48:36-0300
CPF: ***.464.702-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Jim Lau, Dr.
Coordenador do Curso

Banca Examinadora:

Documento assinado digitalmente
Alison Roberto Panisson
Data: 27/06/2023 15:43:37-0300
CPF: ***.887.420-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Alison R. Panisson, Dr.
Orientador

Documento assinado digitalmente
Analucia Schiaffino Morales
Data: 28/06/2023 07:55:26-0300
CPF: ***.256.420-**
Verifique as assinaturas em https://v.ufsc.br

Profa. Analúcia Schiaffino Morales, Dra., Dr.
Avaliadora
Instituição UFSC

Documento assinado digitalmente
Jim Lau
Data: 27/06/2023 15:48:53-0300
CPF: ***.464.702-**
Verifique as assinaturas em https://v.ufsc.br

Prof. Jim Lau, Dr.
Avaliador
Instituição UFSC

Concorrência em Sistemas Embarcados: Uma Revisão de Escopo

Felipe Bonato* Alison R. Panisson†

2023, July

Resumo

Diversos sistemas computacionais possuem a necessidade de um baixo consumo de energia associado ao bom desempenho, como por exemplo, sistemas embarcados e IoT. Uma direção adotada pela indústria para aumentar o desempenho de sistemas ao longo dos anos foi a abordagem de processadores *multi-core*, como capacidade de processamento paralelo diretamente no *hardware*. Porém, essa abordagem, muitas vezes, vai em direção oposta a necessidade de alguns sistemas, já que normalmente essa abordagem de hardware tem um maior custo e consome mais energia que uma abordagem *single-core*. Nesse sentido, é importante entender como a literatura tem lidado com esses sistemas de computação. Dessa forma, esse artigo tem como objetivo entender como concorrência que não utiliza de hardware paralelo funciona em sistemas embarcados. Durante a investigação realizada, foram analisadas corrotinas, green threads e assincronismo. Com base nisso, uma metodologia inspirada no método prisma foi adotada, e questões de pesquisa foram formuladas com o objetivo de sintetizar características de linguagens de programação e bibliotecas utilizadas para esse fim, e procurar tendências na área. Além disso, esse estudo contribui com uma discussão sobre qual e porque determinados tipos de concorrência são utilizados em sistemas embarcados.

Palavras-chaves: Sistemas Embarcados. Concorrência. Linguagens de Programação.

*fel.bonato@gmail.com
†alison.panisson@ufsc.br

Cuncurrency in Embedded Systems: A Scoping Review

Felipe Bonato* Alison R. Panisson†

2023, July

Abstract

Several computer systems require low energy consumption associated with good performance, such as embedded systems and IoT. One approach adopted by the industry to improve computers performance over the years has been the use of multi-core processors, which provide parallel processing capabilities directly in the hardware. However, this approach often is in conflict with the needs of certain systems, as it typically has a higher cost and consumes more energy than a single-core approach. Therefore, it is important to understand how the literature has dealt with these computing systems. In this regard, the objective of this article is to explore how non-parallel concurrency functions in embedded systems. During the conducted investigation, coroutines, green threads, and asynchrony were analyzed. Based on this analysis, inspired by the PRIMA protocol, research questions were formulated to synthesize programming language features and libraries used for this purpose and identify trends in the field. Additionally, this article contributes to the discussion of which types and why concurrency is used in embedded systems.

Key-words: Embedded Systems. Concurrency, Programming Languages.

*fel.bonato@gmail.com
†alison.panisson@ufsc.br

1 Introdução

Em 1965, Gordon Moore realizou a observação que o número de transistores em circuitos integrados dobrava a cada 2 anos (MOORE, 1965). Essa observação ficou conhecida como *"Lei de Moore"*. De forma geral, esse aumento levava a constante melhora na capacidade de processamento de microprocessadores. Porém, na metade dos anos 2000, o crescimento exponencial do número de transistores encontrou uma barreira: os limites físicos de quão pequenos semicondutores podiam ser feitos (SUTTER, 2005), levando a dúvidas acerca da continuidade da Lei de Moore. Com isso, a indústria buscou alternativas para continuar os avanços em relação ao desempenho de computadores. A principal estratégia adotada foi o processamento paralelo, ou seja, ao invés de aumentar a capacidade de processamento reduzindo os transistores, incrementou-se o número de núcleos de processamento (KNIGHT, 2005), iniciando a era de microprocessadores *multi-core*. Essa abordagem permitiu a continuação da Lei de Moore, mesmo após a barreira encontrada quanto ao progresso da diminuição do tamanho dos transistores.

No entanto, ao contrário de microprocessadores com foco em PCs (*Personal Computer*), frequentemente, microcontroladores ainda são majoritariamente *single-core*. Ainda que o mercado de sistemas embarcados tenha atingido um valor de USD 86,5 bilhões, de acordo com pesquisas de mercado de 2020, publicadas em (MARKETS AND MARKETS, 2020), existe a necessidade de melhorar a performance dos dispositivos utilizados nesses tipos de sistemas. Porém, devido a diferenças significativas nos paradigmas de sistemas embarcados, que por natureza são altamente concorrentes, orientado a eventos e de tempo real (BARR; MASSA, 2006), é vantajoso considerar melhorias de performance alternativas, que sejam adaptadas aos requisitos específicos desses sistemas, ao invés daqueles adotados para PCs que adicionam múltiplos núcleos ao hardware (CHRISTIAN, 2014). Esse necessidade se deve ao custo, complexidade e consumo de energia que processadores *multi-core* possuem.

Diante disso, torna-se necessário que linguagens de programação, e sistemas operacionais, incorporem ou permitam a criação de construções que facilitem programação concorrente em dispositivos embarcados. Esse trabalho busca realizar uma investigação sistemática para responder como as linguagens de programação usam de alternativas para permitir concorrência. Dentre as alternativas para lidar com este desafio, algumas ganham destaque: ***threads de kernel***, ***green threads*** (também chamadas de *threads* leves, *threads* de nível de usuário ou *threads* virtuais), **corrotinas** e **assincronismo**. Cada uma delas possui objetivos específicos, e apresentam vantagens e desvantagens distintas, conforme será discutido na Seção 2. Porém, levando em consideração as características e importância das quatro opções, este trabalho fará a análise apenas de corrotinas, *green threads* e assincronismo, tendo em vista que threads de kernel já são utilizadas a anos em PCs, possuem padrões de uso e arquitetura bem estabelecidos e não são frequentemente a melhor opção para sistemas embarcados (MARWEDEL, 2021).

Essa trabalho está organizado da seguinte forma: Seção 2 apresenta uma descrição sucinta das alternativas de concorrência e paralelismo, justificando o foco nas que serão estudadas. Seção 3 apresenta o método, as bases de dados, questões de pesquisa, string de busca, critérios de inclusão e exclusão, bem como uma descrição do processo de seleção utilizada nessa revisão de escopo. Seção 4 apresenta os resultados obtidos em relação a cada questão de pesquisa, seguido de uma discussão mais ampla apresentada na Seção 5. Por fim, na Seção 6 são apresentadas nossas conclusões e trabalhos relacionados, respectivamente.

2 Alternativas de Concorrência

Dentre as alternativas para lidar com o desafio de melhorar aspectos de concorrência em sistemas, as que mais se destacam são **threads de kernel**, **corrotinas**, **assincronismo** e **green threads**. As quais serão descritas de forma sucinta nessa seção.

A opção mais clássica é a **thread de kernel**. Amplamente utilizada em sistemas operacionais de PCs (SUTTER, 2005). Porém, ela apresenta algumas características frequentemente indesejadas em sistemas embarcados: sua inicialização e trocas de contexto[1] possuem alto custo computacional, e a obrigatoriedade da preempção[2] sem controle do usuário. Os custos associados a inicialização e troca de contexto frequentemente são positivos em sistema *multi-core*, pois threads nesses sistemas podem ser despachadas para núcleos de processamento separados no hardware, permitindo execução paralela poderosa (KNIGHT, 2005). Porém, em sistemas embarcados *single-core*, o custo não vem acompanhado com as vantagens, pois não há múltiplos núcleos para serem despachados. Além disso, a preempção pode ser indesejada para códigos críticos, onde a pausa da execução num momento indesejado pode trazer problemas (SHORT; PONT; FANG, 2008).

