Parallel graph matching in Python with BSPy
BSc Thesis

Student	Supervisor
Mitchell Faas 4289986	Prof. Dr. Rob H. Bisseling

January 10, 2020

Utrecht University

1 Introduction

Nowadays, computers capable of parallel processing have become ubiquitous, with some suppliers going so far as to sell 32-core 64-thread processors in their consumer/pro-sumer lineup (like the AMD Thread-ripper 3970x). Yet if you ask people how often they've written parallel code, the answer is almost universally never. Though speculation on why is always exactly this - speculation - we suspect one of the reasons for this is that the world of parallel processing has a high barrier of entry. In this work we attempt to introduce a library which gives python developers the tools they need to start writing parallel programs while still enjoying the rapid development time of the language they're used to; we call this library BSPy.

BSPy is a simplified version of the BSP protocol [1], leaving out some functions entirely, without loss of applicability on shared memory machines. BSP as a protocol stands for *Bulk Synchronous Parallel* and defines a set work-flow for developers to follow in order to turn their sequential programs into parallel ones. This protocol is implemented through various libraries, such as *multicore-BSP* [2] for Java and C, and *Bulk* [3] or *BSPLib* [4] for C++. At the time of writing, one attempt has been made by Konrad Hinsen to write such a library for Python; this solution is only capable of real parallel computing when BSPLib or MPI has been installed on the system beforehand [5]. This work aims to function as a proof of concept to show that a similar library is possible – and practical – in pure Python; without the need of additional modifications or dependencies. In a practical display of the library, we will apply it to the problem of approximate bipartite cardinality matching on graphs using the Karp-Sipser algorithm [6].

2 BSPy

Knowing the essential background, we can delve deeper into the specific implementation of BSPy, and the design choices that went along with it. One important factor to talk about straight away is the so-called **Global Interpreter Lock** [7] or GIL. Since Python's memory management is not thread safe (that is to say, problems can arise if multiple threads were to edit and/or access the same piece of memory), the GIL is constructed to prevent multiple threads from executing Python byte-code at once. This is of course problematic for parallel programming and why any Python implementation of BSP would have to rely on different strategies than you would expect if you are used to a language like C. You would not, for example, just be changing what's stored in another processor's memory, as you are able to do with `bsp.put()`.

The defining factor of the BSP model is that it is divided up into distinct *super-steps*, which are of two categories: Computation and Communication and separated by *synchronization* steps. Importantly, no actual communication happens during either super-step (even though the name might suggest this), but instead, this communication is done during the synchronization steps. In short: a computation super-step lets a computation cell calculate whatever it needs to calculate based on information it already has, and a communication super-step stages the communication necessary before the next super-step happens. Synchronization will then execute the actual communication and stop all cells from doing any computations until they have all the information necessary.

Beyond that, we'll start by simplistically listing all the available functions within the BSPy library and talk a little about the back-end of these functions. We then move into the overarching design, which cannot be covered by just talking about functions, and finish off with some benchmarks.

2.1 Functions

The BSPy library revolves around one central object called `BSPobject`. Most of BSPy's functionality is found within this class, but there are two exceptions to this rule; these are `max_cores` and `run`.

For future reference, please note that if the functionality is a method of `BSPObject`, we will write the function as `bsp.foo()`, and similarly if the functionality is an attribute of `BSPObject`, we write it as `bsp.foo`.

`BSPObject` Before we continue, it is important to talk about what exactly a `BSPObject` is. In BSPy, we consider it important that computational cells (or processors) are clearly separated in the software layer. This means that we don't assume any inherent interconnectivity between two computational cells, aside from the communication protocol we are using. The advantage of this is that, although the current solution has been designed to work on a single shared memory machine, it becomes relatively easy to expand the use-case to include multiple machines on a local network, or even multiple machines over the internet. So long as the communication protocol of pipes is set up properly, it does not matter how these instances are connected. This is also what lets us get around the GIL; by running multiple Python interpreters, and setting up appropriate communication lines.

The `BSPObject` is a book-keeping object that functions as the foundation on which to build all other functions. It keeps track of the aforementioned communication lines (called pipes), but also keeps track of the

barrier used to ensure synchronized processing, how many computation cells there are, the id of this computation cell; it keeps track of what messages to send, and lastly contains a queue for received messages. This sounds like a lot, but in reality is nothing more than book-keeping.

