CSP at the Cyber-Physical Edge

Lukasz Sergiusz MICHALIK [1], Michael J. MURPHY, John Markus BJØRNDALEN, Otto J. ANSHUS

UiT The Arctic University of Norway

Abstract. Today, to do ground-based in-situ observations of the arctic tundra, researchers carry wild life cameras and other observation units into the field, manually configure the devices while on the arctic tundra, and fetch the collected data several months later. This approach does not scale. Instead, observing and reporting of data must be automated using a distributed wireless network of autonomous observation units.

We present the basic hardware and software architectures of the Distributed Arctic Observatory (DAO) observation units. A DAO observation unit is composed of both heavy and light computer cores. The idea is to use the heavy cores for resource demanding tasks and then power them off and shift the workload to light cores when possible in order to increase energy efficiency and extend battery life.

We report on initial thoughts and experiences in applying a CSP network on an observation unit to ease development of advanced functionalities, while still achieving energy efficiency for observation units.

In order to rapidly develop prototype systems and learn from them, we have composed observation units with a combination of Raspberry Pi computers as the heavy cores, and Arduino, Nucleo, and Sleepy Pi microcontrollers as the light cores.

This is work in progress.

Keywords. CSP, Go, microcontroller, concurrency, energy efficiency, heterogeneous computing

Introduction

The arctic tundra is the biome that is most exposed to and sensitive to climate change. This implies an urgent need for reliable and timely observation as stated in the Climate-Ecological Observatory for Arctic Tundra (COAT) science plan [1]. Today, however, much less than 1 % of the arctic is being observed by in-situ observation stations collecting data from various sensors. While satellites can provide valuable information, ground-based in-situ observations are essential for measurements of the type and resolution needed.

To scale with the number of observation units and the volume of data, the deployment, observing, reporting and analytics of data must be automated. Such automation can be achieved with a ground-based distributed wireless network of autonomous observation units. Wireless sensor networks (WSN) typically have nodes comprised of a microcontroller, a radio, and a battery. While microcontroller-based observation units have very good energy-efficiency, they typically lack in other respects. The microcontrollers have few and limited computing, memory, and network resources. Consequently, only relatively simple functionalities can execute at the edge sensor nodes. The programming and run-time environments are also limited, lacking the tools of the trade we are used to on larger computers. This is costly in several ways, including having to use limited software models (say, not being able

[1]Corresponding Author: *Lukasz Sergiusz Michalik*. E-mail: `lukasz.s.michalik@uit.no`

to use multi-threading, though efforts are underway [2]), as well as increased development time and more bugs.

Consequently, it is harder to achieve autonomous operation in the face of harsh environmental conditions, network partitioning and limited quality of service, and unexpected failures. It also becomes more difficult to update and reconfigure observation units after deployment.

This is critical in an environment like the arctic tundra where energy and data networks are sparse resources, the frigid temperatures degrade battery and electronics operation, and observation units cannot realistically be visited by humans for updates, repair and maintenance when they fail or observation needs change.

Advanced observation units suited for the arctic tundra must therefore support advanced and varied types of networking, energy management, data processing, storage and retrieval of data, and ease of development and programming. They must also be adaptable with regards to future needs and flexible with regards to interactive changes during ongoing observations. Flexibility is especially important as we have experienced that ecologists regularly come up with new sensor applications and use cases. Agile adaptation to new sensor types and uses of sensors will be increasingly important.

Consequently, the wireless sensor network systems must shift from being microcontroller-based to *also* being computer-based. To achieve the needed energy-efficiency and low energy usage, heterogeneous architectures such as ARM's big.LITTLE architecture [3] provide heavy cores for the resource and energy demanding tasks, and light cores for lighter tasks drawing very little energy. Such systems will turn cores and other resources on and off as demanded by the available energy and mission tasks to be performed.

ARM's big.LITTLE architecture has hardware advantages that make programming simpler. Both big and little cores have the same architecture and instruction set, and share memory through a hardware cache-coherence layer [4]. This hardware support allows the operating system scheduler to leverage classic symmetric multiprocessing (SMP) techniques to move threads between heterogeneous cores. Kernel scheduling code is modified to add heuristics that schedule threads between big and little cores based on thread load statistics. In another mode, hardware virtualization presents the big and little cores together as one core with two clock speeds, stopping processes and transparently moving them between cores when the OS requests a clock-speed change. This allows the operating system's existing clock-speed regulating code to make use of the heterogeneous architecture with no modification at all, though it only uses half of the processors at any given time. In either case, no modifications are required to application code, beyond normal use of multiprocessing.

Programming a microcontroller-based sensor node includes flashing the microcontroller with a sequential program, running applications on top of or embedded inside a tiny operating system, and providing a sensor node with a simple interface for interacting with remote applications running on remote servers or in the cloud. Sometimes basic software must be installed into the sensor nodes before they are deployed.