Threads de Hardware oferecidas com threads de kernel já são utilizadas nos PCs a décadas, e portanto já possuem padrões de desenvolvimento bem estabelecidos. Além de não serem frequentes em sistemas embarcados. Portanto, *threads de hardware* não serão analisadas nesse artigo. Uma alternativa, similar as threads de kernel, são **fibers**, que existentes em diversos sistemas operacionais, não possuem preempção, e oferecem uma solução ao problema de códigos críticos. Porém, continuam com o custo das trocas de contexto e inicialização, e requisitarem um sistema operacional robusto para sua implementação. Portanto, da mesma forma, menos adequadas a sistemas embarcados.

Isso leva à opção que possuem implementações a nível de usuário, e podem ser consideradas mais adequadas para sistemas embarcados. São elas: **corrotinas**, **assincronismo** e **green threads**.

- **Corrotinas** oferecem "funções que podem ser pausadas" (CONWAY, 1963). Elas oferecem natureza cooperativas, permitindo que o usuário "pause" uma função, concedendo controle a quem a invocou. Sua implementação geralmente salva todo o contexto de função numa pilha, comumente nomeada *Coroutine Stack Frame* (MOURA; IERUSALIMSCHY, 2009), possuindo seu controle por meio de uma máquina virtual[3] ou, mais frequentemente, biblioteca *run-time*[4]. Apesar de não serem utilizadas em controle de fluxo geral, um uso já comum para corrotinas são *generators*, geralmente usados para implementar *iteradores*, usados em laços de repetição.

- **Assincronismo**, usando async/await, é frequentemente implementado como uma especialização de corrotinas, com o foco em IO (*input/output*) (ELIZAROV et al., 2021). A implementação faz o uso de um *event-loop*, onde tarefas em execução são

[1] Troca de contexto é o ato em que o estado de execução de uma tarefa é salvo em memória, e outro é colocado em execução.

[2] Preempção ocorre quando um agente externo interrompe temporariamente uma tarefa em execução, com o objetivo de que outra tarefa possa ser executada (gerando uma troca de contexto), para que posteriormente a tarefa original volte a ser executada (SILBERSCHATZ; GALVIN; GAGNE, 2021).

[3] Máquina Virtual é um motor de execução, que é responsável pela de programa (ARNOLD et al., 2005).

[4] Uma biblioteca *run-time* é uma biblioteca que é chamada pelo código do usuário e oferece funcionalidade a execução do programa

armazenadas até que seu resultado seja obtido, permitindo a continuação da execução da tarefa. Assim, não há espera até que a tarefa de IO seja completada.

- ***Green threads*** são implementações bastante semelhantes a threads do kernel, porém, em nível de usuário. O controle da preempção é feito normalmente por uma máquina virtual (por exemplo, no caso de Java), ou por uma biblioteca de *run-time* (por exemplo, no caso da biblioteca Tokio, da linguagem Rust) (TOKIO, 2023). Como elas não precisam passar pelo sistema operacional, possuem um custo de inicialização e de troca de contexto muito menor, o que pode levar elas a terem melhor performance (SEO et al., 1999)

Assim, um estudo da área deve ser realizado com o objetivo de reunir os conhecimentos acerca de como as linguagens implementam as técnicas apresentadas acima. Pesquisas realizadas pelos autores, demonstram que o único trabalho correlacionado apresentando uma revisão que abrange a mesma área desse estudo, foi publicado em (BELSON et al., 2019). Porém, esse estudo apresenta uma revisão com foco apenas na análise de técnicas de assincronismo, e não incluindo corrotinas e *green threads*. Esse trabalho demonstrou que existiam diversos padrões, e que a implementação a nível de linguagem poderia trazer vantagens para o melhor uso e manutenibilidade.

3 Método

Esse trabalho segue uma metodologia adaptada do método prisma (PRISMA, 2020) para revisão sistemática. Com o objetivo de coletar e sintetizar conhecimentos acerca de métodos concorrentes em sistemas embarcados, como foco em implementações a nível de usuário.

3.1 Questões de Pesquisa

O objetivo principal dessa revisão de escopo é entender como processos concorrentes que não utilizam de hardware paralelo são implementados em sistemas embarcados. Para atingir esse objetivo, foram formuladas questões de pesquisa que visam sintetizar características de linguagens de programação e bibliotecas, e procurar tendências em áreas, ou seja, implementação de concorrência em sistemas embarcados. Além de sintetizar quais tipos de concorrência foram utilizados ao longo do desenvolvimento das linguagens de programação, bem como entender o porque aqueles tipos são utilizados, com foco em sistemas embarcados.

Com base nisso, as seguintes questões de pesquisa foram formuladas:

RQ1. Qual o paradigma da linguagem de programação utilizada?

RQ2. Qual linguagem de programação e/ou biblioteca foi utilizada?

RQ3. Quando ocorreu a inclusão das técnicas de concorrência identificadas?

RQ4. Qual tipo de concorrência foi utilizada?

RQ5. Quais foram os motivos de incorporar concorrência na linguagem e/ou biblioteca?

RQ6. Quais as limitações encontradas?

3.2 Fontes de Informação

A escolha das fontes de informações utilizadas se deu devido a sua relevância no tópico, seu amplo uso acadêmico, seu grande número de artigos indexados (ROGERS, 2008) e por possuir acesso gratuito. Porém, algumas linguagens apresentam informações das suas implementações na literatura cinzenta, principalmente nas páginas web das linguagens, em *blogs* e em apresentações informais realizadas em eventos diversos, e portanto, difíceis de serem incluídas em sua totalidade. Também, alguns artigos são do tipo *technical report*, onde apenas são descritas implementações de projetos, que citam a técnica de interesse. Ambos casos foram excluídos neste estudo, pois foram consideradas difíceis de serem incluídas de forma abrangente, correta e sistemática. Sendo assim, as seguintes bases de dados foram utilizadas:

- **Scopus**.
- **Web of Science**.
- **IEEE Xplore**.

Nessas fontes de informação a mesma string de busca foi utilizada para fazer a busca em todas as bases. A busca foi realizada em 15 de janeiro de 2023, portanto esse estudo abrange artigos publicados e disponíveis nas bases de conhecimento até essa data.

3.3 Busca nas Bases

Com o objetivo de responder as questões de pesquisa, a seguinte *string* de busca foi modelada e utilizada:

```
   (
    "virtual thread"  OR "virtual threading" OR "thread virtualization" OR
    "green thread" OR "cooperative multitasking" OR coroutine OR
    "asynchronous algorithm" OR "asynchronous programming" OR
    "asynchronous network" OR "asynchronous communication" OR async
   )
AND
   (
    "embedded system" OR "Internet of Things" OR IoT OR
    "ubiquitous system" OR "ubiquitous computing" OR "real-time computing"
   )
```

Para busca de *threads virtuais*, sinônimos foram usados, como *virtual thread*, *thread virtualization*, *virtual threading* e *green thread*, também foi utilizado o termo *cooperative multitasking* que, apesar de não ser um sinônimo, também apresenta artigos da área.

Os termos *coroutine*, *asynchronous algorithm*, *asynchronous programming*, *asynchronous network*, *asynchronous communication* e *async* foram utilizados para trazer artigos relacionados a *programação assíncrona*. Não foi possível utilizar termos mais abrangentes como *asynchronous*, pois acabavam gerando resultados com grande número de artigos não relacionados, devido a amplitude de utilização desse termo.

Já os termos *embedded system*, *Internet of Things*, *IoT*, *ubiquitous system*, *ubiquitous computing* e *real-time computing* são utilizados para trazer artigos relacionados a *Sistemas embarcados*.