`run()` The run function is exactly what creates `BSPObjects` and sets up all the appropriate communication lines, as well as the synchronization barrier. There are a few peculiarities to consider in the specific application, which are discussed in appendix A, as well as in the documentation [8]. Here we'll focus on the implementation.

The function will start off by creating communication lines (pipes) between computing cells. Each combination of cells has exactly one pipe, for a total of $\binom{n}{2}$ pipes, assuming n cores. It will then create a barrier of size n. All this means, is that any computing cell that "hits" the barrier is prevented from continuing its program, until all n cells have hit this same barrier. The barrier used here is the `multiprocessing.Barrier` [9]; it is critical in the functioning of `bsp.sync()`, which we'll discuss later.

After creating all pipes, and the barrier, the `bsp.run()` function starts spawning new Python instances using the `multiprocessing.Process` function[9]. Each of these processes is given the BSP function which the developer wrote to execute, and is endowed with their own `BSPObject`. This object contains only the information needed by this processor. Specifically, no computation cell has access to pipes from which it is not supposed to receive information, nor access to pipes to which it is not supposed to send information.

As a concrete example: Let a, b, c be computation cells, then the `BSPObject` that a is endowed with will have access to the receiving end of the $b \to a$ and $c \to a$ pipes, and the sending end of the $a \to b$ and $a \to c$ pipes, but access to no other pipes. This "need to know" concept minimizes the amount of information each cell has, and helps prevent errors by simply not allowing communication where there should not be.

`bsp.send()` This is the primary communication protocol used in BSPy, and it works similarly to the usual `bsp.send()`. As discussed at the start of this chapter, no communication actually happens until we reach a synchronization; so when calling `bsp.send()`, it will stage your message in a "to send" queue of the `BSPObject`. What makes this function interesting is that arbitrary python objects can be sent through the bsp.send() function. As a rule of thumb, anything you can assign to a variable can be sent via this function.

Since all communication is non-deterministic, it is important that the user adds their own structure when such structure is needed.

`bsp.sync()` synchronization is one of the crucial steps in any BSP library and BSPy is no different. Before we start with any sort of synchronization, we need to know that all processors are done with what they are supposed to do. Because of this, we start off by throwing up a barrier (which we talked about in our discussion of `bsp.run()`).

Once every computation cell is ready for synchronization, we move into a sending phase, where every message put in the queue of computation cells by the `bsp.send()` function is processed, and sent to the appropriate cell. The order in which this happens is *not* deterministic in our implementation, and need not be in other (future) implementations. This phase is finished by clearing the "to send" queue for a future round.

To ensure everyone is done transferring we throw up another barrier, before starting the receiving process. We also clear the previous "received" queue, so messages received before a synchronization but not used will be lost at this point.

During the receiving process, all messages which are still in the pipeline are stored in a "received queue", and can be accessed using `bsp.move()`. Note again that the order in which these elements are added to the queue is *not* deterministic.

The synchronization is finished off by putting up a final barrier to ensure every cell has cleared its pipeline and communication has been successful.

`bsp.move()` This function will return the first message in the queue, and return `None` in the case where no messages are left in the queue.

It bears repeating that since all communication is non-deterministic, it is important that the user adds their own structure when such structure is needed.

`bsp.cores`, and `bsp.nprocs()` Returns the total number of computing cells (cores) which are currently being used. Both of these do exactly the same thing, but `bsp.nprocs()` is included as a parallel to the usual `bsp.nprocs()` function.

`bsp.pid` Returns the id of this specific computing cell. This is completely congruent with the usual `bsp.pid`.

`max_cores()` This function, in many ways, speaks for itself. It's nothing more than Python Multicore's `cpu_count`[9] and counts the number of available virtual cores (the number of cores your computer sees, often called threads).

Note that `max_cores()` does *not* return the cores which are currently available in a program, but the total number of cores in the system. If you wish to know the number of cores your program is currently running on, use `bsp.cores`. On shared memory machine `bsp.cores` ≤ `max_cores()` at all times.

`bsp.time()` Just returns the current Unix time in seconds. Can be called even if no BSPObject has been defined (it is a static method).

2.2 Workflow

BSPy has been designed with simplicity in mind; it needs to be easy to use, and fit within the work-flow you're already familiar with. We believe that BSPy achieves pillars; here we'll delve into some of the reasons why. If you're interested in trying BSPy for yourself, please refer to appendix A.