We have built multiple prototype sensor nodes, which we call observation units, following the idea of mixing heavy and light cores. We apply a well-known commodity computer (Raspberry Pi), and one or multiple well-known microcontrollers (Arduino). In normal operation, the Raspberry Pi will boot into Linux and run necessary mission software for a short duration before shutting down to save energy. The microcontroller is always on in low-powered mode to perform light tasks, and it will decide when to power the Raspberry Pi on or off. Power to the Raspberry is controlled by a Sleepy Pi, which is itself a microcontroller board that is preconfigured to control power to a Raspberry Pi. The Sleepy Pi is necessary because a Raspberry Pi is not actually capable of powering itself down fully. Figures 2b and 3 show this prototype's architecture and implementation.

The advantage of such a node for development is that the "heavy core" is running a mature and widely used operating system (Raspbian, which is based on Debian Linux) that has all necessary tools of the trade. Consequently, we are able to use the programming models and languages that we are used to and prefer using (such as CSP in Python or Go).

In this paper, we are assuming that we can write a normal PyCSP or Go program, and seamlessly deploy the processes/co-routines between the heavy and light cores. The idea is similar to OpenMP [5], where the same source code is used to write and verify a sequential program and, through annotations, compile a parallel program. A similar approach to sensor system development would let us write a CSP program and identify processes that will execute on, for instance, a microcontroller. While we ignore discussing how to do this in practise, we briefly look at how Linux and Android approaches using multiple cores. We focus on a few of the issues arising when cores, at any time, can enter sleep and power off modes for shorter and longer periods. We assume this to be the common case.

The original CSP paper from 1978 states that "the programs expressed in the proposed language are intended to be implementable both by a conventional machine with a single main store, and by a fixed network of processors connected by input/output channels (although very different optimizations are appropriate in the different cases)." [6]. When cores running CSP processes or goroutines enter sleep mode or power off, we are interested in the behaviour of the parallel, input, output, alternative and repetitive commands. We identify some of the issues we believe are of interest, and experiment with Go running on a Raspberry Pi and a Nucleo microcontroller in order to gain some practical experience with the challenges arising when cores sleep and power on and off.

The work flow we envision to develop concurrent functionalities for observation units is comprised of five stages.

Stage 1 is to design and implement a CSP/Go program for a single computer (laptop, PC with Linux).

Stage 2 is to manually move the whole program from the PC to one of the heavy cores (a Raspberry Pi) of an observation unit for early deployment and testing with sensors.

Stage 3 is to manually move a process/goroutine of the program from the heavy core (Raspberry) to a light core (Nucleo microcontroller) for initial power optimisation.

Stage 4 is to find a mapping of processes/goroutines to heavy and light cores and actually do the migration to the cores. All cores will be powered on all the time.

Stage 5 is to find a mapping of processes/goroutines to heavy and light cores and actually do the migration to the cores. Cores can be in powered on, powered off, and sleep states.

Instead of migrating processes/goroutines between cores, we are also considering keeping a program intact on a single core, and instead migrate whole programs between cores.

1. CSP and Heterogeneous Architectures

A **parallel command** is specified in [6] as a "concurrent execution of its constituent sequential commands (processes). All the processes start simultaneously, and the parallel command ends only when they are all finished."

With a heterogeneous architecture, the challenges arising from the parallel command executed across cores include simultaneous starting, scheduling, power management and simultaneous ending.

The main issue with "start all simultaneously" is that some of the cores on which processes are meant to run are sleeping or powered off. The command could use already powered cores with resources to accept the process and its workload. However, for the application domain we are interested in, advanced sensor nodes on the arctic tundra, energy is a sparse resource to be carefully used. Consequently, heavy cores will normally be powered off, and

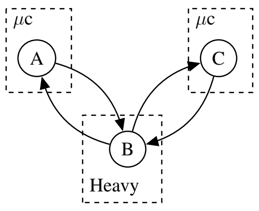


Figure 1. Hypothetical distribution of CSP processes across devices

must be powered on before assigning processes to them. Light cores on the other hand will as common case be powered on or can be quickly powered on from various power levels, but may have too few resources to accept many of the processes. Taken together, this has implications for how many cores can be used for “simultaneous” start of processes. If cores have to be powered on before being used, the latency of starting processes increases. Furthermore, it may be beneficial to consider co-scheduling some of the processes to reduce the number of wake-ups and shutdowns of the heavy cores.

We envision that the parallel command should allow assignment of processes to powered off or suspended cores. In effect, this would be a pending migration until the cores are powered on for some reason. One approach to handle this would be to add “power guards” and “scheduling guards” that would allow the programmer to provide hints about acceptable delays and scheduling decisions for the runtime.