A partir da busca realizada nas bases de dados em janeiro de 2023, informações de título, abstract, autores, editora e tipo foram obtidas. O número total de artigos retornados foi de 448, distribuídos da seguinte forma entre as bases de dados:

- **Scopus**: 256
- **IEEE**: 98
- **Web of Science**: 94

3.4 Critério e Processo de Seleção

Tendo-se o conjunto de artigos retornados pelas bases de dados, os mesmos foram selecionados usando a ferramenta *Rayyan* (OUZZANI1 et al., 2016). Dos 448 artigos, 154 artigos (34,4% do total) foram excluídos por serem duplicados. Na sua distribuição, 95 artigo (33% do total após a remoção dos duplicados) são classificados como "Journal Article" (artigos de periódicos) e esses foram triados por 2 revisores durante o processo de seleção, enquanto os outros tipos, como "Conference Papers" (artigos de congressos) foram triados por apenas 1 revisor[5]. O processo de seleção excluiu artigos que não eram relacionados a área de estudo, sendo que alguns dos artigos apresentam múltiplos motivos para exclusão. Na tabela 1 são apresentados os números de exclusão por motivo, em (BONATO; PANISSON, 2023) pode ser encontrado a tabela com os dados de inclusão e exclusão com todo os artigos.

Tabela 1 – Número de artigos não relacionados por tipo.

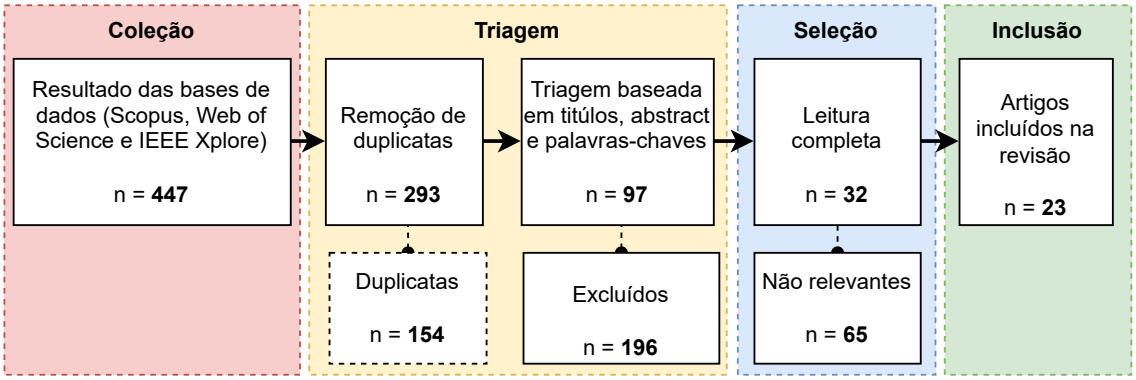
Critério	**Quantidade**
Redes	60
Hardware	48
Projeto	48
Sistemas distribuídos	31
Protocolo	21
Algoritmo	21
Modelo	19
AI	4
Segurança	3
Teoria da Computação	4
Blockchain	2

Fonte: O Autor

Após a triagem, um total de 32 artigos foram selecionados para leitura completa. Dentre esses 5 foram excluídos por motivo de serem fora de escopo deste trabalho, e 2 foram excluídos por não possuírem acesso gratuito.

[5] Estudos demonstram que triagem feita por apenas 1 revisor deixam de incluir cerca de 13% dos estudos relevantes (GARTLEHNER et al., 2020), além de incluir risco de bias nas escolhas de artigos. Portanto esse estudo está parcialmente sujeito a essa limitação.

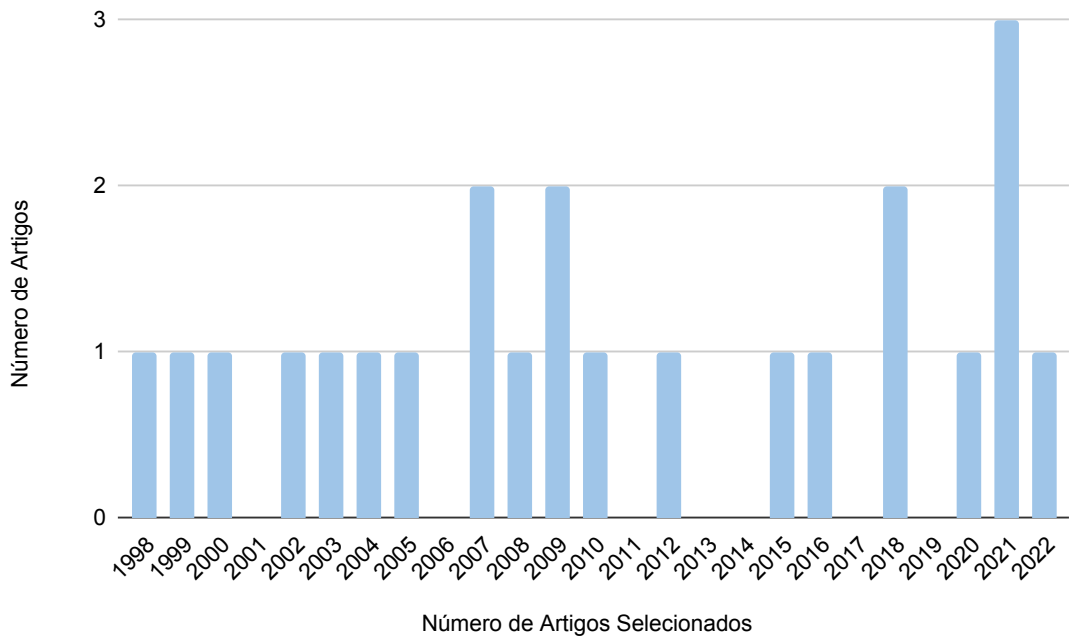
Figura 1 – Processo de Seleção.



Fonte: O Autor

Na Figura 1, são apresentadas as etapas de seleção dos artigo, incluindo a informação sobre o número de artigos em cada parte do processo. Também, no gráfico da Figura 2 é apresentado o conjunto de artigos resultante do processo de seleção distribuído por ano de publicação. Percebe-se que foram incluídos artigos de 1998 até 2022. Percebe-se também que o ano de 2021 possui a maior quantidade de artigos.

Figura 2 – Número de artigos selecionados por ano.



Fonte: O Autor

Na seção seguinte, são apresentados os resultados obtidos para cada questão de pesquisa descrita nessa seção 3.

4 Resultados

4.1 RQ1. Qual o paradigma da linguagem de programação utilizada?

Foram encontrados 6 diferentes paradigmas associados as diferentes linguagens de programação utilizadas, sendo eles Orientado a Objeto (10 referências), Procedural (6 referências), Orientada à Evento (3 referências), Funcional (2 referências), e Síncrona (2 referência). Vale ressaltar, que algumas linguagens são multi-paradigmas (possuem mais de um paradigma), então, apenas o paradigma que possui relação com o objetivo deste estudo foi considerado. A Tabela 2 apresenta a relação dos paradigmas utilizados pelos trabalhos selecionados.

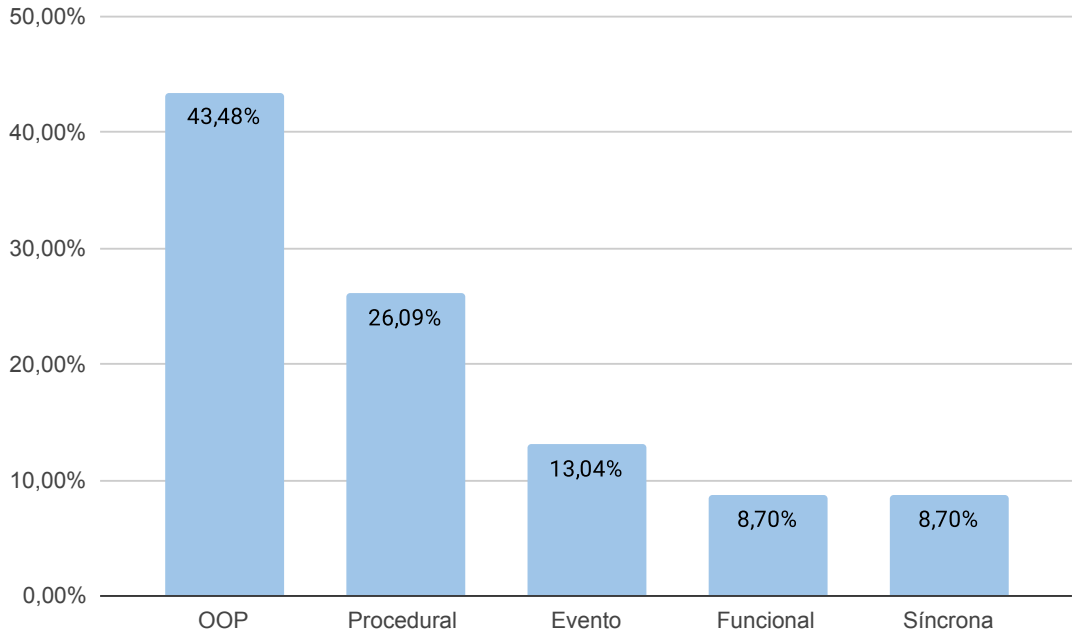
Tabela 2 – Paradigmas usados por artigo.

