A Low-Power Low-Cost Implementation of Multi-Join on Raspberry Pi Pico

Giles Drake[1], Howard Goodsall[1], Albertus Alvin Janitra[1], Alistair Jensen[1], Benjamin Lilley[1], Matthew McHale[1], M.Reza HoseinyFarahabady[1], and Albert Y. Zomaya[1]

[1] The University of Sydney, School of Computer Science,
Center for Distributed and High Performance Computing, NSW, Australia
{gdra3694,hgoo9099,ajan8924,ajen6045,blil9516,mmch8995}@uni.sydney.edu.au
{reza.hoseiny,albert.zomaya}@sydney.edu.au

Abstract. An emerging problem in the field of data computation is the rising total power consumption of data servers. Traditional data servers typically use between 500 and 1200 Watts of electricity. Low-power low cost micro-controllers such as the recent Raspberry Pi Pico have the potential to replace these large data servers whilst only consuming a fraction of the electricity. The Raspberry Pi Pico consumes only 0.3 Watts but only has 264 Kilobytes of RAM and 2 Megabytes of flash storage for a cost of roughly $4 USD. In this work we demonstrate an efficient join algorithm that is performed using both a computer and Raspberry Pi Pico to establish the capability of micro-controllers in performing typical data computation operations. We also compare this with an implementation in MySQL to simulate a typical data server's operation of the same algorithm. This work aims to prove that such operations can be done using micro-controllers, allowing for future work into using multiple micro-controllers cohesively to match the same operational power of a traditional data server with only a fraction of the electricity consumption.

Keywords: Data intensive computation · MultiJoin algorithm · Low-power low-cost Computing · Prototyping platforms

1 Introduction

Processing and analysis of large scale data sets, either in batch or streaming modes, requires massively parallel software based operations running over clusters with thousands of server nodes. The significant growth of processing power required to perform such operations introduces several new challenges to be overcome [1]. In recent years, the design of green data-centers have gained a lot of attraction in both industry and academia sectors, mainly due to the rapid increasing level of power consumption of servers in conventional data-centers [2]. Data centres represent a comparatively large percentage of all countries' total power consumption, accounting for up to 4 per cent of Australia's total energy

consumption and approximately 10 percent of the world's energy [3–5], which present a great challenge to optimal design of green data-centers.

In this paper, we investigate the possibility of utilising micro-controller devices for large-volume data computation in a set of low-cost low-power devices. Due to their competitive *price* to *performance* ratio and significantly less power consumption (in particular when compared to the conventional server nodes), we design and implement a software architecture that can replace the traditional method of using energy-hungry processing servers with a low-power low-cost micro-controller device for performing large complex data operations [6]. Such a solution is explored by developing an implementation of a complex Multi-join algorithm to be executed on a micro-controller device.

The micro-controller chosen to be used in this study is the `Raspberry Pi Pico` – an emerging micro-controller board with an impressive processor but quite limited memory. The implementation of Multi-join algorithm was based on research originally proposed by authors in [7], and aims to serve as a useful benchmark for development of relational database communication with the Pico device. This primary research should also serve as a foundation for improved power efficiency in database operations, due to the Raspberry Pi Pico's relatively low power consumption of 0.3 watts in conjunction with the recent developments in Join Algorithm complexity [8–10].

Join is a common operation found in data management systems. These include, but are not limited to *databases*, *stream processing systems* and *rule based engines*. The most common join algorithms is the *hash-join* [11]. In this paper, we focus on a specific type of join known as multi-join. The implementation of multi-join algorithm is deemed to be worst case optimal as shown by authors in [7]. Due to the importance of the join function, this paper investigates the performance and resource utilisation of multi-join operations implemented and running on the Pico device. Moreover, a similar multi-join algorithm was executed on data in MySQL as a means of benchmarking to compare this implementation to an unbiased measure. This was done to give an indication of the relative performance and efficiency of the newly developed approach. Since the Raspberry Pi Pico only supports C, C++ and Python [12], the core of the developed implementation is written in C++, with some pre-processing done in Python. Other accompanying scripts including benchmark and algorithm testing are also written in C and Python.

