A Stencil Skeleton for a DICE-Friendly Parallel Skeleton Library

Sheng Li

Master of Science
School of Informatics
University of Edinburgh
2019

Abstract

As processors moves from single-core towards multi-core, software transit from single-threaded to parallel applications. Skeletons are designed to simplify the task of parallelising programs and produce well-written parallel programs. But skeleton-based libraries are hard to configure and maintain. This project aims to extend an existing Distributed Informatics Computing Environment (DICE) friendly skeleton library by adding a stencil skeleton. We describe the design and implementation of the new skeleton and evaluate its performance on two multi-core platforms.

Acknowledgements

I would like to thank my supervisor, Prof. Murray Cole, for all his help and guidance throughout this project, and for his swift responses and patient instructions to help me get better results. And thanks to my parents for their firm support.

Declaration

Contents

Chapter 1

Introduction

Moore's law is coming to an end. Single-threaded software can no longer benefit from the free ride of processor improvement. In order to exploit multi-core processors, software needs to be redesigned concurrently.

1.1 Structured Parallel Programming

Redesigning a single-threaded application to be multi-threaded is easier said than done. Parallel programming is hard and some problems are not inherently parallelisable. A common way to simplify parallel programming and produce well-written code is structured parallel programming. Structured parallel programming is based on a composition of skeletons. A skeleton captures a common algorithmic structure of certain type of problems. Common skeletons are Map, Reduce, Scan and Stencil. For example, stencil skeleton abstracts common pattern of stencil computation and hides away low-level details of parallel programming. It provides a higher-level vocabulary for designing parallel algorithms and eases the task of understanding others' algorithms by providing a friendly abstraction (McCool et al., 2012). Programmers can define operation to be applied concurrently, and the skeleton library will encapsulate user defined operation in parallel program. The idea behind Structured parallel programming is decoupling computation and coordination (Gonzalez-Velez and Leyton, 2010) so that programmers can express their intent in a sequential manner and let skeleton library handles coordination between threads.

1.2 Problem and Solution

A skeleton library often depends on certain versions of compilers and tools. Configurations and maintenance of such libraries is overwhelming for inexperienced programmers. We proposed to design a DICE-friendly parallel skeleton library which allows students to easily use skeletons in DICE. DICE stands for Distributed Informatics Computing Environment. It encompassed all sorts of computing services maintained by Informatics School, University of Edinburgh. DICE desktops and servers run a version of Scientific Linux. DICE is available to students, researchers and staff from Informatics School. The goal of this project is to build and evaluate a stencil skeleton for an existing skeleton-based parallel programming library which has no dependencies on tools unavailable in DICE. Our DICE-friendly parallel skeleton library requires GCC 4.8.5 (C++11) and Pthreads.

1.3 Thesis Structure

Chapter 2 presents related work and background information. It briefly covers the design of SkePU2 (Ernstsson et al., 2018), work done in a previous project (Valias, 2018). Chapter 3 presents the interface design and implementation of our new stencil skeleton. Chapter 4 provides information on hardware used for evaluation and demonstrates results of evaluation. Chapter 5 concludes this project and suggests future work.

Chapter 2

Background and Related Work

2.1 Stencil

Stencil skeleton is a generalisation of Map skeleton. Map replicates a function over a set of elements. This mapped function is called an elemental function. Map's elemental function takes a single element as input and return some value as output. Instead of applying elemental function to a single element, Stencil applies its elemental function to a region of elements. The output of this elemental function depends on a set of neighbours. Figure 2.1 is a example of stencil computation in 2D where each output is computed from 4 neighbours. Each neighbour is located a fixed offset from the central element. This example is also known as heat transfer or Jacobi. Listing 4.4 is a code snippet of this example. Problems that Map and Stencil try to solve give us the chance to exploit data parallelism. Stencil computation is very common in image processing and partial differential equation solvers. We will see more examples in section 4.3.

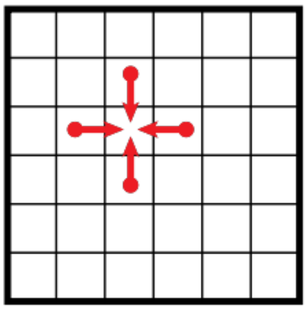
Figure 2.1: 2D stencil computation

2.2 SkePU2

There exist many skeleton libraries, such as eSkel (Cole, 2004), SkePU2 (Ernstsson et al., 2018), Muesli (Ciechanowicz et al., 2009), HaskSkel (Hammond and Rebón Portillo, 2000). These skeleton libraries have different programming paradigms. eSkel uses imperative programming framework; SkePU2 and Muesli is object-oriented; HaskSkel uses functional programming. Skeleton libraries also differ in parallel architectures. SkePU2 is based on shared memory model; Muesli is based on distributed memory model. Our library is an object-oriented library based on shared memory model, and we mainly refer to the design of SkePU2.

SkePU2 is an object-oriented skeleton programming framework implemented in C++11. It supports multi-core and multi-GPU. It includes 6 skeletons, Map, Reduce, MapReduce, MapOverlap (also known as Stencil), Scan and Call. Its programming model is shared memory model.

MapOverlap's elemental function takes a region of elements as input and returns a single element as output. In SkePU2, this region is passed as a pointer. Users assess the data by pointer arithmetic. The pointer points to the centre element.

MapOverlap supports both 1D and 2D. `MapOverlap1D` uses vectors as containers; `MapOverlap2D` uses matrices as containers. The order of parameters for MapOverlap's elemental function is important. It always take arguments in this order: radius, stride, a pointer of contained type pointing to the centre of the overlap region. `MapOverlap2D` has one more argument, radius in y-direction, right after radius in x-direction.

MapOverlap provides 3 edge modes, cyclic, pad and duplicate to handle out-of-bounds accesses. It is implemented using a switch-statement. Cyclic mode wraps around input vector or matrix at boundaries. Pad mode pads constant values. Default value is zero. Users can supply the constant value by a setter `setPad(pad)`. Duplicate mode duplicates the closest element. The default edge mode is cyclic. Users can set edge mode by a setter `setEdgeMode(mode)`.

Listing 2.1 is an example usage of SkePU2 MapOverlap1D. It uses a vector as the input container. Function `conv()` is the elemental function, and it take arguments in order mentioned above. `overlap` is the radius of the window. The region of elements is accessed via a float pointer `v`. `stencil` is a filter. `scale` is an extra parameter used in user defined computation. Caller function `convolution()` takes a reference to input vector. First, a function object `convol` is created. It is bounded to user defined function `conv()`. `result` is a output vector with the same size as input. Edge mode is set to padding constant values. Finally, it calls this functor `convol` by the overloaded parenthesis operator. Parenthesis operator takes output vector and input vector as first 2 arguments, followed by a 1D filter `stencil` and a scale factor.

```
  float conv(int overlap, size_t stride, const float *v, const
     Vec<float> stencil, float scale) {
        float res = 0;
        for (int i = -overlap; i <= overlap; ++i)
        res += stencil[i + overlap] * v[i*stride];
        return res / scale;
  }
  Vector<float> convolution(Vector<float> &v) {
        auto convol = MapOverlap(conv);
        Vector<float> stencil {1, 2, 4, 2, 1};
        Vector<float> result(v.size());
        convol.setOverlap(2);
        return convol(result, v, stencil, 10);
}
```

Listing 2.1: SkePU2 MapOverlap example usage (Ernstsson, 2016)

2.3 A Previous Work

In a previous DICE-friendly skeleton project, Valias (2018) implemented Map, Reduce, MapReduce and Scan. This project continues Valias's design and coding style. The common design of all skeletons is a wrapper model. A wrapper class encapsulates all components of a skeleton. A skeleton is mainly composed of an implementation class, an elemental class and a thread argument class. Implementation class defines the behaviour of skeleton algorithm, including thread code which will be parallelised and serial code which initialises thread arguments and coordinates threads. An elemental object will contain the user defined function. This wrapper model prevents users from accessing inside skeletons and changing skeleton's behaviour. It only allows users to create a functor of the skeleton and invoke the operation of the skeleton. The wrapper model modularises our skeletons and makes maintenance and subsequent improvement easier.