To “end the command when they are all finished,” the system can force cores to remain powered on until all processes are finished. This increases the energy usage. Some processes may even never “finish” as such, and all cores used by the parallel command remain powered on all the time. We need to let processes reside on cores that are sleeping or powered off, without this being seen as a crash, failure or termination of the processes. One question here is whether we can use light versions of processes that can run on light cores as representatives, or proxies, for heavy processes running on sleeping or powered off heavy cores.

An **input and output command** “is delayed until the other process is ready with the corresponding output or input. Such delay is invisible to the delayed process [...] An input command fails if its source is terminated. An output command fails if its destination is terminated [...]” [6].

With a heterogeneous architecture, the challenges arising from input and output commands include delays and failing. Delays can arise from, for instance, a core running a process being suspended after the process starts to send or receive on a channel. The corresponding operation may not be established or completed before the suspended core becomes active, at a cost in latency and energy.

A process should not be detected as failed unless it fails to wake up again. One of the issues here is separating failed communication channels or units in a system from processes located on a suspended core.

The few issues raised here impact other commands like input and output guards and the alternative and repetitive commands. To evaluate input and output guards, sleeping and powered off cores have to be powered on and awaken on a potentially large scale. This can be costly in energy and in increasing the delays before a decision is made on which alternative to execute.

For the repetitive command with input guards, it is meant to terminate if all the sources named by them have terminated. Again, “termination” has to be defined not to imply that a process on a sleeping or powered off core has terminated.

As an example case, consider the process network in fig. 1. A program that runs correctly on a single desktop computer could be mapped to run process A on one microcontroller, process C on another and a coordinator, B, on a heavy core[1]. If A tries to coordinate with C through B, it can do this with messages passed on channels. If the core running B is suspended anywhere in this process, we may end up with A and C having an inconsistent view of the result as B may be able to complete the communication with A just before being suspended, but not have the time to coordinate with C.

In effect, we have created a problem similar to the two-generals problem [7]. A practical solution to this might involve some form of notifications to let the runtime and power management system take application intentions into account and also notify peers and the user programs about actions. This might require modifications to the runtime.

Allowing coordinating entities, termed narcoleptic processes in our group, to suspend and resume at any point in time introduces issues similar to partitioning in distribute systems, except some of the processes will be sleeping instead of disconnected. Forcing cores to stay awake during co-ordinations can quickly lead to problems similar to priority inversions, where a long-running channel communication keeps the entire system running. We are currently using the term sleep deprivation internally for this situation. One potential solution to this is to model some process interactions as short-lived transactions with deadlines.

2. Related Work

Prior relevant state of the art on using cyber-physical systems includes research on wireless sensor networks (WSN), deployment and use of WSNs in various fields and application domains like agriculture, habitat and environmental monitoring [8–15].

A WSN typically is composed of small computing devices being microcontrollers (often called motes) [16, 17, 17–22]

For the sensor nodes the WSN community traditionally applied MICA2 motes [16–19], the Sun SPOT platform [20], Libelium Wasp-motes [21], and custom-made architectures [17, 22].

From about the 2010 this shifted gradually towards Arduino-based architectures, such as Arduino Uno [23] and Arduino Mega [24]. AVR ATmega 1281V is also in use [25]. All of these approaches use microcontrollers with very limited resources, but being very energy-efficient vs. general purpose tiny computers like the Raspberry Pi.

Of special interest for this paper is the hardware and software platforms used for development of software for sensor nodes, and platforms running on the nodes. While energy-efficient and often able to communicate via radio, a sensor node typically has very limited processing and memory resources. The reported experience is that such platforms are difficult and time-consuming to write software for [26–29].

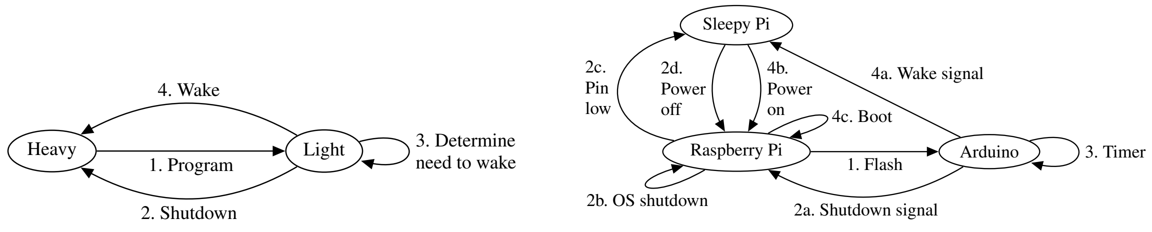
To determine the suitability for prototyping, Occam, Go, and Python/PyCSP have been compared [30].

The conclusion is that Go provides both ease of development, results in stable programs, allows for low level hardware interfacing, and CSP-style of concurrency and communication.