1.1 Original Contribution

In this research, we develop a better understanding of importance regarding low-power low-cost design of iterative processing over data sets. This is still an emerging area built on top of a dense history that requires further evaluation in the future. At this point, there are only very few applications and researches on performing low-power low-cost processing using micro-controllers. In this paper, we apply such a solution to the multi-join algorithm and relate it to the existing implementation on other platforms (e.g., MySQL). The main contribution of the current study is to develop a serialised Multi-Join algorithm to be executed on

a Raspberry Pi Pico device. The experimental part of this research (second contribution) evaluates the performance of the proposed solution by measuring the accuracy, performance, power, and resource consumption of the developed algorithm and comparing it with the join implementation in a conventional MySQL platform. This is carried out by building a system prototype that generates multiple large scale data sets at different complexities to act as input data for the multi-join algorithm and then continuously outputs the system resource logs.

The rest of this paper is organized as follows. Section 2 highlights the main challenges associated with performing multi-join on traditional systems (such as a Relational Database Management System, RDBMS). Section 3 presents the details of our proposed scheme. The performance of the proposed solution is evaluated in Section 4. Finally, Section 5 concludes our work.

2 Problem Statement

Efficient join processing is a fundamental function of many big-data analytics tasks. With the rise of stream processing engines, there are possibilities for improving performance of various computational tasks across different industries. With data being generated as indefinite streams of events, it is valuable to process data responsively and in real-time rather than expensive operations of batching and aggregating. Recently, the proposal and implementation of iterative, high-performance systems has gained great attention from researchers. An example of such a problem is multi-join algorithm over streaming flow with high volume data streams. This is an ongoing research area, but a definitive gap exists where there is currently no insight into applying an efficient join algorithm operating in the low-power devices. In particular, there is a research gap for evaluation of system performance and performance tuning parameters regarding workload and computing resources.

In general, there are two major approaches for join algorithms. The first of these is using a structural approach, and the second is a cardinality-based approach[1]. On the theoretical side, many algorithms use some structural property of the query. On the other hand, commercial RDBMSs place little emphasis on the structural property of the query and more emphasis on the cardinality side of join processing. The approach we are going with was derived by Grohe-Marx and Atserias-Grohe-Marx (`AGM`). The `AGM` approach combines the 2 approaches we mentioned earlier.

2.1 Multi-join Algorithm

Given a set of relations, each consisting of tuples of two or more attributes, to perform a multi-join on this data, we split the processing into two stages, "propose" and "intersect". Both steps are performed once for every attribute in the data set and each time this process is performed is known as a round. If

[1] The cardinality of a set is a measure that reflects the give set's size

an attribute appears in more than one relation, propose and intersect are still only performed once for this attribute. The propose stage's role is to extract the unique data points in every relation for a particular attribute. The intersect stage then takes the common data points between all relations, similar to an inner join, and intersects the entire tuples with the results of previous iterations.

The first stage, namely propose, creates a set of the unique values in each relation for a particular attribute. If an attribute appears in more than one relation, a set of unique values will be created for each one separately. The values that are proposed by more than one relation are the values that will be accepted in the intersect step. For relations that do not contain the attribute being proposed, an asterisk "*" is used to denote that this relation is not proposing anything, but any value can take its place in the intersect step. For subsequent propose stages, the result of the previous rounds are enforced on all remaining rounds. This means that if a relation contains two attributes and is entering the second round, the propose step for the second attribute can only propose values that contain the first attributes obtained in the first round.