The usage of a skeleton has 2 phases: instantiation and execution. In phase 1, a skeleton object bounded with a user defined elemental function is created. This skeleton object is a function object which can be called by overloading a function call operator. In phase 2, we invoke this function object to perform the skeleton operation to input.

In Valias (2018)'s third implementation, "task stealing" is introduced for dynamic load balancing. It is implemented using flags and mutexes. Figure 2.2 illustrates an example of task stealing where 2 threads with data block of size 3 are processing the input data. Threads are given a data chunk of the same size in the beginning. The data chunk is divided into data blocks. A thread iterates over data blocks and perform some computation. Each data block associates with a flag indicating whether all elements inside the data block has been processed or not. 1 indicates the data block is not processed; 0 indicates data block has been done. Mutexes prevents multiple threads modify a flag at the same time. Suppose a thread finished its work earlier, it checks if its next neighbour has any data block which have not been processed. If there exists any unprocessed data block, it "steals" it and works on it. This Stencil skeleton keeps task stealing as a way to balance workload between threads.

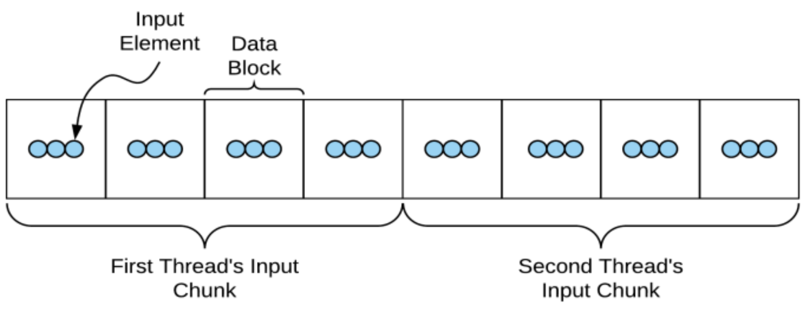

Figure 2.2: Task stealing (Valias, 2018)

Chapter 3

Design and Implementation

3.1 Interface Design

In our design, neighbourhood is passed as a pointer. Elements are assessed by pointer arithmetic. The pointer points to the first element in the region. We have 3 stencil skeletons support 1D, 2D, 3D. They are separated in 3 files named `Stencil1D`, `Stencil2D`, `Stencil3D`. All use vectors as containers. `Stencil2D` and `Stencil3D` use 1D vector to represent 2D and 3D vectors. Figure 3.1 is the class diagram of `Stencil3D`. To use stencil skeletons, we need 2 phases, instantiating a functor of stencil and invoking stencil operation by calling the overloaded parenthesis operator. In the first phase, an instance is created by calling a factory function named after the skeleton. Parameters always start with an elemental function and a `size_t` parameter that is radius of the window. The window is implemented as a square whose sides are $radius \times 2 + 1$. `Stencil2D` also has 2 more parameters, number of rows, number of columns. `Stencil3D` has 3 more parameters dimension in x, y, z directions. The next parameter is padding option. This stencil skeleton supports 3 boundary decisions: wrap around, padding constant values and duplicate the closest element. The last 2 parameters are optional. The first optional parameter is the number of iterations. This stencil skeleton allows users to apply stencil computation multiple times. The default number of iterations is 1. The last parameter is the number of threads. If number of threads is not specified, threads number will be the maximum number of threads that hardware supports. Additionally, user can pass any extra arguments at the end.

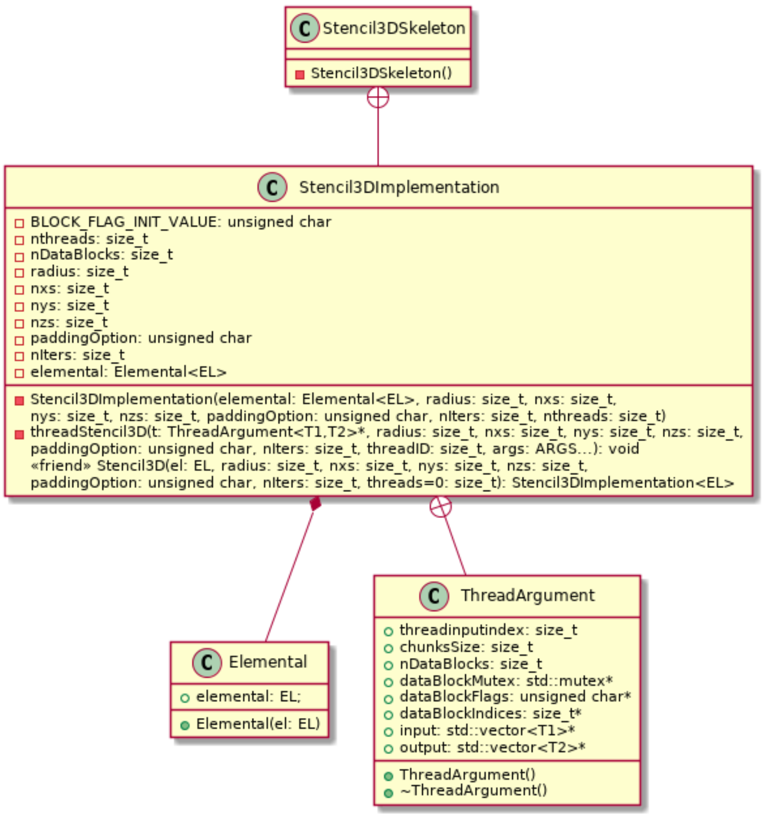
Stencil3DSkeleton
Stencil3DSkeleton()
Stencil3DImplementation
BLOCK_FLAG_INIT_VALUE: unsigned char
nthreads: size_t
nDataBlocks: size_t
radius: size_t
nxs: size_t
nys: size_t
nzs: size_t
paddingOption: unsigned char
nIters: size_t
elemental: Elemental<EL>
Stencil3DImplementation(elemental: Elemental<EL>, radius: size_t, nxs: size_t, nys: size_t, nzs: size_t, paddingOption: unsigned char, nIters: size_t, nthreads: size_t)
threadStencil3D(t: ThreadArgument<T1,T2>*, radius: size_t, nxs: size_t, nys: size_t, nzs: size_t, paddingOption: unsigned char, nIters: size_t, threadID: size_t, args: ARGS...): void
«friend» Stencil3D(el: EL, radius: size_t, nxs: size_t, nys: size_t, nzs: size_t, paddingOption: unsigned char, nIters: size_t, threads=0: size_t): Stencil3DImplementation<EL>
Elemental
elemental: EL;
Elemental(el: EL)
ThreadArgument
threadinputindex: size_t
chunksSize: size_t
nDataBlocks: size_t
dataBlockMutex: std::mutex*
dataBlockFlags: unsigned char*
dataBlockIndices: size_t*
input: std::vector<T1>*
output: std::vector<T2>*
ThreadArgument()
~ThreadArgument()