We are aware of proposals to introduce operators to support asynchronous input and output commands. One proposal supports having a process doing output to only processes being able to do an input command [31]. We find this proposal interesting because we see it as relevant to our question of what to do when deciding which processes to communicate with when some of the possible processes are on sleeping or powered down cores.

A second proposal allows the acknowledgement from the receiver to the sender to be delayed [32]. This is interesting for our research. If a receiver delays sending ack, the sender must accept a delay. We would like to see if this can be used to let the delay be because of a sleeping/powered off core.

[1]The program may, for instance, need to run some computation in B to determine the next state



(a) Conceptual Architecture (b) Prototype Design

Figure 2. DAO observation unit: architecture and design

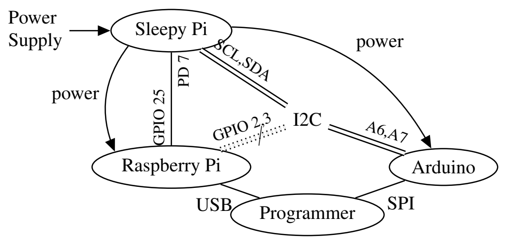


Figure 3. DAO observation unit: hardware connections

A third proposal allows a process to do an output command without having a corresponding input command ready [33]. This proposal directly attacks the issue on whether an alternative command can select a process on a sleeping/powered off node to interact with.

One of the options we are considering is to compile existing languages down to an efficient interpreter that can be deployed on target micro-controllers. MicroPython[2] is an example of a runtime for the Python programming language, but it has limited concurrency support directly, and the asyncio and threading libraries are very limited currently. The Transterpreter, on the other hand, is a system that compiles Occam-π down to run on an interpreter which can run on small embedded architectures [34]. In [35], it was ported to the Cell processor, which is an example of a distributed hybrid processor architecture.

3. Architecture

The idea of the present version of a DAO observation unit is to apply heavy cores for tasks demanding more processing, memory, and other resources, and to apply light cores for less demanding tasks. fig. 2a shows a diagram of the prototype architecture.

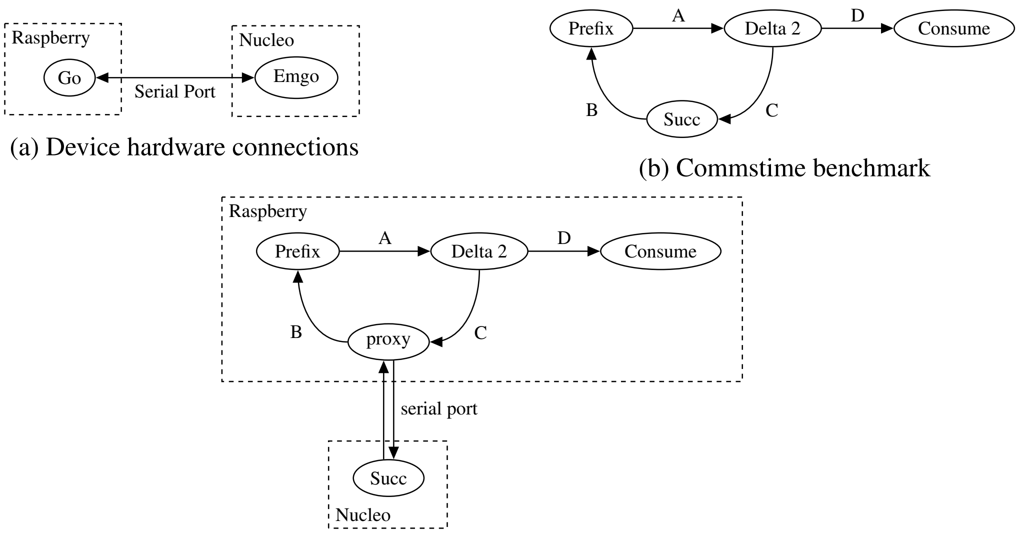
Technically, we combine a well known and widely used tiny computer and operating system with a microcontroller. The tiny computer will handle more resource demanding tasks, and the microcontroller will handle lighter tasks. The microcontroller will also coordinate to switch the computer on and off according to the observation unit's workload.

The software architecture for the heavy core comprises the functionalities: (i) customizing ("programming") light core functionalities as needed, (ii) shutting down and rebooting itself, and (iii) other heavy-duty mission business logic & payloads.

The software architecture for the light core comprises the functionalities: (i) determining when to wake up heavy core, (ii) determining when to request shutdown of heavy core, (iii) keeping time, and (iv) other light-duty mission business logic & payloads.

The prototype design in fig. 2a shows a prototype that we have developed to investigate suspending and resuming the heavy cores, represented by the Raspberry Pi. The Sleepy Pi is necessary as a controller that can properly cut power to the Raspberry Pi because the Raspberry lacks the capability to power itself down fully.