Dynamic typing One of the big advantages of using Python is that it's a so-called **strong dynamically typed** language. What this means is that although the Python interpreter always knows what type a specific variable is (strongly typed), it also allows the developer to change the type of that variable at any arbitrary point in time, say from `int` to `float` (dynamically typed). This flexibility has advantages and drawbacks; it is for example one of the major reasons why Python is relatively slow in execution, but it is also a major reason why Python is so fast in development. We wanted to make sure that we kept this flexibility in typing, and let communication happen between computation cells, even when we don't know what is being sent. Note that this is in stark contrast to how BSP would be applied in a language like C, where you use specific memory pointers and the sizes of variables (like 32 bit for an int) to formally define where each value ends up (like when adding it to an array). In the design of BSPy, this was not an option.

To that end, and to ensure correct memory management, we opted to remove the `put` and `get` functionality entirely, leaving the developer to rely solely on the combination of `send` and `move`; similar to the way Green BSP functions [10]. These have further been simplified to the point where `send` only two parameters: the message, and the computing cell you wish to send the message to. This notably leaves out the option to send a "tag" as is common in other versions of BSP, like BSPLib [4] and multicore BSP [2]. Because we allow for arbitrary messages, we felt no need to include this functionality, as the developer is not prevented from sending a dictionary of the form {`'pid': s, 'message':...`}.

An added benefit is that the developer can send any variable, even if it is not a standard Python type (such as a numpy array).

2.3 Benchmarks

The speed of a BSP computer – or in this case, library – is usually summarized in 3 distinct numbers r, g, and l. Respectively these stand for the amount of flop/s, how fast a message gets sent (in flops) and how long synchronization takes (also in flops). These numbers together provide a good summary of how fast any particular BSP system is. Of course one could be forgiven for thinking along the following lines: *If performance is such an issue, why use python in the first place?* The answer is the same as it has always been: we are using python for the fast prototyping and short development times. Though it's absolutely true that many languages are a lot faster, few are as easy while still being as capable as python. An added bonus is that many libraries are available which execute C code, to also deliver performance where needed.

To maintain a high level of consistency we've opted to rewrite the benchmarking program found in R. Bisseling's BSPedupack[11] in python, and use this for the benchmark results.

This program found the following scores for our python implementation: $r = 15.3$ MFlop/s, $g = 530$, $l = 17590$. Note that the raw cost of synchronization is incredibly high, emphasizing the need to be careful with how often you synchronize; a common theme in BSP development.

3 Graph Matching

Before delving into the matching algorithm itself, we take the time to define some general facts about graphs[1], so that we may use them throughout this section.

Definition A graph G is a set of vertices $V(G)$, together with a set of edges $E(G) \subset \{\{x, y\} | x \neq y \in V(G)\}$.

Definition The degree $|v|$ of a vertex $v \in V(G)$ is the number of edges containing v. That is to say

$$|v| := |\{e \in E(G) : v \in e\}| .$$

Definition A matching $M(G) \subset E(G)$ of G is a subset of the edges in G such that no pair of edges in $M(G)$ shares the same vertex. That is to say

$$e_1 \cap e_2 = \emptyset \quad \forall e_1 \neq e_2 \in M(G).$$

3.1 The Karp-Sipser algorithm

The Karp-Sipser algorithm for creating an approximate maximum matching [6] is not very complicated and is displayed in algorithm 1.

$G :=$ The graph
$M = \emptyset$
while $|V(G)| > 0$ **do**
  if $\exists v \in V(G) : |v| = 1$ **then**
    Pick such a v.
  else
    Pick a random $v \in V(G)$.
  Pick $e \in E(G)$ containing v.
  $M = M \cup e$
  $V(G) = V(G) \setminus e$
  $E(G) = E(G) \setminus \{x \in E(G) : x \cap e \neq \emptyset\}$

Figure 1: The Karp-Sipser Algorithm

Note that the complexity of this algorithm is dependent on the speed at which the degree of a vertex can be calculated. Assuming an $O(1)$ lookup speed for degree, the Karp Sipser algorithm runs in $O(V)$ worst-case time complexity. This is however a gross oversimplification as we will see almost immediately when we start talking about the implementation.

During the parallelization of the Karp-Sipser program, we'll be writing the graph G as an adjacency matrix A where

$$a_{ij} := \begin{cases} 1 & \text{vertex } i \text{ is connected to vertex } j, \\ 0 & \text{otherwise.} \end{cases}$$

In this notation, we can define the degree of a vertex as

$$|v| = \sum_j a_{vj}.$$

Using this methodology, we can write the implementation of our algorithm using the numpy library[12], which provides C-like speed on computations.