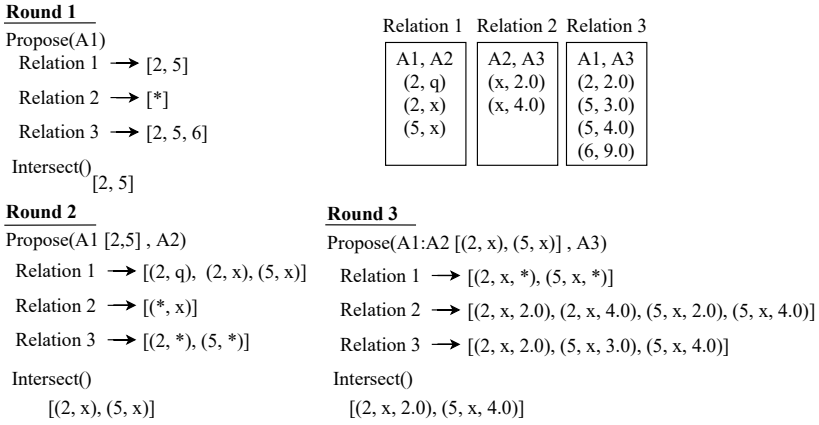

Fig. 1. An example of the Multi-join algorithm performed on a case with three relations, while each relation consists of two attributes (namely A_1, A_2, and A_3).

Moreover, if there is more than one tuple where the value for the attribute currently being operated on is the same (a repetition of the same value) but the other element(s) in the tuple are different and they belong to an attribute that has already gone through a propose and intersect stage, they will all be included in the proposed result. So, if there are values for an attribute which by themselves are not unique, but are unique as entire tuples, they will all be accepted (as long as the other element in the tuple has made it through a previous iteration of propose and intersect). The result of each round is carried over to the next iteration with the tuples in the results growing by one value each time.

The end result of the entire algorithm will be tuples of the same length as the number of attributes.

In the intersect stage, the sets for each relation from the propose stage are compared and the tuple values that are common among all sets are included in the result. As mentioned, if a relation does not contain the attribute which is being operated on, it will propose an asterisk, which represents 'any value'. This means that values only need to appear in the propose results for the relations which contain the particular attribute being operated on. The intersect stage does not only take unique values, it takes unique tuples, following the same rules as propose, where the tuples produced can only contain the attributes that were previously processed or the one being currently processed.

In Figure 1, both the "propose" and "intersect" operations are performed for each attribute ('A1', 'A2' and 'A3). In each iteration, the result of the previous is used to select the unique tuples in the propose step. The final result is the result of the final intersect, which is: [(2, x, 2.0), (5, x, 4.0)].

2.2 Raspberry Pi Pico

The Raspberry Pi Pico is a newly released micro-controller board which offers comparatively high performance when considering its low cost of $4 USD. The key hardware feature of this device is the RP2040 device which provides 2 MB of flash storage and 264 KB of SRAM. The board itself consists of 40 pins with 26 `GPIO` (general purpose input output) pins and a USB 1.1 controller. This board can also be connected to an external SD card reader to provide more accessible memory that can be used by the device. Such features can be seen in the technical diagram in Figure 2.

Looking beyond the technical specifications of the Raspberry Pi Pico, this micro-controller was also found to be suitable due to its unique software features. Processes that are more complex on other devices such as programming the micro-controller's flash can be done easily on the Raspberry Pi Pico by simply copying the required files to the external drive attached via USB. Executing required code functionality of the Pico device is also straightforward, due to the presence of an extensive SDK (software development kit) for developing in either `C` or `C++` programming languages [12].

3 Proposed Approach

3.1 Implementation of the Multi-join Algorithm

To execute the Multi-join algorithm on the Raspberry Pi Pico, a software solution was developed that utilises Python header generation prior to execution. This header generation script is executed to perform pre-processing on relation files which contain each relation's attributes and attribute types alongside the actual data entries. This pre-processing creates a C++ header file containing a static declaration of the total number of relations and attributes as well as the number