[2]MicroPython, see https://micropython.org/



(a) Device hardware connections

(b) Commstime benchmark

(c) Commstime benchmark across devices

Figure 4. Prototype implementation of commstime benchmark across devices

For this paper, we have started working on a prototype that is capable of running a concurrent program written in the Go programming language on the microcontroller, using the Emgo compiler and runtime.[3] Emgo uses the standard Go compiler, but it targets a custom Go runtime that is developed specifically to run on microcontrollers. Emgo does not currently support Arduino hardware, so we have modified our prototype to instead use a Nucleo microcontroller, which Emgo does support. In addition, we have temporarily removed the Sleepy Pi in order to simplify development and focus on the problem of moving goroutines to a microcontroller. This simplified architecture is shown in fig. 4a. This simplified prototype is not intended for extreme temperatures, and is instead focused on the integration of Go programs running on heavy cores and microcontrollers.

4. Implementation

The prototype was implemented using a Rasperry Pi 3B connected to a Nucleo-L476RG microcontroller as shown in fig. 4a. The Raspberry Pi is running programs written in Go, while the Nucleo board runs programs compiled with Emgo. To simplify the prototype, we are currently connecting the Nucleo board to the Raspberry Pi over USB. The USB connection powers the Nucleo board and is used for uploading new programs to the Nucleo board. We are also using the serial port console for communication between the Go program and the Emgo program.

Future implementations will investigate other communication options, such as i2c, as powering the Nucleo from one of the Raspberry's USB ports prevents us from powering down the Raspberry Pi without cutting power to the Nucleo.

To experiment with communication with the Emgo compiled programs on the Nucleo, we have implemented a variation of the commstime benchmark [36]. The benchmark is used to measure context switch and channel communication overhead. The structure of a normal commstime benchmark running on a host computer is shown in fig. 4b.

In commstime, a value is passed around in a cycle, with copies of the values observed by Consume. The Prefix process initiates the execution of the benchmark by sending a value to Delta2 over channel A. Delta2 receives the value and sends a copy over both D and C.

[3]Emgo, see https://github.com/ziutek/emgo

Succ receives a value, adds 1 to it and sends the result over channel B. From now on, Prefix receives on B and sends the received value on A without modification.

Consume first receives a few values over D to make sure all the other processes are running. It then records a timestamp and reads a predefined number of messages before recording a new timestamp. The elapsed time is then used to compute an average time for a channel communication and scheduling of goroutines.

The plan for the prototype was originally to move the Succ process out to the Nucleo board and run the rest of the benchmark on the Raspberry Pi. We did not get remote channel support working in time for this paper, so the prototype now uses a Succ proxy process running on the Raspberry Pi that receives a value from the C channel, passes it over the serial port and reads a new value back over the serial port before sending it over the B channel. The Succ proxy is shown below. It is running on the Raspberry, as shown in fig. 4c.

```
func succ_proxy(in <-chan int, out chan<- int) {
    i := 0
    for {
        n := <-in;
        WriteSerial(i, n)
        n = ReadSerial()
        out <- n
        i++
    }
}
```

The Emgo side runs a version of the Succ process that reads on the serial port, increases the value by 1 and passes the result back over the serial port.

5. Experiments

A group of performance measuring experiments were done on a prototype to document performance characteristics.

We measured electrical current draw during the commstime benchmark (fig. 4b) when running all together on a Raspberry Pi 3, and then when split across a Raspberry Pi and Nucleo board communicating over a serial port (figs. 4a and 4c). During separate runs of these two benchmarks, we have also captured CPU utilization metrics on the Raspberry Pi (user and system times).

In order to determine how much overhead is imposed by serial communication between devices, we wrote an additional benchmark in which Raspberry Pi sends and receives exactly $3\,\mathrm{B}$ of data ($1\,\mathrm{B}$ of actual data and $2\,\mathrm{B}$ end-of-transmission) to and from Nucleo board. This send and receive is repeated 100 times.

To measure power consumption, we connected a digital multimeter (R&S® HMC8012) between the Raspberry Pi and its power supply. We measured electrical current draw and then used it to calculate power consumption, using a measured voltage that was consistently very close to $5\,\mathrm{V}$. To obtain readings over time, we used the multimeter's digital log function to record samples every second (capturing about $5\,\mathrm{samples/s}$).

To obtain CPU utilization readings we used the `sar` command line tool from the `sysstat` package (System performance tools for the Linux operating system). The `sar` command was executed in a separate Raspberry Pi SSH session in order to sample system activity every $1\,\mathrm{s}$ during the commstime benchmarks. The `sar` command calculates its CPU utilization across all cores, so a report of $25\,\%$ CPU utilization corresponds to $100\,\%$ utilization of one of the Raspberry's four cores.

The commstime benchmark was executed with the default Go runtime parameters (GOMAXPROCS=4 for RaspberryPi 3).