Figure 3.1: `Stencil3D` class diagram

Listing 3.1 is an example usage of stencil usage. The computation defined in function `stencilkernel` is illustrated in figure 2.1. `stencilkernel` takes a pointer to neighbourhood. It returns the average of 4 neighbours. We first initialise a function object by calling a factory function `Stencil2D()`. `Stencil2D()` takes parameters in the order described above. We then pass output and input vector to operator function to perform stencil operation.

```
double stencilkernel (double neighbourhood[]) {
        double sum = 0;
        sum = neighbourhood[1] + neighbourhood[3] + neighbourhood
            [5] + neighbourhood[7];
        return sum/4.
}
void parallelHeatTransfer(std::vector<double> &output, std::
    vector<double> &input) {
  auto stencil2d = Stencil2D(stencilkernel, RADIUS, NROWS,
      NCOLS, PADDING, NITERS, NTHREADS);
  stencil2d(output, input);
}
```

Listing 3.1: Stencil example usage

3.2 Implementation

This section explains the implementation by following 2-phases usage mentioned in section 3.1. I will take `Stencil3D` as an example. The first function called by user is the `Stencil3D`. Friend function `Stencil3D` acts as a factory function. It passes all arguments to `Stencil3DImplementation`. `Stencil3D` also instantiates an instance of `Elemental`. `Elemental` has nothing but a constructor and an elemental function of any type. It will contain a function pointer to the user defined function. Both `Stencil3DImplementation` and `Elemental` are inner classes of `Stencil3DSkeleton`. `Stencil3DSkeleton` is their wrapper class. `Stencil3DImplementation` contains `threadArgument` class, thread code `threadStencil3D` and all primitives shared between threads, such as input size, number of threads, radius of window. `threadArgument` contains pointer to input and output vector, size of data blocks, size of data chunk, pointer to corresponding data block flags, pointer to data block indices, `threadInputIndex`. Each thread is given a chunk of data, that is the load of work this thread is going to do. A data chunk is divided into several data blocks. `threadInputIndex` is the index of first element in assigned data chunk. Inside `threadStencil3D`, it builds a temporary buffer, then calls `elemental` function. Stencil's elemental function takes a pointer to this temporary buffer as input and returns a single element as output. At the end of each iteration, each thread synchronises with others and resets block flags. The second phase is to invoke stencil operation by calling overloaded parenthesis operator. This operator function also takes parameters in order. Its first 2 parameters must be an output vector and an input vector. It allows any number of extra read-only arguments. Parenthesis operator assigns all thread arguments, create threads, join threads, finally delete threads and thread arguments. Boundary decisions are implemented using a switch-statement. Wrap-around, constant value and closest element are defined as 3 constants.

Listing 3.2 shows how neighbourhood array is built when wrapping around at boundaries. The outermost loop goes through each item to be processed. The 3 inner loops iterate over 3 dimensions of the filter. In the outermost loop, for each item in data chunk, its 1D index in input vector is converted to corresponding 3D coordinates. In the innermost loop, for each neighbour in the filter window, it calculates its 3D coordinate and wraps around if necessary. Then, it converts this 3D coordinate to 1D index in input vector. It reads the item in input vector by this 1D index and assign the value to local neighbourhood.

```
for(size_t elIdx = dataBlockIndices[dataBlock]; elIdx <
    dataBlockIndices[dataBlock+1]; ++elIdx) {
  int elx = elIdx % nxs;
  int ely = (elIdx / nxs) % nys;
  int elz = elIdx / nxs / nys;
  int neighbourx, neighboury, neighbourz;
  for (int filterz=0; filterz<2*radius+1; ++filterz) {
        for (int filtery=0; filtery<2*radius+1; ++filtery) {
          for (int filterx=0; filterx<2*radius+1; ++filterx) {
                switch (paddingOption) {
                  case WRAP_AROUND: {
                    neighbourx = (elx+filterx-radius+nxs)%nxs;
                    neighboury = (ely+filtery-radius+nys)%nys;
                    neighbourz = (elz+filterz-radius+nzs)%nzs;
                    neighbourhood[filterx+(2*radius+1)*(filtery
                        +(2*radius+1)*filterz)] = input->at(
                        neighbourx+nxs*(neighboury+nys*
                        neighbourz));
            break; }
          // other cases ...
} } } } }
```

Listing 3.2: Neighbourhood

Chapter 4

Evaluation

4.1 Hardware Specifications

Stencil skeletons are benchmarked on 2 machines, a student lab desktop and a multiprocessor node on James cluster. The specifications of 2 machines are listed in table 4.1. A lab desktop has a 4-core processor with maximum frequency of 3.6GHz. A James multiprocessor node has a 64-core processor with maximum frequency of 2.3GHz. The version of g++ compiler I used is 4.8.5 20150623 (Red Hat 4.8.5-36). All programs are optimised by compiler at O2 level.

	CPUs	Threads/core	Max GHz	L1d/KB	L1i/KB	L2/KB	L3/KB
Lab desktop	4	1	3.6	32	32	256	6144
James	64	2	2.3	16	64	2048	6144

Table 4.1: Hardware specifications of lab desktop and James multiprocessor node

4.2 Experimental Programme

In development of stencil skeletons, I implemented a set of examples for testing. For each example, I implemented 3 versions of it: a sequential program written in C++, a parallel program written in C using Phtreads library and a parallel program using our skeleton library. The Pthreads version is not dynamically load balanced. It distributes tasks to threads evenly and statically. In contrast, our skeleton library is dynamic load balanced. All parallel programs had been tested by comparing the results with sequential programs before they were used for experiments.

To evaluate the performance and scalability of our stencil skeletons, I planned 2 sets of experiments. My first experiment set was to evaluate performance. I measured the program execution time with different number of threads running. On a student lab desktop, I tested thread number of 1,2 3, 4. On a James node, thread number was doubled from 1, until it reached maximum number of threads supported by hardware, that is 64. I tested sequential program, Pthreads program and skeleton program of each example with a fixed number of threads. I benchmarked them against 4 metrics: execution time, speedup, efficiency and cost. Cost is defined as the product of execution time and number of threads, and I plotted these 4 metrics with respect to number of threads. According to Amdahl's law, speedup is limited by the serial part of the program. It is expected that speedup will reach a plateau, as number of threads increases. My second experiment set was to evaluate scalability. I measured the program execution time with different problem sizes. We can see whether the program scales well with problem size. I tested skeleton programs with varied number of items, and I plotted the execution time with respect to the size of input vector.

4.3 Examples

Eight examples are implemented for our stencil skeletons. `Stencil1D` has 2 examples, `Sum1D` and `Median1D`. `Stencil2D` has 4 examples, `Sum2D`, `Median2D`, `HeatTransfer2D`, and `GaussianBlur2D`. `Stencil3D` has 2 examples, `Sum3D` and `Median3D`. Example `Sum` sums up neighbouring elements and itself and returns the sum to its position. Example `Median` finds the median value in neighbourhood and returns the median to its position. `HeatTransfer2D` averages 4 neighbours. `GaussianBlur2D` convolves the image with a Gaussian function. For each example, I implemented sequential code, Pthreads code and parallel code using our skeleton library. Each program ran 5 times and we reported averaged execution time with standard deviations.