Paradigma	**Referências**
Orientado a Objeto	(ALRAHMAWY; WELLINGS, 2009) (KOSCHEL et al., 2012) (BELSON et al., 2019) (SUNG et al., 2001) (BERONIć et al., 2022) (ENCHEVA; TUMIN, 2021) (MOHAMMED et al., 2021) (PASSERONE et al., 1998) (JEON; LEE; HWANG, 2021) (ZHAO; WELLINGS; KORSHOLM, 2015)
Procedural	(N.J.; S.; A.G., 2004) (SHIVSHANKAR; VANGARA; DEAN, 2005) (MILLS; JEFFERY, 2016) (AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003) (IERUSALIMSCHY, 2010) (SEO et al., 1999)
Evento	(KHEZRI; SARRAM; ADIBNIYA, 2008) (SALLAI; MARóTI; LéDECZI, 2005) (COHEN et al., 2007)
Funcional	(ZHAO; BERGER; LI, 2020) (BäRENZ; PEREZ, 2018)
Síncrona	(BERRY; SENTOVICH, 1999) (VECCHIE; TALPIN; SCHNEIDER, 2009)

Fonte: O Autor

A Figura 3 apresenta o gráfico demonstrando as proporções dos paradigmas utilizados. Percebe-se que Linguagens Orientadas a Objeto e Procedural dominam cerca de 70% dos paradigmas de programação utilizados. Isso se deve a sua eficiência no uso de hardware, ao fato de historicamente serem os paradigmas mais utilizados e do vasto número de linguagens existentes nesses paradigmas.

Figura 3 – Paradigmas utilizados.



Fonte: O Autor

4.2 RQ2. Qual linguagem de programação e/ou biblioteca foi utilizada?

Em relação as linguagens de programação utilizadas, 9 diferentes linguagens foram encontradas, sendo elas Java (6 referências), C (4 referências), C++ (2 referências), nesC (3 referências), Esterel (2 referências), Python (1 referência), Unmicon (1 referência), Lua (1 referência), Haskell (1 referência) e Bibliotecas sem nenhuma linguagem específica (2 referência). A Tabela 3 apresenta a relação entre linguagens de programação (ou bibliotecas) em relação aos trabalhos selecionados.

Percebe-se que apesar de sua dominância em sistemas embarcados, as linguagens de programação C e C++ estão presentes em 6 referências, número igual a linguagem de programação Java. Isso provavelmente se deve a diversos estudos com objetivo de tornar a linguagem Java uma linguagem competitiva em sistemas embarcados, movimento que iniciou nos anos 2000 (STEINHORN, 1998).

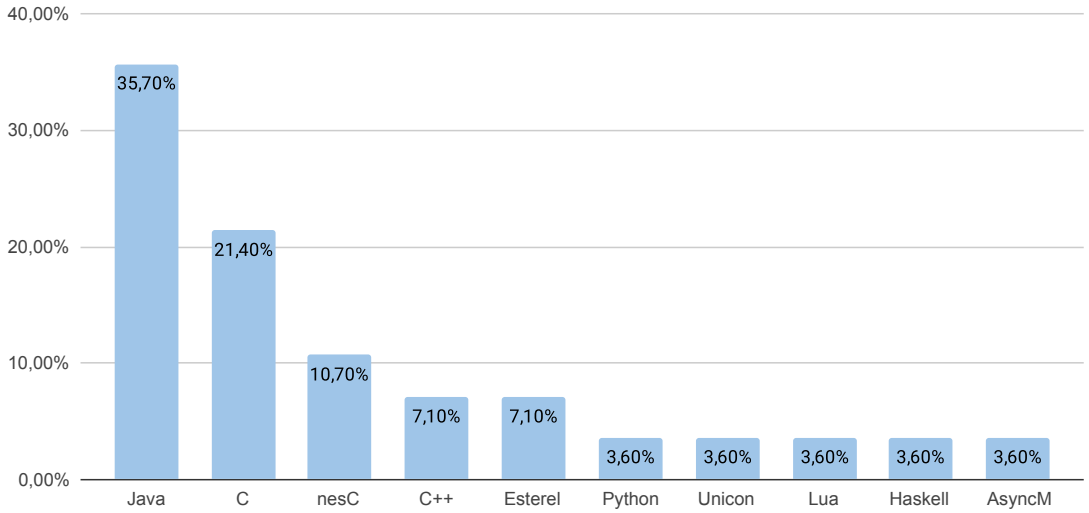
Também é importante ressaltar que diferentes linguagens possuem diferentes usos. Em muitos casos, a linguagem de programação C foi utilizada para implementar técnicas concorrentes a nível de um sistemas operacional embarcado. Enquanto a linguagem de programação Java foi citada com maior frequência em casos onde sua máquina virtual recebeu implementações. A Figura 4 apresenta o gráfico demonstrando as proporções das linguagens utilizadas, destacando a dominância da linguagens de programação Java e C/C++, e a linguagem nesC (usada no tinyOS), que apesar de menos popular, pode ser vista como uma linguagem de nicho no contexto de sistemas distribuídos com foco em redes, apresentando 3 artigos sobre a mesma.

Tabela 3 – Linguagens por referências.

Linguagem	Referências
Java	(ALRAHMAWY; WELLINGS, 2009) (KOSCHEL et al., 2012) (SUNG et al., 2001) (BERONIć et al., 2022) (PASSERONE et al., 1998) (ZHAO; WELLINGS; KORSHOLM, 2015)
C	(N.J.; S.; A.G., 2004) (SHIVSHANKAR; VANGARA; DEAN, 2005) (AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003) (SEO et al., 1999)
C++	(BELSON et al., 2019) (MOHAMMED et al., 2021)
nesC	(SALLAI; MARóTI; LéDECZI, 2005) (KHEZRI; SARRAM; ADIBNIYA, 2008) (COHEN et al., 2007)
Esterel	(VECCHIE; TALPIN; SCHNEIDER, 2009) (BERRY; SENTOVICH, 1999)
Python	(ENCHEVA; TUMIN, 2021)
Unicon	(MILLS; JEFFERY, 2016)
Lua	(IERUSALIMSCHY, 2010)
Haskell	(BäRENZ; PEREZ, 2018)
Biblitecas	(ZHAO; BERGER; LI, 2020) (JEON; LEE; HWANG, 2021)

Fonte: O Autor

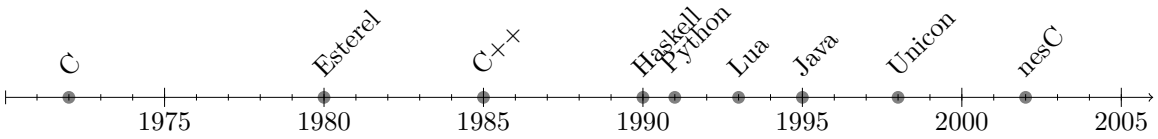
Figura 4 – Linguagens utilizadas.



Fonte: O Autor

Quando se analisa os anos em que as linguagens apresentadas na Tabela 3 e gráfico da Figura 4 foram lançadas, percebe-se que grande parte das linguagens utilizadas pelos artigos selecionados foram lançadas após os anos 90. Porém, a linguagem C aparece como caso atípico de linguagem da década de 70 que ainda possui amplo uso. Na Figura 5 é apresentado uma linha temporal de lançamento das linguagens mencionadas.