In order to capture CPU utilization and current draw during the commstime benchmark executed on Raspberry Pi alone, we increased number of iterations for the consume process from 100 to 100000. This was necessary to increase the execution time of the benchmark to a time scale large enough for the multimeter and CPU utilization sampler to get steady readings.

6. Results

Figure 5 shows context switch times and times taken by a single round of data exchange between Go processes for the commstime benchmark executed on the Raspberry Pi alone and on the combination Raspberry plus Nucleo. It is clear from the presented results that our benchmark needs a few magnitudes less time to finish when GO routines are communicating on the same node (note that the time axis is on a logarithmic scale). Part of the slower times for the Raspberry-Nucleo combination is caused by communication delay. One of the Go processes ("proxy" in fig. 4c) is communicating with the Emgo process on the Nucleo side using a serial port, which imposes and additional delay of $1020\,\mu s$ between each read and write operation.

The slowest benchmarks in the figure are from an early implementation of the serial communication code, where each read from and write to the serial port was imposing a $20\,ms$ delay on the Raspberry Pi side. This from our read implementation waiting for at least four bytes to arrive before declaring the read successful. We changed the implementation to detect single bytes arriving on the serial line and eliminated that delay, resulting in the improved benchmarks show on the right in the figure. In order to improve context switch time further, we will need to find a more efficient way to exchange data between the Raspberry and the Nucleo.

The Raspberry Pi draws an average $264\,mA$ of current when idle and around $370\,mA$ while executing the more intensive variant of the commstime benchmark alone (one thousand times more data exchanges between GO processes). This experiment used between $25\,\%$ to $49\,\%$ of the Raspberry's CPU time during execution. A single run of benchmark takes approximately $17\,s$.

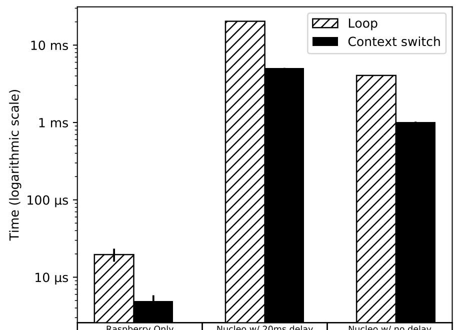
The combined Raspberry Pi and Nucleo idles at around $320\,mA$ and spikes up to $340\,mA$ during the workload. We measured that Nucleo board itself uses $44\,mA$ to $56\,mA$ of current while waiting for data arrival on a serial port. Our experiment showed that communication delay causes the Raspberry's CPU to spend most of the benchmark's time idle (`sar` shows only $0.25\,\%$ to $0.5\,\%$ CPU utilization). Electric current consumption results are depicted in figs. 6a and 6b.

Current consumption results for the Raspberry-Nucleo combination are not terribly impressive considering the applied workload. In order to improve our prototype we are planning to decrease the current draw of Nucleo board below $7\,mA$ in the next iteration by underclocking its CPU and disabling unused board components. Future experiments will also power down the Raspberry Pi to conserve energy, delegating more work to the microcontrollers.

7. Discussion

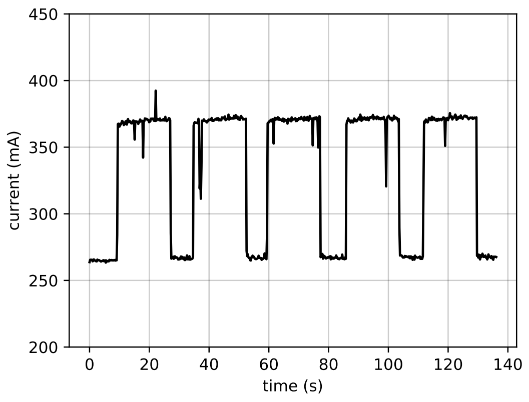
With heavy and light cores potentially entering powered off/on and sleep states at any time, we have characteristics of an asynchronous distributed system with unreliable, untrusted communication channels.

The implementation of CSP commands will have to send acknowledgements when implementing the channel abstraction.

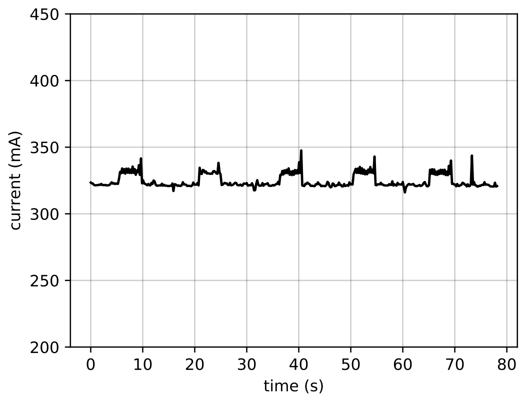


	Raspberry Only	Nucleo w/ 20ms delay	Nucleo w/ no delay
Loop	19.6 µs	20.3 ms	4.1 ms
Context switch	4.9 µs	5.1 ms	1.0 ms