4.3.1 Sum

`Sum1D` elemental function is shown in listing 4.1. Parameter `neighbourhood` points to all elements in a window which slides across the whole data array. The length of this array is $(2 \times radius + 1)$. This elemental function adds all elements in the window and return the sum. `Sum2D` and `Sum3D` are basically the same except that neighbourhood has a different size. In `Sum2D`, neighbourhood's size is $(radius \times 2 + 1)^2$. In `Sum3D`, its size is $(radius \times 2+1)^3$. We can see that users do need to change much of the elemental function to adapt different dimensions. It is all handled by skeletons. When programming the same examples using raw threading API, like Pthreads, apart from defining sum computation, we need to ensure that we access elements of correct indices.

In the first phase, an instance of `Stencil1D` is created by calling its factory function named after itself. The type of this instance is declared as `auto` because it is implementation-defined.

```
int stencilkernel (int neighbourhood[], int radius) {
      int sum = 0;
      for (int i=0; i<radius*2+1; ++i)
  sum += neighbourhood[i];
      return sum;
}
void parallelSum(std::vector<int> &output, std::vector<int> &
    input) {
  auto stencil = Stencil1D(stencilkernel, RADIUS, PADDING,
      NITERS, NTHREADS);
  stencil(output, input);
}
```

Listing 4.1: Sum1D example code

4.3.2 Median

`Median1D` elemental function is shown in listing 4.2. It takes `neighbourhood` and `radius` as parameters. It sorts all elements in neighbourhood using quick sort and return the element in the middle. `Median2D` and `Median3D` share the same idea.

```
int findMedian (int *arr, int size) {
      quickSort(arr, 0, size-1);
      if (size%2 != 0) return arr[size/2];
      return (arr[(size-1)/2] + arr[size/2])/2;
}
int stencilkernel (int neighbourhood[], int radius) {
      return findMedian(neighbourhood, radius*2+1);
}
void parallelMedian(std::vector<int> &output, std::vector<int>
    &input) {
  auto stencil = Stencil1D(stencilkernel, RADIUS, PADDING,
      NITERS, NTHREADS);
    stencil(output, input);
}
```

Listing 4.2: Median1D example code

4.3.3 Gaussian Blur 2D

In Gasussian Blur example, elemental function is shown in listing 4.3. `conv` take 2 array pointers and the array size. LHS array and RHS array represents 2D matrices. In this example, LHS is neighbourhood and RHS is Gaussian filter. I implemented them in 1D. Dot product between neighbourhood and filter becomes sum of products of all pairs of elements. Neighbourhood array is built in the same way as `Median2D`. In this example, Gaussian filter is a 6×6 matrix, and it is implemented as an array of size 36. It expects the radius of filter window is always 2.

```
double conv(double *lhs, double *rhs, int size) {
        double sum=0.;
        for (int i=0; i<size; ++i) {
    sum += lhs[i]*rhs[i];
        }
        return sum;
}
double stencilkernel (double neighbourhood[], int radius) {
        return conv(neighbourhood, filter, (2*radius+1)*(2*radius
            +1));
}
void parallelGaussianBlur(std::vector<double> &output, std::
    vector<double> &input) {
  auto stencil2d = Stencil2D(stencilkernel, RADIUS, NROWS,
      NCOLS, PADDING, NITERS, NTHREADS);
  stencil2d(output, input);
}
```

Listing 4.3: Gaussian Blur 2D example code

4.3.4 Heat Transfer 2D

Heat Transfer 2D elemental function is shown in listing 4.4. The filter size is hard coded as 3×3. It returns the average of 4 neighbours. This example application always expects the radius to be 1.

```
double stencilkernel (double neighbourhood[]) {
        double sum = 0;
        sum = neighbourhood[1] + neighbourhood[3] + neighbourhood
            [5] + neighbourhood[7];
        return sum/4.
}
void parallelHeatTransfer(std::vector<double> &output, std::
    vector<double> &input) {
  auto stencil2d = Stencil2D(stencilkernel, RADIUS, NROWS,
      NCOLS, PADDING, NITERS, NTHREADS);
  stencil2d(output, input);
}
```

Listing 4.4: Heat Transfer 2D example code

4.4 Experimental Results

4.4.1 Experiments for Evaluating performance

Figure 4.1-4.16 show the execution time of each example that ran with different number of threads. Speedup, efficiency and cost graphs are included in appendix chapter A. All experiments set boundary decision to wrap-around and data block size to 100. All examples ran for 20 iterations. In all Sum1D, Sum2D, Median1D and Median2D, radius was set to 3; In Sum3D and Median3D, radius was set to 1. Heat Transfer 2D took radius of 1 and Gaussian Blur 2D took radius of 2 as it is explained in subsection 4.3.4 and 4.3.3. All examples took the same amount of items as input but in different dimensions. All 1D examples took $1,000,000$ items as input. 2D examples' input size was 1000×1000. 3D examples' input size was $100 \times 100 \times 100$. There are a few odd things spotted in this set of experiments. For Median1D, Heat Transfer 2D and Gaussian Blur 2D, Pthreads program's execution time does not meet my expectation. For Sum1D and Median1D, Pthreads program runs faster than sequential program. The cause is not clear at this point, and further investigation is needed.

For the rest examples, when Sequential, Pthreads and skeleton programs all run with only one thread, sequential program was the fastest, Pthreads came in the second, skeleton program was the slowest. The difference in execution time of 3 programs with a single thread indicates the amount of overhead created by parallel programs. For example, with one thread running, Sum2D's Pthreads program took almost the same amount of time as sequential program, while skeleton program took several times more. In Median2D, time differences in 3 programs are relatively smaller. This is because median operation takes much more time than sum operation. This makes overheads accounted for a large amount of time in examples with lighter computation, like Sum. Pthreads programs were mostly faster than skeleton programs, though dynamic load balancing is not implemented in Pthreads code. It means that Pthreads code has fewer overheads than skeleton code. There is another reason makes Pthreads programs faster. Because raw threading programming is more flexible, I can further optimise Pthreads program in some cases. For example, in Sum2D, skeleton program has to create a temporary buffer to hold neighbours and perform sum operation, but Pthreads program can accumulate the sum with only one variable. This might explain why Sum2D's Pthreads program took almost the same amount of time as sequential program. As thread number increased, speedup grew but efficiency also dropped rapidly. Though the execution time declined swiftly, time cost increased quadratically as threads number increased.

In some examples, like Median2D and Median3D on James node, Pthreads programs took longer than skeleton programs with more threads running. It might because dynamic load balancing makes skeleton program performs better on computation-heavy examples like Median. Another possible reason is that James node's processor is actually 32-core CPU. It supports hyperthreading which gives 2 threads per core. Hyperthreading duplicates part of resources to makes one core appear like 2 physical cores. Suppose only 32 threads are running on this 32-core hyperthreading CPU, each thread will run on a physical core. When 64 threads are running, each core with 2 hyperthreads running becomes extremely busy. Hyperthreading usually performs better in I/O. However, in our experiments, all examples are heavy in computation but have less I/O.