Figura 5 – Linha cronológica das linguagens.



Fonte: O Autor

Tabela 4 – Linguagens por referências.

Ano	Linguagem	Referências
1998	Java (PureSR)	(PASSERONE et al., 1998)
1999	Esterel	(BERRY; SENTOVICH, 1999)
1999	C (OS)	(SEO et al., 1999)
2001	Java (PersonalJava)	(SUNG et al., 2001)
2003	C (C-like[6])	(AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003)
2004	C (OS)	(N.J.; S.; A.G., 2004)
2005	TinyVT (nesC-like)	(SALLAI; MARóTI; LéDECZI, 2005)
2005	C (OS)	(SHIVSHANKAR; VANGARA; DEAN, 2005)
2007	nesC	(COHEN et al., 2007)
2008	nesC	(KHEZRI; SARRAM; ADIBNIYA, 2008)
2009	Java (RTSJ)	(ALRAHMAWY; WELLINGS, 2009)
2009	Esterel	(VECCHIE; TALPIN; SCHNEIDER, 2009)
2010	Lua	(IERUSALIMSCHY, 2010)
2012	Java (OSGi)	(KOSCHEL et al., 2012)
2015	Java (SCJ)	(ZHAO; WELLINGS; KORSHOLM, 2015)
2016	Unicon	(MILLS; JEFFERY, 2016)
2018	Haskell (Rhine)	(BäRENZ; PEREZ, 2018)
2018	Java (HL7)	(JEON; LEE; HWANG, 2021)
2020	AsyncM (lib)	(ZHAO; BERGER; LI, 2020)
2020	C++	(BELSON et al., 2019)
2021	Python	(ENCHEVA; TUMIN, 2021)
2021	C++	(MOHAMMED et al., 2021)
2022	Java	(BERONIć et al., 2022)

Fonte: O Autor

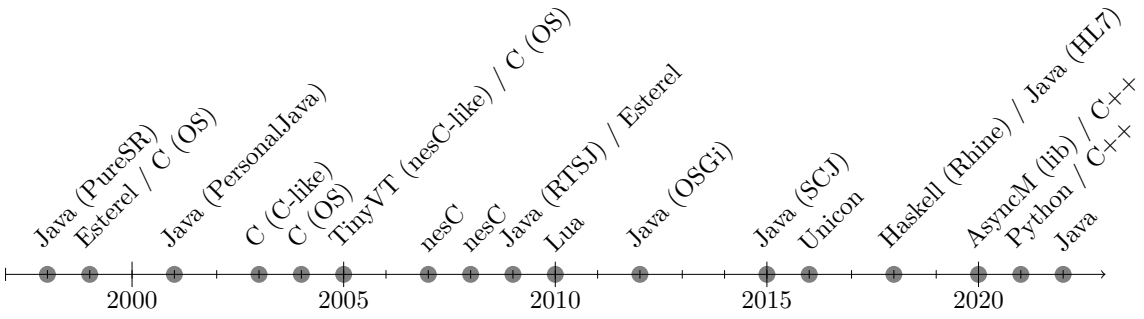
4.3 RQ3. Quando ocorreu a inclusão das técnicas de concorrência nas linguagens identificadas?

Na Tabela 4 são apresentados os anos de inclusão das técnicas de concorrência nas linguagens levantadas, bem como o artigo de referência. É importante lembrar, que

múltiplas implementações podem ter ocorrido em uma mesma linguagem. Em algumas linguagens, como C, existem implementações mais focadas a nível de utilitário fornecido pelo sistema operacional. Em diversas linguagens populares houveram a inclusão oficial, como C++, Python e Java. Em linguagens como C, a inclusão se deu por meio do sistema operacional.

Os dados levantados demonstram que o problema de concorrência em linguagens de programação para sistemas embarcados ainda possui desenvolvimento, sendo constante as produções desde o final dos anos 90. Um destaque bastante recente é Java, que possuía, originalmente, diferentes bibliotecas que implementavam *green threads*, porém, com sua versão lançada oficialmente em 2022 agora possui uma implementação na própria linguagem (ORACLE, 2022). Na Figura 6 é apresentado uma linha temporal destacando os anos de inclusão, para as linguagens listadas.

Figura 6 – Linha cronológica da inclusão das técnicas.



Fonte: O Autor

4.4 RQ4. Qual tipo de concorrência foi utilizado?

É notável que, em linguagens de baixo nível, houve um enfoque recente em concorrência via corrotinas, sendo apresentado em 10 artigos (BELSON et al., 2019; MOHAMMED et al., 2021; IERUSALIMSCHY, 2010; VECCHIE; TALPIN; SCHNEIDER, 2009; N.J.; S.; A.G., 2004; AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003; COHEN et al., 2007; SALLAI; MARóTI; LéDECZI, 2005; SHIVSHANKAR; VANGARA; DEAN, 2005; KHEZRI; SARRAM; ADIBNIYA, 2008). Sua implementação tem sido preferível em contrapartida com *green threads*, devido ao fato de que elas não precisarem de *run-time* ou máquina virtual, permitindo assim, o uso em sistemas em que o tempo de execução e latência tem prioridade. Destaca-se também sua capacidade de ser utilizada quando existe a necessidade da espera por um evento externo, onde uma corrotina pode ficar pausada até que um evento específico ocorra (BELSON et al., 2019). Há também casos, como Icon e Unicon, onde corrotinas foram adicionadas na forma de co-expressões, que fazem parte do fundamento da linguagem (MILLS; JEFFERY, 2016).

Em casos em que o foco é na troca de mensagens, normalmente via rede, programação assíncrona é mais utilizada, aparecendo em 6 artigos (KOSCHEL et al., 2012; ZHAO; BERGER; LI, 2020; JEON; LEE; HWANG, 2021; BERRY; SENTOVICH, 1999; ENCHEVA; TUMIN, 2021; ALRAHMAWY; WELLINGS, 2009). Isso se deve ao fato que as alternativas possuem desvantagens: manipular os eventos com *call-backs*[7] torna a leitura do código difícil, e bloquear a execução de um programa até que o evento responda acaba

[7] Call-back se refere ao registro de funções que são chamadas quando um evento específico ocorre.

causando menor performance. Em sistemas distribuídos, o assincronismo permite uma arquitetura capaz de suportar uma quantidade grande de conexões e se adéqua bem a necessidade de diferentes agentes que não são sincronizados (ALRAHMAWY; WELLINGS, 2009).

As *green threads* se apresentam como uma boa alternativa em linguagens que já possuem uma máquina virtual em 2 casos (SUNG et al., 2001; BERONIć et al., 2022). Isso se deve ao fato que a máquina virtual já é uma *run-time*, então existe um custo de performance baixo de usá-la para controlar as *green threads*. Além disso, elas podem ser implementadas em extensões de linguagem, com Fibers que já estão disponíveis em sistemas operacionais (PASSERONE et al., 1998).

Linguagens funcionais, por possuírem outro paradigma, costumeiramente possuem soluções diferentes, que foram apresentadas em 2 estudos. Uma delas é a programação funcional reativa, geralmente implementado através de uma biblioteca (JEON; LEE; HWANG, 2021; BäRENZ; PEREZ, 2018).

Existem também linguagens síncronas, com alto nível de inspiração de linguagens de descrição de hardware. Em geral, essas linguagens são reativas, ou seja, elas estão obtendo suas entradas continuamente e as processando. Nesses casos, elas implementam construções semelhantes as corrotinas. Porém, trazem algumas diferenças em relação a pilha de execução da corrotina, salvando apenas o contador do programa e um ponteiro para a janela atual (VECCHIE; TALPIN; SCHNEIDER, 2009).