3.2 Parallelizing Karp-Sipser

Due to the random picking of a single matching when no more singletons remain, it is very inefficient to parallelize the Karp-Sipser algorithm exactly. The algorithm we introduce here is strongly related to Karp-Sipser, but functions in a more parallelisable manner. Before we can talk about our parallel version of Karp-Sipser, we will have to talk about dividing the graph up. This segment is included for completeness, as in the algorithm's discussion we will be assuming every core already has access to their subgraph.

[1] In this thesis we only consider un-directed and un-weighted graphs. All the definitions given are thus based on these graphs.

Assumption 1: The graph's nodes are encoded as integers and exist as a dictionary of `int`, `list` pairs, where the list represents an adjacency list.
Assumption 2: Processor 0 has access to the graph when the program starts, e.g. it is able to read the graph from a file.

```
if bsp.pid == 0 then
  G := The graph
  <Set up the empty subgraphs for each processor>
  Subgraphs := an empty list
  for i ∈ {0, 1, 2, ..., bsp.cores − 1} do
    D := an empty dictionary
    Subgraphs.Append(D)
  <Fill the subgraphs for each processor>
  for key ∈ N(G) do
    owner = key  mod bsp.cores
    Subgraphs[owner][key] := G[key]
  <Send the subgraphs to their respective processor and Synchronise>
  for i ∈ {0, 1, 2, ..., bsp.cores − 1} do
    bsp.send(message=subgraphs[i], pid=i)
bsp.sync()
```

Figure 2: Dividing the graph

Although this looks complicated on paper, the code for doing this is all of 5 lines long and can be seen in listing 5, appendix B.

3.2.1 The building blocks

The algorithm itself operates in 3 distinct phases:

1. Match singletons;
2. Match randomly within a computing node;
3. Match completely randomly.

Each of these phases have their own progression, so let us start with the first and move on from there.

Match Singletons In words, this algorithm works in the following way: Go through the entire list of singletons and figure out who owns its connection. If we own it ourselves we can process it locally, but if another processor is the owner, we should pass it off to them. (They will eventually add it to their own graph and process it locally.) In pseudo code this comes down to figure 3, where it's important to note how we keep a set of nodes which are known to be dead.

```
G := Subgraph of this processor
S := A list of singletons
D := A (hash) set of dead nodes
for s ∈ S do
  dest := The connecting (destination) node
  owner = dest   mod bsp.cores
  if owner == bsp.pid then
    if dest ∈ G then
      Add match (s, dest)
      Declare both of {s, dest} as new dead
      D = D ∪ {s, dest}
    else
      // Node no longer exists, so connection is dead
      Declare (s) as new dead
      D = D ∪ {s}
  else
    if coreStatus[owner] == 0 or dest ∈ D. then
      // Node is dead, so connection can't be made.
      Declare (s) as new dead
      D = D ∪ {s}
    else
      // We are not the owner, and they're still working so let them take care of it
      bsp.send(message={'type':  'smatch', 'data':  (s, dest)}, pid=owner)
      Remove s from subgraph
```

Figure 3: Matching singletons

Match internally - Random When no processor has any singletons left, we're forced to start matching randomly within our own subgraph. When this occurs we do so in the manner described by figure 4.

```
G := Subgraph of this processor
S := A list of singletons
D := A (hash) set of dead nodes
while G ≠ ∅ and S == ∅ do
  (node, destNode) := an arbitry connection with node, destNode ∈ G.
  if (node, destNode) exists then
    Add (node, destNode) to matching
    Declare both of {node, destNode} as new dead
    D = D ∪ {node, destNode}
  else
    for pid in 0, ...., bsp.cores do
      bsp.send(message={'type':  'no local matches', 'pid':  bsp.pid}, pid=pid)
```

Figure 4: Internal Random Matching

Match externally - Random In the most extreme case where no processor is capable of matching nodes internally, we proceed to completely randomly make one single match – this is incredibly slow. This part of the algorithm can be described as in figure 5.

```
if bsp.pid == 0 then
  matchingProc := random processor from processors still running
  // Note that the contents of this message is actually irrelevant; as long as a message gets sent.
  bsp.send(messsage=-1, pid=matchingProc)
bsp.sync()
if received message then
  (node, destNode) := random edge
  Add (node, destNode) to matching
  Declare both of {node, destNode} as new dead
```

Figure 5: External Random Matching

Other building blocks Though these form the majority of the computational part of the algorithm, there are some parts, particularily message processing and book keeping which still need to be defined before we can list the entirety of this algorithm. We will start by defining the message processing and move on to cleaning the subgraph.