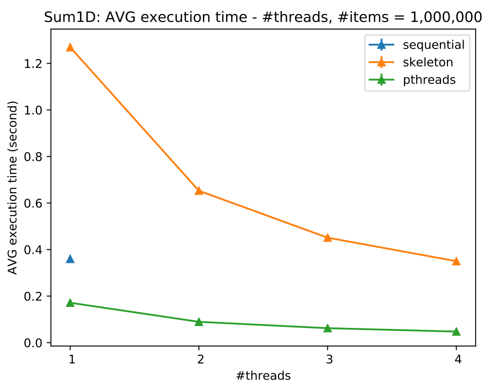

Figure 4.1: Sum1D on student lab desktop

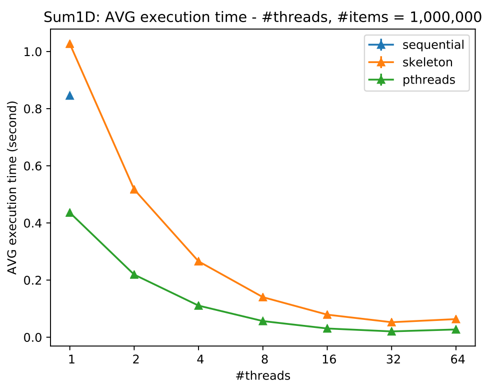

Figure 4.2: Sum1D on James

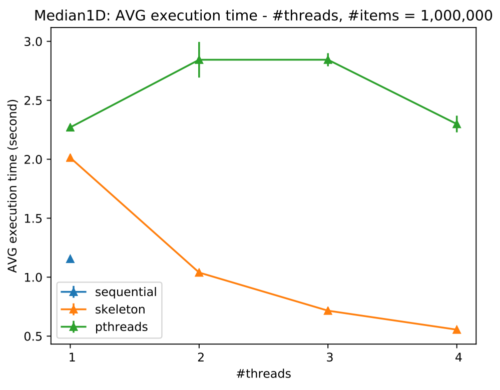

Figure 4.3: Median1D on student lab desktop

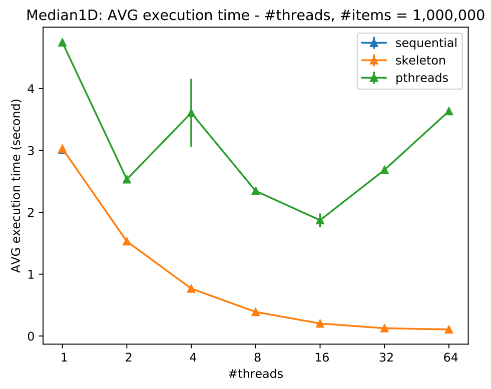

Figure 4.4: Median1D on James

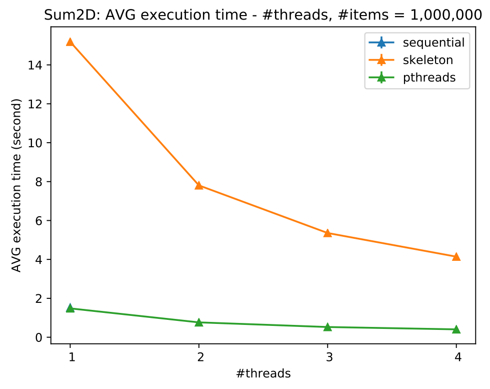

Figure 4.5: Sum2D on student lab desktop

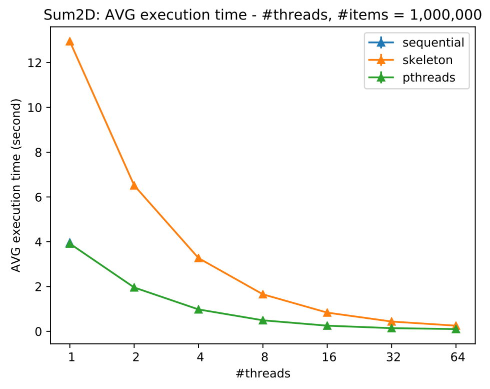

Figure 4.6: Sum2D on James

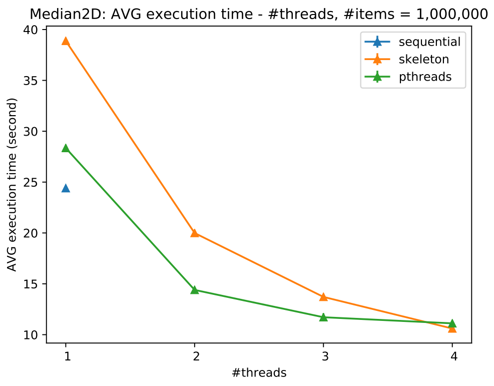

Figure 4.7: Median2D on student lab desktop

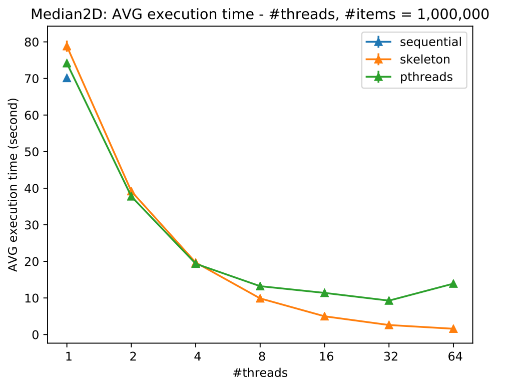

Figure 4.8: Median2D on James

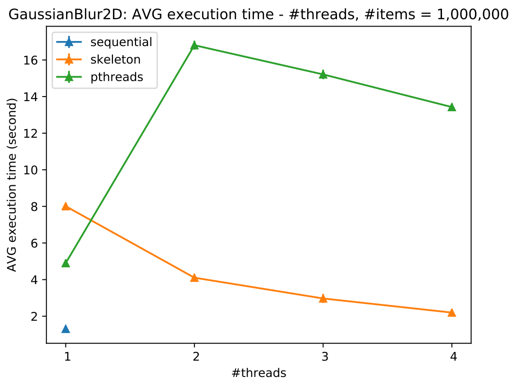

Figure 4.9: GaussianBlur2D on student lab desktop

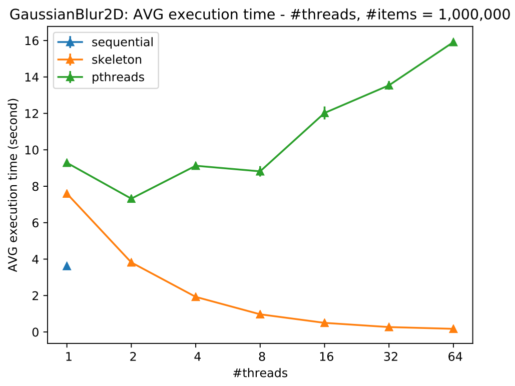

Figure 4.10: GaussianBlur2D on James

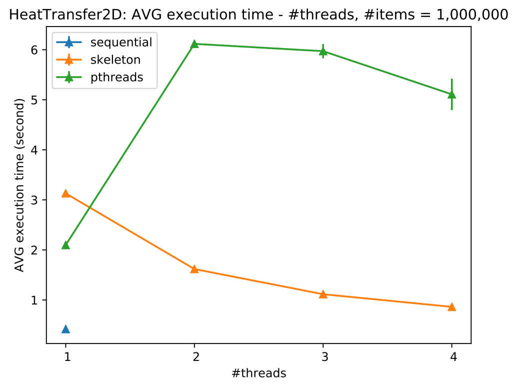

Figure 4.11: HeatTransfer2D on student lab desktop

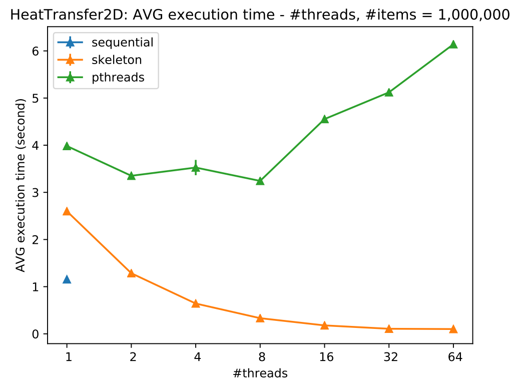

Figure 4.12: HeatTransfer2D on James

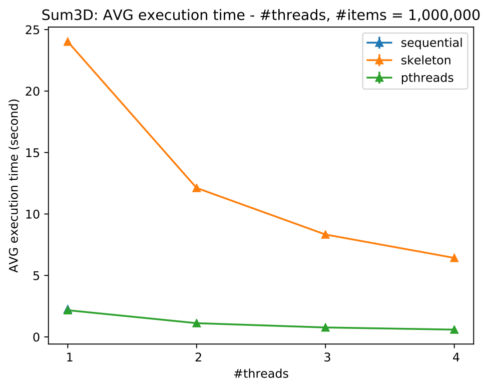

Figure 4.13: Sum3D on student lab desktop

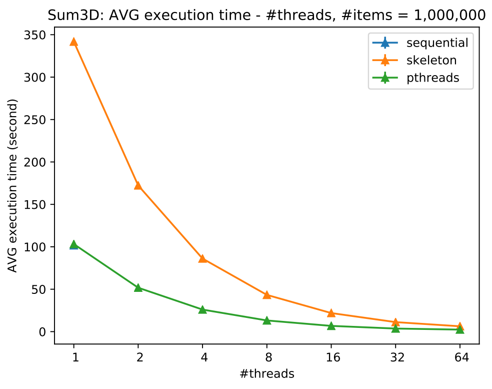

Figure 4.14: Sum3D on James