É também frequente que linguagens tenham implementações de múltiplos métodos, com foco em diferentes casos de uso (ZHAO; WELLINGS; KORSHOLM, 2015). Um exemplo disso é Java, que já possuía acesso a threads de kernel, mas posteriormente, com o projeto *PersonalJava*, trouxe uma implementação de green threads, demonstrada em 2 artigos (SUNG et al., 2001; BERONIć et al., 2022), e mais recentemente, incluiu oficialmente na linguagem uma implementação de virtual threads. Existe também, projetos feitos antes da popularização de processadores com paralelismo real tinham perspectivas diferentes em relação ao paralelismo. Comumente citando processos como forma comum de paralelismo. Portanto a discussão acerca de paralelismo de kernel, processos e *user threads* vinha sem a visão retrospectiva. Então frequentemente possuíam múltiplas implementações, lado a lado (SEO et al., 1999).

4.5 RQ5. Quais foram os motivos de incorporar concorrência?

Uma arquitetura comum de um sistema embarcado evolve diferentes linhas de execução para rede, Interface de Usuário (UI) e controle, por exemplo. Podendo ser implementadas com threads de kernel, ou corrotinas, o que elimina as trocas de contexto existentes em *threads de kernel*, motivação apresentada em 4 estudos (N.J.; S.; A.G., 2004; COHEN et al., 2007; SHIVSHANKAR; VANGARA; DEAN, 2005; KHEZRI; SARRAM; ADIBNIYA, 2008). Isso ocorre por conta das corrotinas precisarem apenas da troca da pilha de quadro da corrotina, permitindo uma melhora significativa no tempo de execução de uma tarefa, o que foi encontrado em 4 artigos (BELSON et al., 2019; MOHAMMED et al., 2021; IERUSALIMSCHY, 2010; SHIVSHANKAR; VANGARA; DEAN, 2005). A sua implementação pode também reduzir o uso de memória, já que é apenas necessário salvar dados referentes a atual função em execução (BERONIć et al., 2022).

Uma das principais vantagens apresentadas pelas *green threads* é que sua construção se assemelha a de *threads de kernel*, sendo familiar aos programadores e permitindo sua

inclusão sem a necessidade de grandes refatorações no código, observação feita em 2 artigos (BERONIć et al., 2022; ZHAO; WELLINGS; KORSHOLM, 2015).

Existem também, 5 casos apresentados de linguagens com o foco em trocas de mensagens em rede, onde programação assíncrona é o método mais eficiente. Permitindo um tempo de processamento muito menor, visto que o processador fica livre para realizar outras tarefas enquanto há a espera pela resposta das mensagens (KOSCHEL et al., 2012; ZHAO; BERGER; LI, 2020; JEON; LEE; HWANG, 2021; BERRY; SENTOVICH, 1999; ALRAHMAWY; WELLINGS, 2009). Usar o assincronismo ajuda também no desacoplamento de quantas *threads de hardware* o software possui acesso, com quantas conexões podem existir paralelamente (ALRAHMAWY; WELLINGS, 2009).

Como programação funcional trás um enfoque em controle de efeitos colaterais, a implementação de programação assíncrona, utilizando uma interfaces monádicas[8], se adéqua fortemente. Sendo demonstrada como um design de biblioteca (ZHAO; BERGER; LI, 2020; BäRENZ; PEREZ, 2018). Problemas relacionados a sincronização podem aparecer, porém, o uso da tipagem, já presente na linguagem, pode ser útil para auxiliar o processo, sem a necessidade de trabalho manual (BäRENZ; PEREZ, 2018).

Em casos onde um sistema operacional embarcado é utilizado, o uso e implementação não são soluções *cross-platform*[9] ou acessíveis sem bibliotecas. Isso torna a inclusão dessas técnicas nas linguagens mais fáceis e práticas para o programador, notado por 2 estudos (MOHAMMED et al., 2021; BERONIć et al., 2022).

Algumas linguagens tem como princípio de design a simplicidade, tanto da implementação da linguagem, como de seu uso. Nesses 2 casos, corrotinas se tornam uma alternativa que permite seguir este princípio (IERUSALIMSCHY, 2010; AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003).

Existem também 6 casos onde a gerência da ordem de execução de tarefas é importante, e portanto, a natureza cooperativa de corrotinas e assincronismo se torna mais adaptada (ENCHEVA; TUMIN, 2021; AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003; COHEN et al., 2007; SALLAI; MARóTI; LéDECZI, 2005; SHIVSHANKAR; VANGARA; DEAN, 2005; KHEZRI; SARRAM; ADIBNIYA, 2008).

Para sistemas em tempo real, o controle acerca da execução exige que o método utilizado para concorrência seja compatível com os requerimentos. A dependência do escalonador[10] de threads do sistema operacional pode ser vantajosa caso ele seja feito para sistemas em tempo real, porém, green threads podem trazer maior controle acerca de seu escalonamento, sendo apresentado em 2 artigos (SEO et al., 1999; PASSERONE et al., 1998).

Também é notável que em processadores *single-core* threads de nível de usuário podem possuir performance melhor que threads de kernel (SEO et al., 1999), isso se deve ao fato que elas não necessitam ir ao nível do sistema operacional, o que é frequentemente custoso.

Linguagens síncronas apresentam motivações diferentes das demais, onde, ao invés do uso de corrotinas para o controle da execução em paralelo, elas utilizam corrotinas como

[8] Interface Monádica se refere a uma interface que se utiliza de *Monads*, que, por sua vez, são estruturas genéricas que combinam funções e encapsulam seu resultado em um tipo com funcionalidade adicionais (WADLER, 2009).

[9] *Cross-platform* se refere a soluções que independem da plataforma final em que será utilizada.

[10] Escalador é o responsável por escolher qual será a próxima tarefa a ser executada.

forma de modularizar o código. Porém, o seu uso necessita de um compilador que tenha capacidade de reescrever o código para trazer as garantias de sincronização (VECCHIE; TALPIN; SCHNEIDER, 2009). Existem casos onde a linguagem que utiliza corrotinas pode fazer uma pré analise de quando as trocas vão ocorrer, permitindo maior otimização (N.J.; S.; A.G., 2004).

Linguagens como Icon ou Unicon, possuem um paradigma diferente, e partem do princípio onde tudo é uma corrotina (nessas linguagens chamadas de co-expressões), onde elas param sua execução e retornam valores (MILLS; JEFFERY, 2016). O retorno dos valores pode ser composto tanto de um resultado, que pode implicitamente continuar a execução, quanto de um erro, que pode ser implicitamente propagado. Assim, o programador não precisa escrever código responsável pelo controle de fluxo em casos de erros.

4.6 RQ6. Quais as limitações encontradas?

Em algumas linguagens, como o C++, a implementação de corrotinas pode oferecer limitações no controle fino da memória da pilha de quadro da corrotina, o que pode restringir seu uso em casos onde é necessário manipulação direta da memória, como o desenvolvimento de sistemas operacionais embarcados (BELSON et al., 2019). Devido a dificuldade de implementar alocadores de memória definidos por usuário para as alocações de memórias das corrotinas, em 2 casos apresentados, as linguagens não possuem essa capacidade, podendo limitar seu uso em sistemas com organização de memória que não permite alocações dinâmicas típicas (MOHAMMED et al., 2021; COHEN et al., 2007).

Fazer o controle de fluxo de execução a nível de usuário pode trazer uma complexidade de implementação muito maior que utilizar o que já é providenciado pelo sistema operacional. Tanto bibliotecas assíncronas (KOSCHEL et al., 2012) quanto green threads podem trazer dificuldades em alguma parte da implementação (SUNG et al., 2001; SEO et al., 1999), algo notado em 3 artigos. Isso pode levar a implementações incompletas, com mais limitações.

Nas linguagens que possuem *garbage collector*[11], o uso indiscriminado de corrotinas e green threads pode também trazer problemas no uso de alocações dinâmicas. Por conta de seu baixo custo para criação e liberação, usuários podem considerar a criação um grande número de *green threads*. Porém, esse grande número pode gerar latência no *garbage collector* quando ele for desalocar essa memória, gerando problemas de performance após sua criação, apresentado em 2 artigos (BERONIć et al., 2022; ENCHEVA; TUMIN, 2021).