Processing messages is essentially just a list of if statements, each telling what to do with a certain type of message. This requires a lot of the overarching datastructures to be present, so we've defined most of them alongside the algorithm in figure 6.

G := Current subgraph
D := A (hash) set of dead nodes
C := A boolean array of length `bsp.cores` signifying the status of each processor. (initialised to `true` only)
L := A boolean array of length `bsp.cores` signifying if a processor has local random matches left. (initialised to `true` only)
S := A boolean array of length `bsp.cores` signifying if a processor has singletons left. (initialised to `true` only)
M := Current matching of this processor.
for message in queue **do**
  if messageType = deadNodes **then**
    $D = D \cup$ messageData
  if messageType = Done **then**
    C[messageData] = false
  if messageType = no Local matches **then**
    L[messageData] = false
  if messageType = no singletons **then**
    S[messageData] = false
  if messageType = match request **then**
    (node, destNode) = messageData
    if destNode $\notin D$ **and** destNode $\notin G$ **then**
      // The node is not dead and not on this processor, so we just sent it away and can add it to matching
      Add (node, destNode) to matching
      Declare both of {node, destNode} as new dead
    else
      Add node to subgraph with destNode as only connection

Figure 6: Message processing

Graph cleaning is a process that ensures there are no discrepancies in a graph (such as nodes without connections). This is an expensive operation and should preferably be done iteratively if going for a speedup. We have opted not to do this in favour for clarity. The pseudocode for this cleaning process can be found in figure 7.

G := current subgraph
D := A (hash) set of dead nodes
deadKeys := an empty list
for key $\in G$ **do**
  if key has no connections **then**
    $D = D \cup$ key
  if key $\in D$ **then**
    deadKeys.append(key)
  toRemove = an empty list
  for node $\in$ G[key] **do**
    if node $\in D$ **then**
      toRemove.append(node)
  G[key] = G[key]\ toRemove
  if G[key] = $\emptyset$ **then**
    deadKeys.append(key)
$G = G\setminus$ deadKeys

Figure 7: Graph cleaning

3.2.2 The algorithm

Having completed the primary building blocks we can now apply the overlaying structure which makes the program functional. This overarching program consists of some bookkeeping and calling of the segments we defined above with just a few added details and can be found in detail within figure 8. At this point it does becomes important to mention that this is not in fact an efficient algorithm; in fact it is very inefficient and you're better off running on one core. This graph matching is not done to show great algorithmic design, but rather to illustrate how BSPy is capable of handling complicated algorithms. In the next section we'll be using the simpler test cases of a Prime sieve and the NQueens problem to illustrate the additional efficiency that can be achieved with BSPy.

```
G := The subgraph
D := ∅ A (hash) set of dead nodes
C := [true, ..., true] A boolean array of length bsp.cores signifying the status of each processor.
M := ∅ Matching of this processor
while true do
  bsp.sync()
  L := [true, ..., true] A boolean array of length bsp.cores signifying if a processor has local random
  matches left.
  sLeft := [true, ..., true] A boolean array of length bsp.cores signifying if a processor has singletons
  left.
  newDeadNodes = ∅ A list of dead nodes created this superstep
  parseMessages()
  // If all processors are done
  if true ∉ C then
    // End algorithm
    bsp.send(message=|M|, pid=0)
    bsp.sync()
    if bsp.pid == 0 then
      totalMatchingSize := 0
      for i ∈ 1, 2, ...., bsp.cores do
        totalMatchingSize += bsp.move()
      print(totalMatchingSize)
    // Stop Running
    break
  cleanSubgraph()
  S := list of singletons
  if true ∉ L then
    matchRandomExternal()
  if G = ∅ then
    for pid ∈ 1, 2, ...., bsp.cores do
      bsp.send(message='type': 'Done', 'pid': bsp.pid, pid=pid)
  if S ≠ ∅ then
    matchSingletons()
  else
    if true ∉ sLeft then
      matchRandomInternal()
    else
      for pid ∈ 1, 2, ...., bsp.cores do
        bsp.send(message='type': 'no singletons', 'pid': bsp.pid, pid=pid)
  for pid ∈ 1, 2, ...., bsp.cores do
    bsp.send(message='type': 'deadNodes', 'data': newDeadNodes, pid=pid)
```