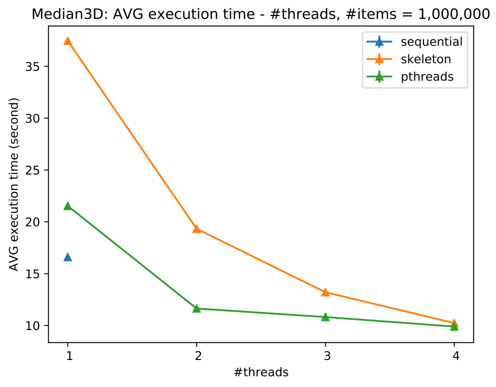

Figure 4.15: Median3D on student lab desktop

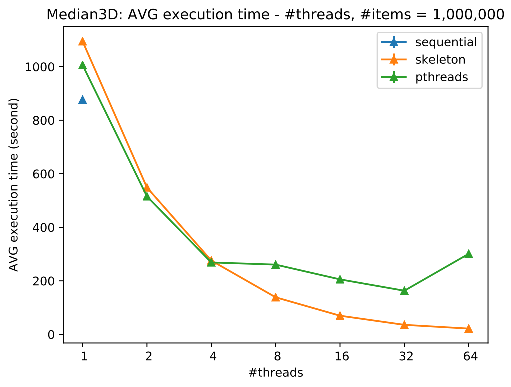

Figure 4.16: Median3D on James

4.4.2 Experiments for Evaluating Scalability

Figure 4.17-4.32 show the execution time of the skeleton program for each example with different problem size. I kept the most settings same as the first set of experiments except for the problem size. I tested each example with 5 problem sizes. All 1D examples started from $1,000,000$ items, then input size increased by $1,000,000$ every time. All 2D examples started from 100×100 items, then row and column number increased by 100 every time. All 3D examples starts from $10 \times 10 \times 10$ items, and each dimension increased by 10 every time. This is why execution time of 1D examples appears to be linear, but execution time of 2D and 3D examples appears to be quadratic. As problem size increased, execution time of programs running with fewer threads rose sharply. Programs with more threads experienced very gentle rise in execution time.

For the same problem size, every time thread number was doubled, execution time was nearly halved.