Figure 5. Commstime benchmark results



(a) Running commstime on Raspberry only



(b) Running commstime across Raspberry and Nucleo

Figure 6. Current consumption during commstime benchmark

Depending on how the implementation is done, this can produce situations where an output command done at one core correspond to an input command at another core, while the implementation of the channel in effect is "hosted" at a third core. The sender and receiver can reach an inconsistent state with regards to the state of the output and input because the acknowledgements may or may not have reach both. This can happen because the third core can enter powered off or sleep state between sending out ACKs.

In effect we have a problem like the two-generals problem [7].

8. Conclusions

Advanced sensor nodes for harsh, resource limited, and hard to get at environments need to shift from being microcontroller based to having significant more resources available. An architecture with heavy and light cores provides for both ease of development, rich functionalities and energy efficiency.

Programmability is critical to support deployment of a system growing over time, and having to adapt to future technologies and observational needs.

We are interested in researching if a CSP network of processes can support all of the above requirements and needs. Concurrent programs or processes/goroutines will have to be mapped to cores, and then migrate between cores according to a range of dimensions including resource availability (energy, CPU, memory, network), and observational needs.

The research is in an early stage, and we are presently focusing on identifying the challenges, and constructing an early prototype.

Acknowledgements

The DAO project is supported by the RCN IKTPluss program, project number 270672. The COAT Tools project is supported by the UiT The Arctic University of Norway. Thank you very much to the COAT ecologists, and to Ken-Arne Jensen at the technical staff at the UiT Dept. of Computer Science.

References

[1] Rolf A. Ims, Jane U. Jepsen, Audun Stien, and Nigel G. Yoccoz, editors. *Science plan for COAT: Climate-ecological Observatory for Arctic Tundra*. The Fram Centre by the University of Tromsø, 2013.

[2] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 234–251, New York, NY, USA, 2017. ACM.

[3] Arm big.LITTLE technology.

[4] *ARM Cortex-A Series Programmer's Guide for ARMv8-A (DEN0024A)*. ARM, 2015.

[5] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[7] E. A. Akkoyunlu, K. Ekanadham, and R. V. Huber. Some constraints and tradeoffs in the design of network communications. *SIGOPS Oper. Syst. Rev.*, 9(5):67–74, November 1975.

[8] Aline Baggio. Wireless sensor networks in precision agriculture. In *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), Stockholm, Sweden*, pages 1567–1576. Citeseer, 2005.

[9] Antonio-Javier Garcia-Sanchez, Felipe Garcia-Sanchez, and Joan Garcia-Haro. Wireless sensor network deployment for integrating video-surveillance and data-monitoring in precision agriculture over distributed crops. *Computers and Electronics in Agriculture*, 75(2):288 – 303, 2011.

[10] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 8 pp.–, April 2006.

[11] W.S. Lee, V. Alchanatis, C. Yang, M. Hirafuji, D. Moshou, and C. Li. Sensing technologies for precision specialty crop production. *Computers and Electronics in Agriculture*, 74(1):2 – 33, 2010.

[12] Tamoghna Ojha, Sudip Misra, and Narendra Singh Raghuwanshi. Wireless sensor networks for agriculture: The state-of-the-art in practice and future challenges. *Computers and Electronics in Agriculture*, 118:66 – 84, 2015.

[13] Luis Ruiz-Garcia, Loredana Lunadei, Pilar Barreiro, and Ignacio Robla. A review of wireless sensor technologies and applications in agriculture and food industry: State of the art and current trends. *Sensors*, 9(6):4728–4750, 2009.

[14] T. Wark, P. Corke, P. Sikka, L. Klingbeil, Y. Guo, C. Crossman, P. Valencia, D. Swain, and G. Bishop-Hurley. Transforming agriculture through pervasive wireless sensor networks. *IEEE Pervasive Computing*, 6(2):50–57, April 2007.

[15] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[16] Lindsay A Seders, Caitlyn A Shea, Michael D Lemmon, Patricia A Maurice, and Jeffrey W Talley. Lakenet: an integrated sensor network for environmental sensing in lakes. *Environmental Engineering Science*, 24(2):183–191, 2007.

[17] Xin Wang, Longquan Ma, and Huizhong Yang. Online water monitoring system based on zigbee and gprs. *Procedia Engineering*, 15:2680 – 2684, 2011. CEIS 2011.

[18] Teresa Ko, Shaun Ahmadian, John Hicks, Mohammad Rahimi, Deborah Estrin, Stefano Soatto, Sharon Coe, and Michael P. Hamilton. Heartbeat of a nest: Using imagers as biological sensors. *ACM Trans. Sen. Netw.*, 6(3):19:1–19:31, June 2010.