Figure 8: Parallel Karp-Sipser

4 Computational efficacy

As mentioned previously, we will use the simpler test cases of a prime sieve and an NQueens problem to illustrate the applicability of BSPy for real performance gain. Figure 9 shows timed examples of both these algorithms, the code for which can be found in the documentation of BSPy [8]. These results aim to show that real speedups are possible with efficient algorithmic design. For more details about the Prime Sieve we would like to refer you to a solution to the original course assignment [13], and for more details about the NQueens problem, as well as a view in to the early days of this library, we would like to refer you to the original BSPy paper [14]. As a fun little sidenote: This figure is actually an error-bar plot but the errors are too tiny to see without zooming in, which display the variance in time over 5 consecutive runs.

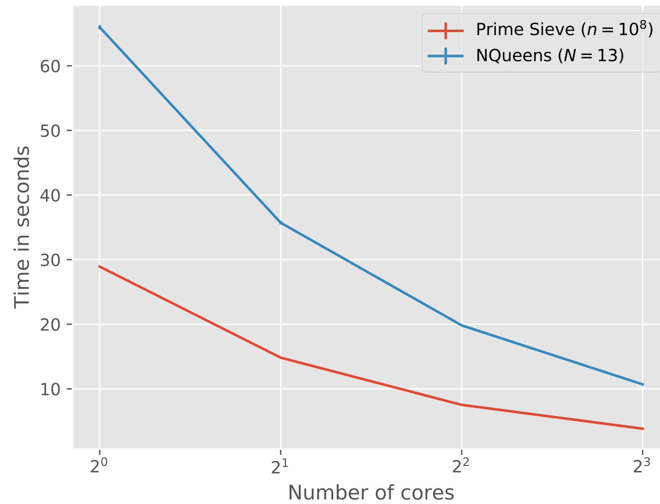

Figure 9: Comparing the time it takes in second to run an instance of the respective algorithms depending on how many cores are utilised.

As for the results of our parallel implementation of Karp-Sipser, there are some issues. In figure 10 we compare the speed of applying the Karp-Sipser algorithm sequentially versus running our parallel algorithm on one core. The latter is a factor 6 slower than the preceeding, which has two primary causes:

1. Due to the added communication and synchronisation, the algorithm has some overhead.
2. Because we need to be careful about communication when we run the algorithm, we actually need to go through several supersteps (3) before the program exits - even when using a single core; this takes a lot of time.

Additionally, when looking at figure 11, we notice that moving from one core to two actually slows the program down significantly. This oddity can be explained by the following line of reasoning: When running the program on one core, no actual communication is done. To the contrary, when two cores are involved, both need to communicate a lot to ensure no mistakes are made. This causes many more supersteps and slows the program down significantly. If one were able to assume some structure on the graph, a better partitioning could likely be found which would reduce communication. Since we don't assume such a structure, this is not the case here.

Once the program is run with eight cores, a speedup becomes noticible. At this point the added computation power is enough to counteract the additional communication.

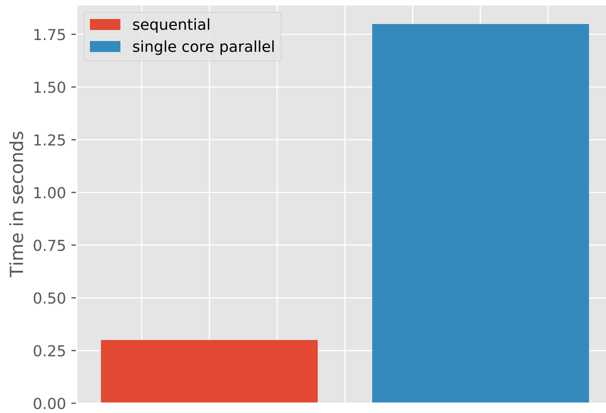

Figure 10: Comparing the time it takes in second to run an instance of the sequential algorithm (in red) to the time it takes to run an instance of the parallel algorithm on one core (in blue).