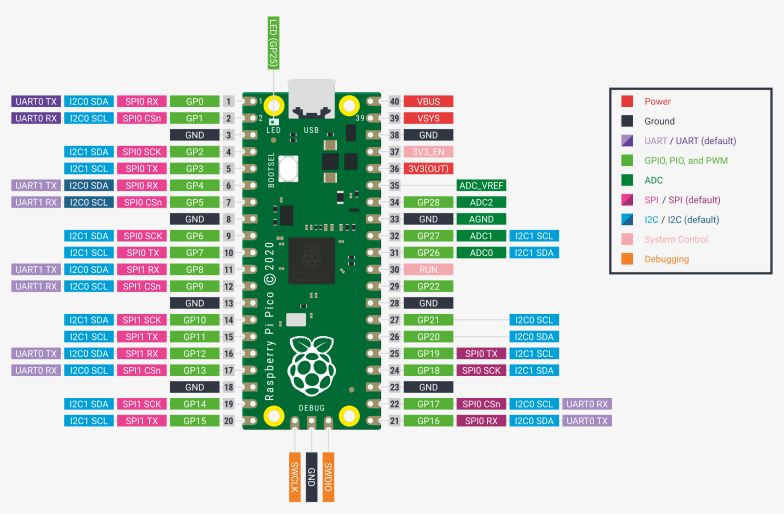

Fig. 2. A diagram that highlights the general purpose input output (`GPIO`) pins and the internal structure of a Raspberry Pi Pico. The diagram retrieved from Raspberry Pi Pico Datasheet [12].

of data entries in each relation. This allows for the static allocation of memory dependent on header file constants within the main C++ script.

Once the header file for the C++ script has been generated, the C++ script that performs the multi-join operation is compiled and executed on the Pico device. Within this script, data for the multi-join operation is stored using a simple array-based memory structure for both the Propose and Intersect stages. This structure allows for tuple entries to be accessed with relative ease using indexing when compared to more memory intensive approaches utilising pointers. Memory usage of this script was of high importance due to the 264 kB of SRAM available on the Raspberry Pi Pico. Only a couple of fundamental data types were supported with the algorithm performed by this script, including integer, char and double.

There are many potential improvements that could be made upon this implementation. At a memory management level, more efficient data structures could be chosen to allow for faster retrieval of tuples. The most obvious of these would be a balanced binary search tree or hash map approach to accessing and modifying data entries. Although these changes would provide much faster run-time, this would likely come at the cost of more memory overhead needing to be used by the script.

3.2 Implementation on Raspberry Pi Pico

Our implementation of the Multi-join algorithm on the Raspberry Pi Pico involves reading in relation files from the SD card, and outputting results to the computer via serial connection by USB. To achieve this, the following tools are required:

- An open source GitHub project [13]: A simple library allowing input and output interactions with the SD card connected to the Pico device.
- `Pico SDK`[12]: Provides headers and libraries necessary for Pico development
- `Cmake`: which is a open-source system to control the software compilation process using simple platform and compiler independent configuration files, and generate native build files.
- `Putty`: Allows observation of Pico serial USB output

First, the Pico SDK setup steps are followed in the Getting started with Raspberry Pi Pico guide [12]. A directory, `pico_version`, is created alongside the `pico-sdk` folder, and within it, all necessary executables and header files are placed. Within `pico_version`, a new sub-directory, lib, is created and within this, the simple library for SD cards GitHub project is cloned to. The `hw_config.c` file is found and copied to the `pico_version` directory, where it is altered to match our specific SD card wiring. This involves linking the `MISO`, `MOSI`, `SCK` and `CS` to their dedicated `GPIO` pins on the Pico, as well as stating which `SPI` we are using (in this case, 1). The SD card wiring is demonstrated in the diagram of Figure 3.

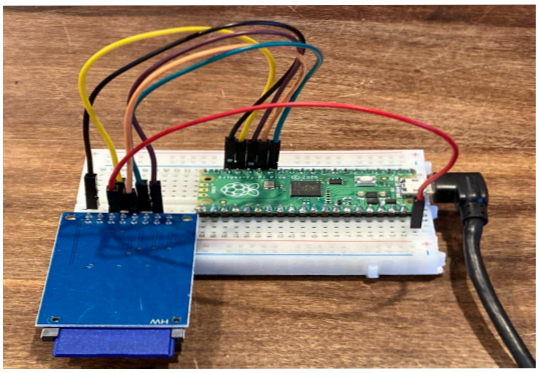

Fig. 3. SD card to Raspberry Pi Pico wiring

The SD card reader is connected to the Pico according to the following colour code:

- BLACK: `GND` (ground) to `GND` (Pin 18)
- YELLOW: `CS` (chip select) to `GP13` (Pin 17)
- PURPLE: `MISO` (master in slave out) to `GP12` (Pin 16)
- ORANGE: `MOSI` (master out slave in) to `GP11` (Pin 15)
- GREEN: `SCK` (serial clock) to `GP10` (Pin 14)
- RED: `VCC` (5V) to `VBUS` (micro usb input voltage (Pin 40)

A `CMakeLists.txt` file is needed, which links project directory, `pico_stdlib` and `Fat_Fs SPI` libraries and includes `prototype.cpp` as an executable. This file tells CMake/nmake how to create `*.uf2` executable files for the Pico to run.