[19] Liqian Luo, Qing Cao, Chengdu Huang, Lili Wang, Tarek F. Abdelzaher, John A. Stankovic, and Michael Ward. Design, implementation, and evaluation of enviromic: A storage-centric audio sensor network. *ACM Trans. Sen. Netw.*, 5(3):22:1–22:35, June 2009.

[20] M. Zennaro, A. Floros, G. Dogan, T. Sun, Z. Cao, C. Huang, M. Bahader, H. Ntareme, and A. Bagula. On the design of a water quality wireless sensor network (wqwsn): An application to water quality monitoring in malawi. In *2009 International Conference on Parallel Processing Workshops*, pages 330–336, Sept 2009.

[21] Nikolaos Rapousis, Michalis Katsarakis, and Maria Papadopouli. Qowater: A crowd-sourcing approach for assessing the water quality. In *Proceedings of the 1st ACM International Workshop on Cyber-Physical Systems for Smart Water Networks*, CySWater'15, pages 11:1–11:6, New York, NY, USA, 2015. ACM.

[22] Peng Jiang, Hongbo Xia, Zhiye He, and Zheming Wang. Design of a water environment monitoring system based on wireless sensor networks. *Sensors*, 9(8):6411–6434, 2009.

[23] Cesar Eduardo Hernández Curiel, Victor Hugo Benítez Baltazar, and Jesús Horacio Pacheco Ramírez. Wireless sensor networks for water quality monitoring: Prototype design. *International Journal of Environmental, Chemical, Ecological, Geological and Geophysical Engineering*, 10(2):162 – 167, 2016.

[24] A. S. Rao, S. Marshall, J. Gubbi, M. Palaniswami, R. Sinnott, and V. Pettigrovet. Design of low-cost autonomous water quality monitoring system. In *2013 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 14–19, Aug 2013.

[25] Vladimir Dyo, Stephen A. Ellwood, David W. Macdonald, Andrew Markham, Niki Trigoni, Ricklef Wohlers, Cecilia Mascolo, Bence Pásztor, Salvatore Scellato, and Kharsim Yousef. Wildsensing: Design and deployment of a sustainable sensor network for wildlife monitoring. *ACM Trans. Sen. Netw.*, 8(4):29:1–29:33, September 2012.

[26] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 159–172, New York, NY, USA, 2007. ACM.

[27] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 200–210, New York, NY, USA, 2007. ACM.

[28] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[29] Adriano Branco, Francisco Sant'anna, Roberto Ierusalimschy, Noemi Rodriguez, and Silvana Rossetto. Terra: Flexibility and safety in wireless sensor networks. *ACM Trans. Sen. Netw.*, 11(4):59:1–59:27, September 2015.

[30] Brian Vinter, Lawrence J. Dickson, Lindsay O'Brien Quarrie, Patrick Dyhrberg Sørensen, and Tor Skovsgaard. "a concurrent data collection environment for wasteful communication satellite system". In Jan Bækgaard Pedersen, Kevin Chalmers, Jan F. Broenink, Brian Vinter, Kevin Vella, and Peter H. Welch, editors, *"Communicating Process Architectures 2017"*, pages 187 – 196. "Open Channel Publishing Ltd., Bicester, UK", "August" 2017.

[31] S. Gruner, D. G. Kourie, M. Roggenbach, T. Strauss, and B. W. Watson. A new csp operator for optional parallelism. In *2008 International Conference on Computer Science and Software Engineering*, volume 2, pages 788–791, Dec 2008.

[32] Peter Marwedel. Embedded system foundations of cyber-physical systems, 2011.

[33] Jan F. Broenink and Antoon H. Boode. "asynchronous readers and writers". In Kevin Chalmers, Jan Bækgaard Pedersen, Jan F. Broenink, Brian Vinter, and Peter H. Welch, editors, *"Communicating Process Architectures 2016"*, pages 197 – 210. "Open Channel Publishing Ltd., Bicester, UK", "August" 2016.

[34] Christian L. Jacobsen and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In Ian R. East, David Duce, Mark Green, Jeremy M. R. Martin, and Peter H. Welch, editors, *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering Series*, pages 99–106, Amsterdam, September 2004. IOS Press.

[35] Damian J. Dimmich, Christian L. Jacobsen, and Matthew C. Jadud. A Cell Transterpreter. In Peter Welch, Jon Kerridge, and Fred Barnes, editors, *Communicating Process Architectures 2006*, volume 29 of *Concurrent Systems Engineering Series*, pages 215–224, Amsterdam, September 2006. IOS Press.

[36] Fred Barnes and Peter H. Welch. Prioritised Dynamic Communicating Processes - Part I. In James Pascoe, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, pages 321–352, sep 2002.