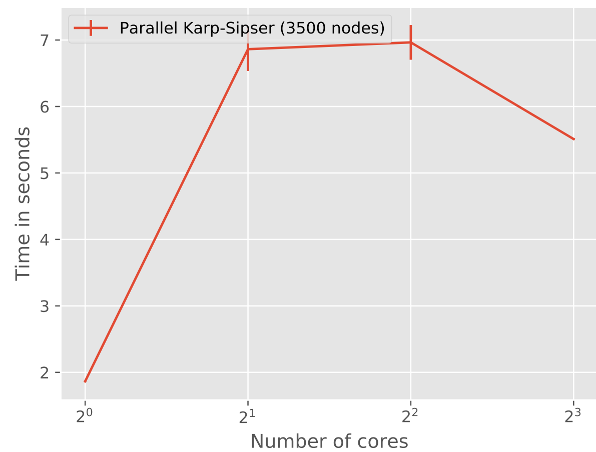

Figure 11: Showing the time it takes to run the parallel algorithm on a test graph of approximately 3500 nodes.

5 Discussion

Though the graph matching algorithm we use in this thesis is not any you would be recommended to use for any real application, it shows the ability of BSPy to handly complex algorithms, with – for example – synchronisations within while loops. Combined with the computationally efficient results found in the parallelisation of the Prime Sieve and NQueens algorithms in section 4, we believe that this functions as a sound basis for the proof of concept we hoped to provide in this thesis.

The library itself lacks many features, perhaps most importantly an elegant way to handle exceptions. (Currently your best bet to debug programs is to use lots of print statements.) Others include quality of life features such as being able to efficiently loop over a received queue. e.g.

while `bsp.queue` **do**
  message = `bsp.move()`

There are some stability issues present as well, which can result in unpredictable behaviour such as a program requiring either 1 or more than 8 cores to run effectively. We found that these stabilities often go hand in hand with algorithmic mistakes but cannot explain why this would be the case.

All in all, we believe that BSPy as it starts is a functional proof of concept which can serve as a foothold for future expansion, and may serve useful to beginning (parallel) programmers or those that only have experience programming in python.

References

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, pp. 103–111, Aug. 1990.

[2] A. Yzelman and R. H. Bisseling, "An object-oriented bulk synchronous parallel library for multicore programming," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 5, pp. 533–553, 2012. `http://www.multicorebsp.com/`.

[3] J.-W. Buurlage, T. Bannink, and R. H. Bisseling, "Bulk: A Modern C++ Interface for Bulk-Synchronous Parallel Programs," in *Euro-Par 2018: Parallel Processing* (M. Aldinucci, L. Padovani, and M. Torquati, eds.), pp. 519–532, Springer International Publishing, 2018.

[4] M. van Duijn, K. Visscher, and P. Visscher, "BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model.." `http://bsplib.eu/`.

[5] K. Hinsen, "Parallel scripting with python," *Computing in Science and Engineering*, vol. 9, pp. 82–89, 12 2007.

[6] R. M. Karp and M. Sipser, "Maximum matching in sparse random graphs," in *Proceedings of the 22Nd Annual Symposium on Foundations of Computer Science*, SFCS '81, (Washington, DC, USA), pp. 364–375, IEEE Computer Society, 1981.

[7] Python Contributors, "Global interpreter lock." `https://wiki.python.org/moin/GlobalInterpreterLock`. [Online; accessed 11-Nov-2019].

[8] Mitchell Faas, "Bspy documentation." `http://mitchellfaas.com/BSPy/`. [Online; accessed 11-Nov-2019].

[9] Python Contributors, "multiprocessing." `https://docs.python.org/3.8/library/multiprocessing.html`. [Online; accessed 11-Nov-2019].

[10] M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas, "Portable and efficient parallel computing using the bsp model," *IEEE Transactions on Computers*, vol. 48, no. 7, pp. 670–689, 1999.

[11] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, 2004.

[12] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[13] C. Aronis and M. Faas, "Prime sieve." `http://mitchellfaas.com/BSPy/download/Prime_sieve.pdf`, 2018. Accessed: 30-12-2019.

[14] M. Faas and C. Aronis, "Bspy." `http://mitchellfaas.com/BSPy/download/BSPy--original-project.pdf`, 2019. Accessed: 30-12-2019.

A Quickstart

There are a few steps to go through in order to start using your version of BSPy. Here we'll discuss everything you need to get started and introduce a minimal program which uses all the available functions.

Step 1: Installing BSPy We're going to assume that you've got Python already installed. If this is the case, the easiest way of installing BSPy is the usual pip install; via the command `pip install BSPython`.

As for requirements there are relatively few, we just require that you use Python version 3.6 or higher.