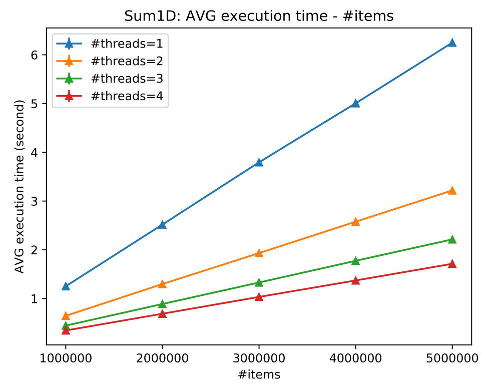

Figure 4.17: Sum1D on student lab desktop

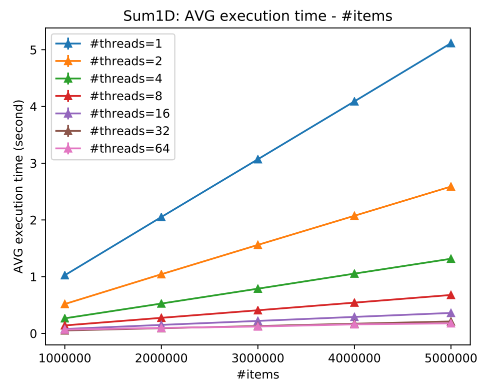

Figure 4.18: Sum1D on James

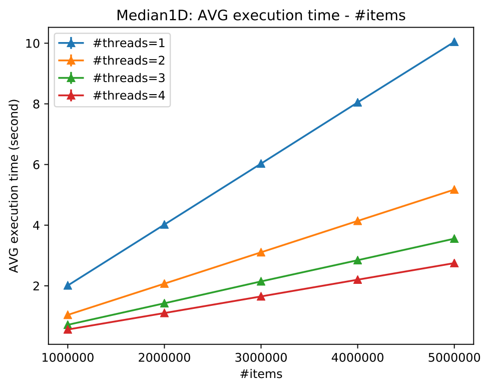

Figure 4.19: Median1D on student lab desktop

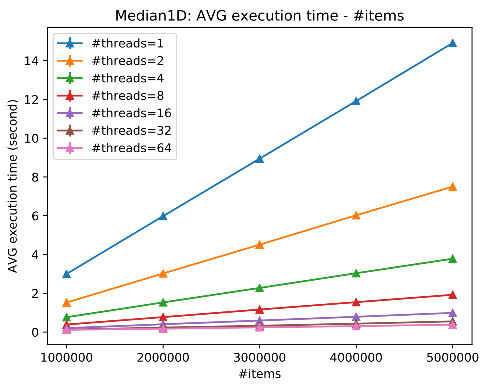

Figure 4.20: Median1D on James

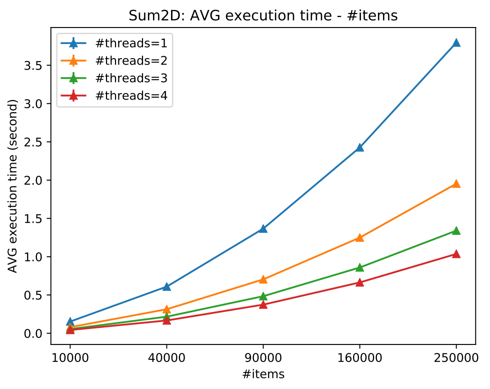

Figure 4.21: Sum2D on student lab desktop

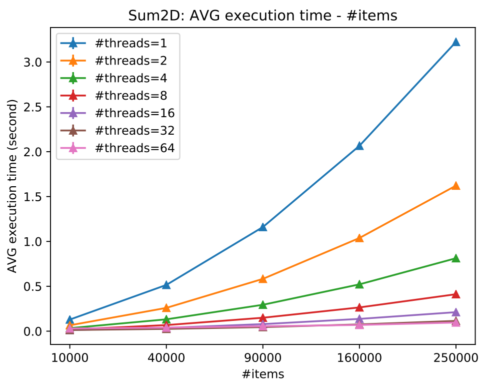

Figure 4.22: Sum2D on James

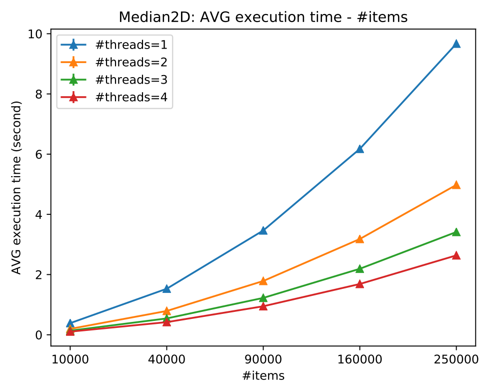

Figure 4.23: Median2D on student lab desktop

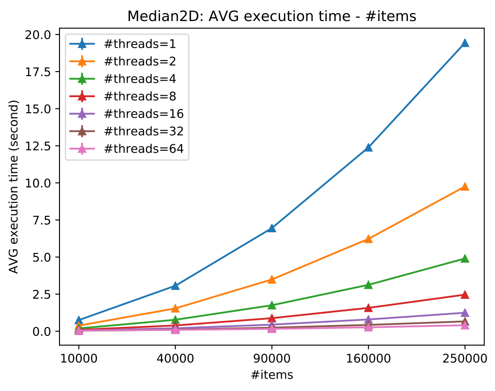

Figure 4.24: Median2D on James

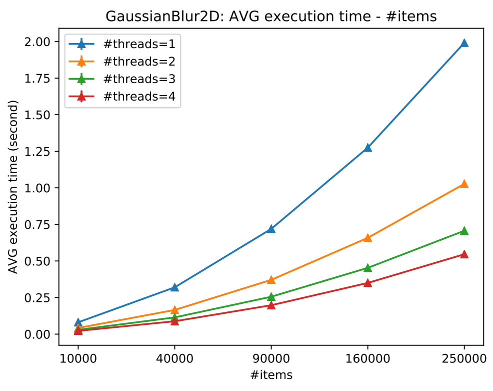

Figure 4.25: GaussianBlur2D on student lab desktop

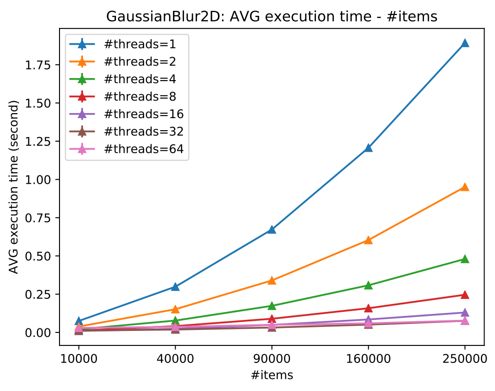

Figure 4.26: GaussianBlur2D on James

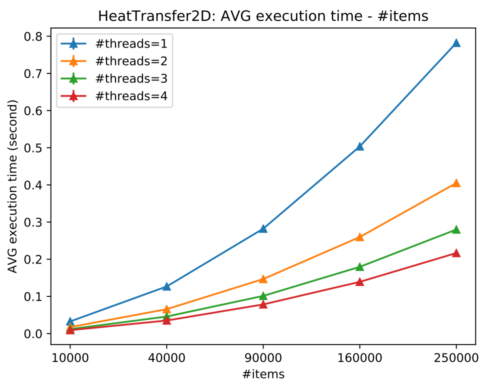

Figure 4.27: HeatTransfer2D on student lab desktop

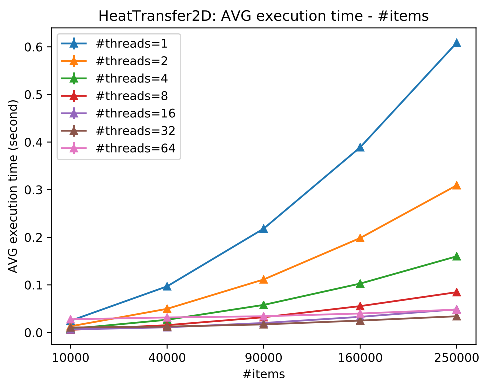

Figure 4.28: HeatTransfer2D on James

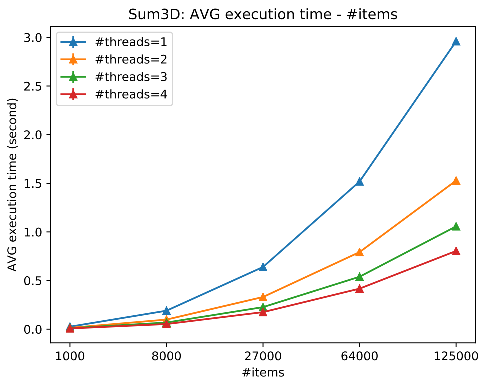

Figure 4.29: Sum3D on student lab desktop

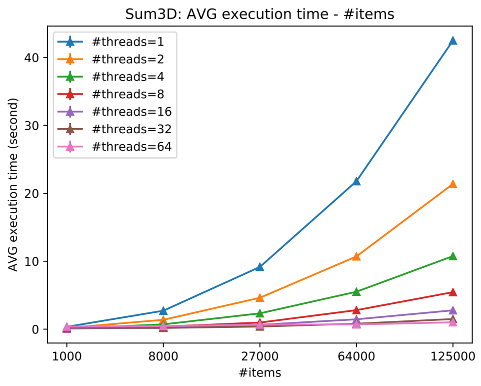

Figure 4.30: Sum3D on James

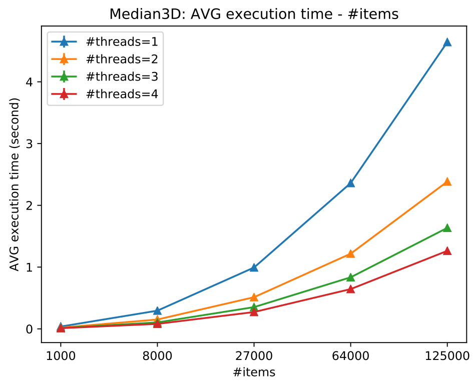

Figure 4.31: Median3D on student lab desktop

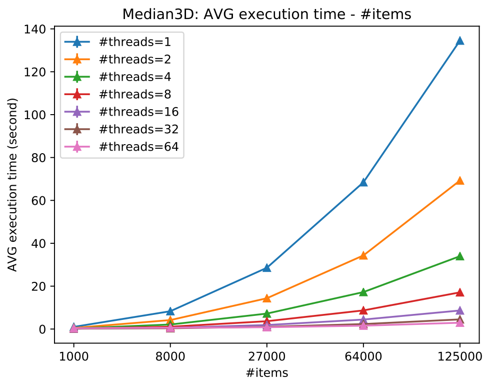

Figure 4.32: Median3D on James

Chapter 5

Conclusion and Future Work

Some parts of this skeleton should be developed further. First, input validation and sanitisation are not implemented. Second, boundary decisions, like closest element and constant value have many if-statements for checking, and `Stencil3D` only has wrap-around implemented. Third, introduction of number of iteration forces the type of input and output vector to be the same. Fourth, because pointers to input and output vector are passed by value to function `threadStencil()`, swapping input and output vector pointer does not swap input and output vector in parenthesis operator. Thus, in this implementation, input and output vector will be swapped in parenthesis operator if iteration number is even. Fifth, stride can be added as another parameter to skeleton interface. In addition, as mentioned in 4.4.1, some examples that gave odd results need further investigation.