Algumas técnicas, como programação funcional reativa, por serem muito recentes, apresentam poucos exemplos práticos de seu uso, podendo ter desafios não esperados no seu uso, isso foi demonstrado em 3 artigos (BäRENZ; PEREZ, 2018; JEON; LEE; HWANG, 2021; ZHAO; BERGER; LI, 2020).

Um problema apresentado para *green threads* é o baixo controle de *thread migration*[12]. Uma possível solução seria a definição de uma rotina de migração e regras de migração, que definem quando e para onde a tarefa deve ser migrada (ZHAO; WELLINGS; KORSHOLM, 2015), porém, nenhuma linguagem estudada apresenta essa característica atualmente.

[11] *Garbage colletor* é uma *run-time* responsável por desalocar memória dinâmica alocada pelo programa em execução.

[12] *Thread migration* ocorre quando um dada tarefa precisa ser movida de uma thread de hardware para outra (INTEL, 2023).

Em 4 casos, uma mudança no método de concorrência exige mudança na maneira de estruturar software, e portanto pode trazer custos adicionais. Consegue-se encontrar exemplos de programação assíncrona (ALRAHMAWY; WELLINGS, 2009; ENCHEVA; TUMIN, 2021), corrotinas (SALLAI; MARóTI; LéDECZI, 2005), e a linguagens como Icon ou Unicon (MILLS; JEFFERY, 2016).

Enquanto em 3 diferentes estudos, o controle da preempção pode ser visto como vantajoso, ele também pode fazer com que as tarefas que devam ser executadas tenham que esperar para a execução. Em um sistema em tempo real, onde essas tarefas podem ser baseadas em sensores ou atuadores, essa possível latência pode ser indesejada, ou até mesmo, tarefas longas podem criar *starvation*[13]. Uma possível solução é que partes do programa possam ser preemptadas, enquanto outras partes sejam completamente cooperativas (AKHMETSHINA; GBURZYNSKI; VIZEACOUMAR, 2003; KHEZRI; SARRAM; ADIBNIYA, 2008; IERUSALIMSCHY, 2010).

Nos casos onde há assincronismo com uma hierarquia de comunicação (um método assíncrono comunicando com outro), a arquitetura de comunicação precisa ser robusta o suficiente para permitir isso, sendo necessário para sistemas de maior escala (PASSERONE et al., 1998).

Em linguagens síncronas, o uso de corrotinas apresenta um aumento substancial na troca de contextos, e também traz dificuldade de implementação de processadores com paralelismo de hardware (VECCHIE; TALPIN; SCHNEIDER, 2009). Implementações assíncronas trazem dificuldade de implementação em hardware (BERRY; SENTOVICH, 1999).

Algumas linguagens eliminam a ideia de um sistema operacional com escalonamento dinâmico, onde escalonamento é realizado enquanto o programa está em execução, e realizam escalonamento estático, onde os locais onde pode ocorrer trocas de contexto são pré-analisados e fixos. Assim, a linguagem permite que sejam trocados por corrotinas. Isso gera ganho de performance, porém, código deve ser copiado em diversas partes do programa pelo compilador, gerando um tamanho do executável que pode ser muito maior (N.J.; S.; A.G., 2004).

Uso de corrotinas possui performance que pode variar de acordo com o microcontrolador sendo utilizado, em alguns casos tendo ganho de performance, em outros tendo perda (SHIVSHANKAR; VANGARA; DEAN, 2005). Corrotinas também podem possuir *debugging* ineficiente (MILLS; JEFFERY, 2016).

5 Discussão

De forma conclusiva, fica evidente a necessidade de processamento concorrente para sistemas embarcados devido ao interesse nas linguagem em realizar implementações de concorrência, como apresentado nas questões de pesquisa 4.2 e 4.3, porém, também fica evidente que as alternativas aos paralelismo de hardware não oferecem vantagens universais, como apresentado na questão de pesquisa 4.6. Em 12 casos, essas técnicas já são embutidas nas linguagens, em 8, são implementadas através de bibliotecas e em 3 são bibliotecas do sistema operacional. Essa fragmentação da implementação deixa confuso o processo de escolha de qual técnica usar, e obriga o programador a aprender seu uso em

[13] *Starvation* ocorre quando uma *thread* não dá controle para outras, fazendo com que elas não sejam executadas, ou sejam executadas com menos frequência que necessário.

cada sistema utilizado, ao invés de apenas aprender a linguagem de programação e fazer uso de uma técnica universal. Portanto, existe a necessidade de maior padronização delas, e, sua implementação a nível de linguagem pode oferecer isso.

As linguagens de programação que incorporam *green threads* proporcionam um paradigma de programação semelhante ao já familiar das *threads de kernel*, tendo assim, uma curva de aprendizado leve. Também apresentam uma capacidade de concorrência maior em relação a *threads de kernel*, devido ao menor custo computacional, sendo até possível instanciar um grande número de threads sem grandes problemas. No entanto, *green threads* compartilham algumas das desvantagens das threads de kernel, como a preempção, o que é problemático em códigos críticos que precisam ser executados em totalidade. Isso leva a necessidade de ferramentas de sincronização de fluxo, como *mutexes*[14] e *semáforos*[15], que podem criar erros como *dead-locks*[16] e *race conditions*[17].

Por outro lado, as linguagens com corrotinas possuem uma execução cooperativa, onde a corrotina é responsável por "pausar" sua própria execução, corrigindo o problema de códigos que devem ser executados em sua totalidade. Porém, introduz uma nova classe de problemas: uma corrotina pode ficar presa, por exemplo, através de um laços de repetição infinito, e nunca permitir que outras tarefas executem. Este problema causa *starvation*.

O Assincronismo apresenta grande relação com tarefas com a necessidade contínua de IO. Possuindo semelhanças as corrotinas, se apresenta como boa solução para casos onde o foco é troca de mensagens. Porém, se limita a esse tipo de problemas.

6 Conclusão

Este artigo apresentou uma análise sobre o uso de concorrência em linguagens de programação utilizadas em sistemas embarcados e IoT, através de uma revisão de escopo da literatura. Através do estudo realizado, percebe-se que existem três diferentes opções para as linguagens e bibliotecas implementarem concorrência em sistemas embarcados: *green threads*, corrotinas e assincronismo. *Green threads* apresentam semelhança na programação ao modelo clássico de threads de kernel, porém, são preemptivas e trazem pouco controle de recursos. Corrotinas contrastam com sua natureza cooperativa, e demonstram uma possível alternativa em casos onde códigos críticos não devem possuir sua execução parada. Assincronismo traz ótima concorrência quando existe a necessidade de IO, porém, esse é seu único foco. A melhor alternativa para um dado projeto vai depender dos requisitos que ele apresenta. Existindo casos de uso para cada uma das alternativas.

A revisão apresentada nesse artigo possuiu foco em artigos publicados, e não incluiu conteúdo da literatura cinzenta e isso pode levar a um viés, onde parte do conteúdo relevante não tenha sido incluído. Percebe-se também, que algumas linguagem com maior interesse acadêmico (como nesC) aparecem com frequência, apesar de seu uso na indústria não ser difundido. Outra fonte de possível viés é o fato de que apenas uma parcela dos artigos foram triados por múltiplos revisores, o que pode levar alguns artigos a não serem

[14] *Mutex* é o mecanismo de controle que tem como objetivo impedir que mais de uma linhas de execução acessem uma *sessão crítica* ao mesmo tempo.

[15] *Semáforo* é o mecanismo de controle que tem como objetivo permitir apenas um determinado número de linhas de execução acessem uma *sessão crítica* ao mesmo tempo.

[16] *Dead-lock* é a condição onde múltiplas linhas de execução são paradas por estarem esperando um recurso que as outras estão também esperando.

[17] *Race condition é a condição onde o resultado de uma operação depende de uma ordem não controlada pelo usuário.*

incluídos. Mesmo com essas considerações, considera-se que, através das análises realizadas, elas não compromete o resultado final desse estudo.

Como trabalho futuro pode-se realizar uma análise na literatura cinzenta, dando foco em conteúdos que estão oficialmente ligados as linguagens de programação (por exemplo, as linguagens levantadas pelo estudo apresentado nesse documento). Nesse estudo, pode-se incluir fontes como as páginas web das linguagens de programação, e relatórios técnicos publicados pelos criadores e mantenedores das linguagens.