Then, the python `file_autogen` script is run on target test case which will generate the header file for `prototype.cpp` to include. This must be run again for each different test case used. The test cases are also stored in a folder in the SD card and relevant `relationX.txt` files are added to the main directory of the SD card to be read by the Pico for each test case. `Cmake` and `nmake` are run which creates an executable file `sd_prototype.uf2`.

The Pico is then plugged into the computer by USB in `BOOTSEL` mode and the `.uf2` file is copied to the Pico directory. This reboots the Pico and runs the program. In order to observe serial output, Putty is quickly opened and connected to the dedicated serial line at a baud speed of 115200.

4 Experimental Results

4.1 MySQL BenchMarking

As a control, we did a MySQL benchmark on various measures. This benchmarking is done on a PC with Intel i5-10400F CPU. The measures we focused on are total execution-time, memory usage, and CPU usage. The data-set used to perform such join operations are filled with integers generated randomly with range from 0 to 4000. There are 3 tables joined for this benchmarking. Moreover, we measure 2 different parameters of system time, and user time in seconds. In terms of memory usage, and CPU usage there are the maximum usage, and average usage. Memory usage is displayed in GB, and CPU usage in percentage(100% in this case means 100% utilization of 1 core, so it is possible to have usage above 100%).

4.2 Implementation of Multi-Join Iterative on PC

We did the same benchmark to our implementation as well while running it on the same PC as when we did MySQL benchmarking. For evaluation purpose, we measure 2 different parameters of system time, and user time in seconds. In terms of memory usage, and CPU usage there are the maximum usage, and average usage. Memory usage is displayed in GB, and CPU usage in percentage.

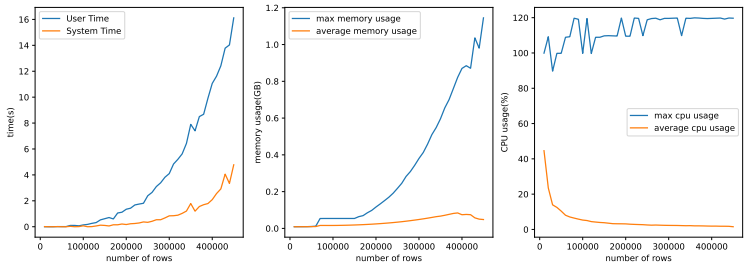

Fig. 4. Benchmarking result of MySQL. The most left plot depicts the total user space and system space execution time in respect to the number of rows. The middle depicts the memory usage in GB in respect to the number of rows. The left-most graph shows the CPU usage in percentage in respect to number of rows in each table.

4.3 Performance Measurement of Implementation of Multi-Join Iterative on Pico Device

Our Pico implementation was tested on system and user run-times, as well as the maximum number of relation rows the Pico could handle before running out of memory. This was tested on multiple frequencies to determine if higher frequencies could run the Multi-Join algorithm more efficiently.

System and user run-times were recorded as less than 0.1 milliseconds for all possible working frequencies of the device [2] where number of rows increases up to $N = 1500$, suggesting that altering frequencies for the Pico does not change its memory efficiency in running the Multi-Join algorithm. This creates difficulty around comparisons with the implementation run on PC, as the Pico cannot reach anywhere near the same number of test rows without any additional external memory. Once memory allocation is reached on the Pico, it outputs a `PANIC` message, informing the user there is no memory left to run the program. This can be quite dangerous as memory may be overlapped if the program is not interrupted.