Step 2: Writing your first program Now that you've installed BSPy, we'll start by writing a simple "Hello world!" program to verify everything is working correctly.

First, we'll start by importing BSPy, and defining our `hello_world` function:

```
import bspy  # Import the library

def hello_world(bsp):  # Define the function. Note that the first argument is reserved for the
    BSPObject
  pass
```

In this `hello_world` function, we can now write our parallel program. Let's have every processor shout "hi!" together with its id.

```
import BSPy

def hello_world(bsp):
  cores = bsp.cores  # How many cores are there in this run?
  pid = bsp.pid  # Which core is this?
  print(f"Hello World from processor {pid}! (out of {cores})")  # Say hello!
```

Now that we have the function, and have imported BSPy, the only thing left to do is add the run command. Let's run this program on 4 cores.

```
import bspy

def hello_world(bsp):
  cores = bsp.cores
  pid = bsp.pid
  print(f"Hello World from processor {pid}! (out of {cores})")

if __name__ == '__main__':  # Required construct.
  bspy.run(hello_world, 4)  # Run hello_world () on 4 cores
```

Listing 1: Hello World Program

```
>>> Hello World from processor 0! (out of 4)
>>> Hello World from processor 2! (out of 4)
>>> Hello World from processor 1! (out of 4)
>>> Hello World from processor 3! (out of 4)
```

Listing 2: Output

You'll note that the only strange thing here is the `if __name__ == '__main__':` construct. This is required to let every instance of Python run well and behave. If you don't do this, you'll get an error telling you to do exactly this, so don't worry about remembering it.

Every processor should print hello to screen. Note that the order in which this printing occurs is semi-random and that multiple processors printing to the same console is often not very pretty. That said, it works as a simple verification of your install.

Step 3: Minimum program for all functions Okay, so now that we've written a simple hello world program, it's time to get familiar with the functionality BSPy has to offer. We'll write a similar program as above, but add a little needless complexity to show the use of all functions. Before saying hello, we're going to pass a message on to the next processor, and only if we receive that message, will we say hello. In this example we won't care about what message we send, just that we can send and receive it.

Using our hello world program as boiler plate code, we're really not that far away, so will write the changes in one go:

```
import bspy

def hello_world(bsp):
  cores = bsp.cores
```

```
5    pid = bsp.pid
6
7    bsp.send('You have green light!', (pid+1)%cores)  # Send green light to the next processor.
8
9    bsp.sync()  # synchronize the processors
10
11   message = bsp.move()  # Grab the message from the received queue
12
13   if message:  # Python jargon for message is not None
14     print(f"Hello World from processor {pid}! (out of {cores})")
15
16 if __name__ == '__main__':
17   bspy.run(hello_world, 4)
```

Listing 3: Hello Communication

Note here that we're just sending the message to a processor, and don't need to specify anything along the lines of memory addresses; nor are we concerned with specifying any variable names. Also note that the move and sync function don't take any arguments, the sync function is positioned in between the send and move (to make the actual communication happen), and that the move function *returns* the message. This is in contrast with the more common way of assigning the message to an already existing variable; and allows for dynamic typing.

B Code

B.1 Sequential Karp-Sipser

```
1  import networkx as nx
2
3  G = nx.barbell_graph(25, 5)
4  matching = []
5
6  while G.order() > 0:  # O(n) (or O(V))
7    # Pick a node
8    for node in G:  # O(n) (or O(V))
9      if G.degree(node) == 1:
10       # Select the first edge
11       current_vertex = node
12       break
13   else:  # O(1)
14     # No node of degree 1 exists, so pick arbitrary
15     current_vertex = list(G.nodes)[0]
16
17   # Try to find an edge, if the vertex has one
18   try:
19     edge = list(G.edges(current_vertex))[0]
20     matching.append(edge)
21     # Remove nodes of the edge
22     for node in edge:  # O(1)
23       G.remove_node(node)
24   except IndexError:
25     # Just remove this node
26     G.remove_node(current_vertex)
```

Listing 4: Sequential Karp-Sipser

```
1 if bsp.pid == 0:
2   # Create the subgraphs. Note we require nodes to be integer encoded
3   subgraphs = [{key: graph[key] for key in graph if key % bsp.cores == s}
4                                       for s in range(bsp.cores)]
5
6   for i in range(bsp.cores):
7     bsp.send(message=subgraphs[i], pid=i)
8 bsp.sync()
```

Listing 5: Graph division