Assim, este estudo conseguiu reunir como diversas linguagens de programação implementam concorrência, além de demonstrar tendências de paradigmas, linguagens, tipos e problemas nas implementações. Dessa forma, respondendo como é o estado da concorrência em sistemas embarcados a nível de implementação nas linguagens e bibliotecas.

Referências

AKHMETSHINA, E.; GBURZYNSKI, P.; VIZEACOUMAR, F. Picos: A tiny operating system for extremely small embedded platforms. CSREA, 2003. Citado (6) vezes nas páginas [9, 11, 12, 13, 15 e 17].

ALRAHMAWY, M.; WELLINGS, A. An rtsj-based reconfigurable server component. ACM, 2009. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 17].

ARNOLD, M. et al. A survey of adaptive optimization in virtual machines. IEEE, 2005. Citado na página [4].

BARR, M.; MASSA, A. *Programming embedded systems: with C and GNU development tools*. [S.l.]: OŔeilly Media, 2006. Citado na página [3].

BELSON, B. et al. A survey of asynchronous programming using coroutines in the internet of things and embedded systems. ACM, 2019. Citado na página [5].

BELSON, B. et al. C++20 coroutines on microcontrollers - what we learned. IEEE, 2019. Citado (6) vezes nas páginas [9, 11, 12, 13, 14 e 16].

BERONIć, D. et al. Comparison of structured concurrency constructs in java and kotlin – virtual threads and coroutines. IEEE, 2022. Citado (6) vezes nas páginas [9, 11, 12, 14, 15 e 16].

BERRY, G.; SENTOVICH, E. M. An implementation of constructive synchronous programs in polis. Kluwer, 1999. Citado (6) vezes nas páginas [9, 11, 12, 13, 15 e 17].

BONATO, F.; PANISSON, A. *Dados do processo de exclusão dos artigos*. 2023. Disponível em: <https://github.com/felipe-bonato/review-result-concurrency_embedded_systems.git>. Citado na página [7].

BäRENZ, M.; PEREZ, I. Rhine: Frp with type-level clocks. ACM SIGPLAN, 2018. Citado (6) vezes nas páginas [9, 11, 12, 14, 15 e 16].

CHRISTIAN, M. Multicore processors: challenges, opportunities, emerging trends. 2014. Citado na página [3].

COHEN, M. et al. Using coroutines for rpc in sensor networks. IEEE, 2007. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 16].

CONWAY, M. E. Design of a separable transition-diagram compiler. ACM, 1963. Citado na página [4].

ELIZAROV, R. et al. Kotlin coroutines: design and implementation. ACM, 2021. Citado na página [4].

ENCHEVA, S.; TUMIN, S. Cooperative dynamic programmable devices using actor model for embedded systems of microcontrollers. Springer, 2021. Citado (7) vezes nas páginas [9, 11, 12, 13, 15, 16 e 17].

GARTLEHNER, G. et al. Single-reviewer abstract screening missed 13 percent of relevant studies: a crowd-based, randomized controlled trial. PUBMED, 2020. Citado na página [7].

IERUSALIMSCHY, R. Programming with multiple paradigms in lua. WFLP, 2010. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 17].

INTEL. *OS Thread Migration*. 2023. Disponível em: <https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/os-thread-migration.html>. Citado na página [16].

JEON, D. cheol; LEE, D. hyung; HWANG, H. Reactive server interface design for real-time data exchange in multiple data source and clients. IEEE, 2021. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 16].

KHEZRI, M.; SARRAM, M. A.; ADIBNIYA, F. Simplifying concurrent programming of networked embedded systems. IEEE, 2008. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 17].

KNIGHT, W. Two heads are better than one. IEE Review, 2005. Citado (2) vezes nas páginas [3 e 4].

KOSCHEL, A. et al. Asynchronous messaging for osgi. Journal of Computing and Information Technology, 2012. Citado (6) vezes nas páginas [9, 11, 12, 13, 15 e 16].

MARKETS AND MARKETS. *Embedded System Market by Hardware*. 2020. Disponível em: <https://www.marketsandmarkets.com/Market-Reports/embedded-system-market-98154672.html>. Citado na página [3].

MARWEDEL, P. *Embedded System Design*. [S.l.]: Springer, 2021. 128 p. Citado na página [3].

MILLS, P.; JEFFERY, C. Embedding concurrent generators. IEEE, 2016. Citado (6) vezes nas páginas [9, 11, 12, 13, 16 e 17].

MOHAMMED, A. S. et al. Development of coroutines usage model for cooperative multitasking implementation on the systems with limited resources. Annals of Operations Research, 2021. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 16].

MOORE, G. E. Cramming more components onto integrated circuits. IEEE, 1965. Citado na página [3].

MOURA, A. L. D.; IERUSALIMSCHY, R. Revisiting coroutines. ACM, 2009. Citado na página [4].

N.J., K.; S., S.; A.G., D. Asynchronous software thread integration for efficient software implementations of embedded communication protocol controllers. ACM SIGPLAN, 2004. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 16 e 17].

ORACLE. *JDK 19*. 2022. Disponível em: <https://openjdk.org/projects/jdk/19/>. Citado na página [13].

OUZZANI1, M. et al. Rayyan — a web and mobile app for systematic reviews. BioMED, 2016. Citado na página [7].

PASSERONE, C. et al. Modeling reactive systems in java. CODES/CASHE, 1998. Citado (6) vezes nas páginas [9, 11, 12, 14, 15 e 17].

PRISMA. *PRISMA*. 2020. Disponível em: <https://www.prisma-statement.org/>. Citado na página [5].

ROGERS, L. I. M. e Y. Citation counting, citation ranking, and h-index of human-computer interaction researchers: A comparison of scopus and web of science. JASIST, 2008. Citado na página [6].

SALLAI, J.; MARóTI, M.; LéDECZI, A. A concurrency abstraction for reliable sensor network applications. Springer, 2005. Citado (6) vezes nas páginas [9, 11, 12, 13, 15 e 17].

SEO, Y. et al. Supporting preemptive multithreading in the arx real-time operating system. IEEE, 1999. Citado (7) vezes nas páginas [5, 9, 11, 12, 14, 15 e 16].

SHIVSHANKAR, S.; VANGARA, S.; DEAN, A. G. Balancing register pressure and context-switching delays in asti systems. ACM, 2005. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 15 e 17].

SHORT, M.; PONT, M. J.; FANG, J. Exploring the impact of task preemption on dependability in time-triggered embedded systems: a pilot study. IEEE, 2008. Citado na página [4].

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating Systems Concepts*. [S.l.]: Wiley, 2021. Citado na página [4].

STEINHORN, J. Compiling java: Java compilers for embedded systems. Embedded Systems Programming, 1998. Citado na página [10].

SUNG, M. et al. Comparative performance evaluation of java threads for embedded applications: Linux thread vs. green thread. Elsevier Science B.V., 2001. Citado (5) vezes nas páginas [9, 11, 12, 14 e 16].

SUTTER, H. The free lunch is over. 2005. Citado (2) vezes nas páginas [3 e 4].

TOKIO. *Module tokio::task*. 2023. Disponível em: <https://docs.rs/tokio/latest/tokio/task/>. Citado na página [5].

VECCHIE, E.; TALPIN, J.-P.; SCHNEIDER, K. Separate compilation and execution of imperative synchronous modules. IEEE, 2009. Citado (7) vezes nas páginas [9, 11, 12, 13, 14, 16 e 17].

WADLER, P. Comprehending monads. Cambridge, 2009. Citado na página [15].

ZHAO, S.; WELLINGS, A.; KORSHOLM, S. E. Supporting multiprocessors in the icecap safety-critical java run-time environment. ACM, 2015. Citado (6) vezes nas páginas [9, 11, 12, 14, 15 e 16].

ZHAO, T.; BERGER, A.; LI, Y. Asynchronous monad for reactive iot programming. ACM SIGPLAN, 2020. Citado (6) vezes nas páginas [9, 11, 12, 13, 15 e 16].