5 Conclusion and Future Work

For future studies, there are many potential areas that could be explored. The most promising of these is the development of a distributed version of the proposed implementation. By separating the work to be performed by the Multi-join algorithm over distributed micro-controller devices, the memory limitations of each device could be bypassed to perform operations with much larger data inputs. This would require a complex means of dividing the input data for the

[2] In particular, we tested the experiment under three different working frequencies of $f \in \{48, 90, 132\}$MHz for Pico device.

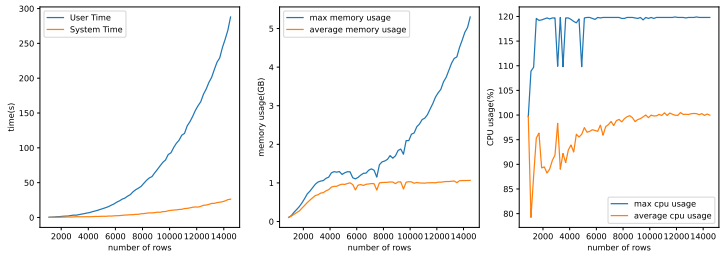

Fig. 5. Benchmarking result of our implementation on a PC with Intel i5-10400F CPU. The left-most plot depicts the total execution time spent on the user space and system space in respect to the number of rows. The middle plot depicts the memory usage in GB in respect to the number of rows. The right-most plot depicts the max. and the average CPU usage in percentage in respect to number of rows in each table.

propose step of the algorithm into discrete chunks that can be individually processed before being combined and stored in external memory.

Potential future research goals include increasing the scale of the architecture, by allowing for transmission between multiple Pico devices over distributed data and further improvements to complexity of the Multi-Join (and potentially other database) algorithms. Due to time constraints, an implementation utilising `std::unordered_set`, `std::tuple` and `variadic templates`, as a way of reducing the complexity was ultimately unsuccessful, however future solutions to our difficulties with templates and static declarations may be possible. Another area for future developments is improvements to the time complexity of the developed software solution. The approach covered in this paper utilises a naive array-based approach for storing large blocks of memory which could be easily optimised. Introducing a hash-map or binary search tree approach to storing data would allow for much faster operations to be performed including searching, inserting or deleting data entries within the Pico memory. However, such changes may result in additional memory overhead to store these data structures.

References

1. Ralph Hintemann. *Energy consumption of data centers continues to increase-2015 update*. Borderstep Inst. für Innovation und Nachhaltigkeit gGmbH, 2015.
2. Haoran Cai, Qiang Cao, Hong Jiang, and Qiang Wang. Greenhetero: Adaptive power allocation for heterogeneous green datacenters. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 160–170. IEEE, 2021.
3. National Renewable Energy Research Laboratory Golden CO (United States). Measurement and instrumentation data center, 2016.

4. Carolina Koronen, Max Åhman, and Lars J Nilsson. Data centres in future european energy systems—energy efficiency, integration and policy. *Energy Efficiency*, 13(1):129–144, 2020.
5. Ralph Hintemann. *The Future of Data Center Energy Demand. The Impact of the Changing Structure of Data Centers.* Borderstep Inst. für Innovation und Nachhaltigkeit gGmbH, 2014.
6. Jens Nothacker. Low-cost single-board-computers and learning-sets and the relation to the "digital" didactic goals of the 21st century. *Social Science Learning Education Journal*, 6(10):556–567, 2021.
7. Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2014.
8. Madhavan Thothadri. An analysis on clock speeds in raspberry pi pico and arduino uno microcontrollers. *American Journal of Engineering and Technology Management*, 6(3):41–46, 2021.
9. Stephen Smith. How to connect pico to iot. In *RP2040 Assembly Language Programming*, pages 265–289. Springer, 2022.
10. Mannu Lambrichts, Raf Ramakers, Steve Hodges, Sven Coppers, and James Devine. A survey and taxonomy of electronics toolkits for interactive and ubiquitous device prototyping. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 5(2), June 2021.
11. David J DeWitt and Robert Gerber. *Multiprocessor hash-based join algorithms.* University of Wisconsin-Madison, Computer Sciences Department, 1985.
12. Raspberry Pi (Trading) Ltd. Raspberry pi pico tech specs.
13. Carl J Kugler. Simple library for sd cards on the pico.