Our stencil skeleton needs to be optimised further. Out stencil skeleton uses many nested loops to access neighbourhood. Standard compiler optimisation, such as GCC O2, O3, are not enough to be delegated for optimising loops (Cardoso et al., 2017). Manual code transformation is necessary to improve performance. A common optimisation technique applied to loops is tiling. Tiling reorders data accesses to maximise cache hits. Listing 5.1 is an attempt to tile loops in Sum2D. A special case of loop tiling is known as loop sectioning or strip mining in which only one loop is tiled. An example of strip mining is shown in listing 5.2. In multi-dimensional stencil examples, data is laid out row by row contiguously. For example, in Sum2D, accessing horizontal data is often faster than accessing vertical data because horizontal data that locates nearby tends to be in the same cache line. Figure 5.1 illustrated the cache layout of strip mining. In this way, it optimises cache line data locality. Strips are cache line wide and number of rows high. Tile size should be large enough to avoid reading too much unused data. Choosing the right tile size is the key thing to do. To pick a proper tile size, we need to consider the data size and cache size. I tested tiled programs of Sum1D and Sum2D and compared execution time with non-tiled version, but they did not achieve any performance gain.

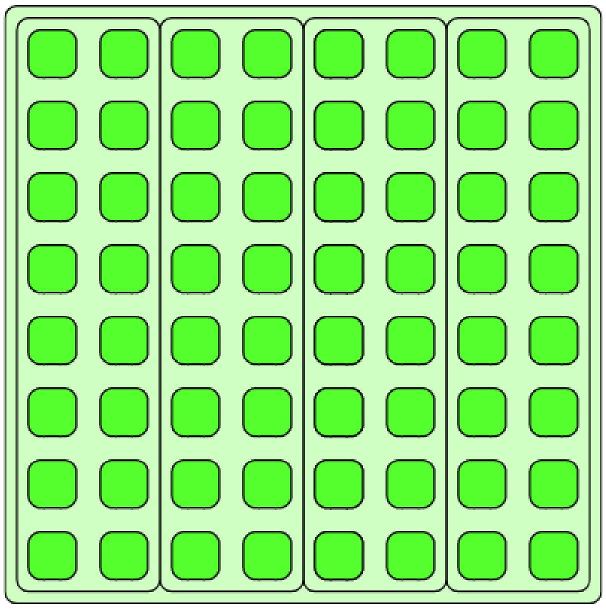

Figure 5.1: Strip mining for cache optimisation (McCool et al., 2012)

```
for (int row=0; row<2*RADIUS+1; row+=TILE) {
      for (int col=0; col<2*RADIUS+1; col+=TILE) {
        for (int rr=row; rr<row+TILE; ++rr) {
              for (int cc=col; cc<col+TILE; ++cc) {
                neighbourCol = (elCol+cc+NCOLS-RADIUS)%NCOLS;
                neighbourRow = (elRow+rr+NROWS-RADIUS)%NROWS;
                sum += input[neighbourCol+neighbourRow*NCOLS];
                    } } } }
```

Listing 5.1: An example of loop tiling

```
for (int row=0; row<2*RADIUS+1; row++) {
      for (int col=0; col<2*RADIUS+1; col+=TILE) {
        for (int cc=col; cc<col+TILE; ++cc) {
              neighbourCol = (elCol+cc+NCOLS-RADIUS)%NCOLS;
              neighbourRow = (elRow+row+NROWS-RADIUS)%NROWS;
              sum += input[neighbourCol+neighbourRow*NCOLS]; }
```

```
                                    }   }
```

Listing 5.2: An example of strip mining

Bibliography

Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C. (2017). Chapter 5 - source code transformations and optimizations. In Cardoso, J. M., Coutinho, J. G. F., and Diniz, P. C., editors, *Embedded Computing for High Performance*, pages 137 – 183. Morgan Kaufmann, Boston.

Ciechanowicz, P., Poldner, M., and Kuchen, H. (2009). The münster skeleton library muesli: A comprehensive overview. Technical report, University of Münster. `https://www.econstor.eu/bitstream/10419/58419/1/717284182.pdf`, accessed July 2019.

Cole, M. (2004). Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406.

Ernstsson, A. (2016). Skepu 2 user guide for the preview release. Technical report, Linköping University. `https://www.ida.liu.se/labs/pelab/skepu/docs/userguide-2_0.pdf`, accessed June 2019.

Ernstsson, A., Li, L., and Kessler, C. (2018). Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80.

Gonzalez-Velez, H. and Leyton, M. (2010). A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw., Pract. Exper.*, 40:1135–1160.

Hammond, K. and Rebón Portillo, Á. J. (2000). Haskskel: Algorithmic skeletons in haskell. In Koopman, P. and Clack, C., editors, *Implementation of Functional Languages*, pages 181–198, Berlin, Heidelberg. Springer Berlin Heidelberg.

McCool, M., Reinders, J., and Robison, A. (2012). *Structured Parallel Programming:*

Patterns for Efficient Computation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Valias, A. (2018). A parallel skeleton library for dice. Master's thesis, University of Edinburgh, UK. `https://project-archive.inf.ed.ac.uk/all/msc/20183037/msc_proj.pdf`, accessed June 2019.

Appendix A

Experiment Plots

A.1 Speedup

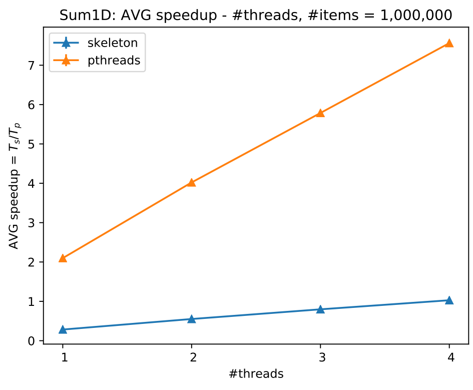

Figure A.1: Sum1D on DICE

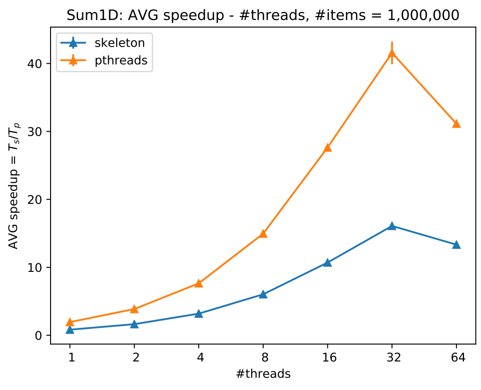

Figure A.2: Sum1D on James

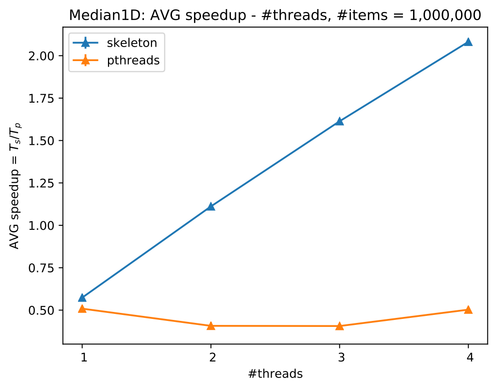

Figure A.3: Median1D on DICE

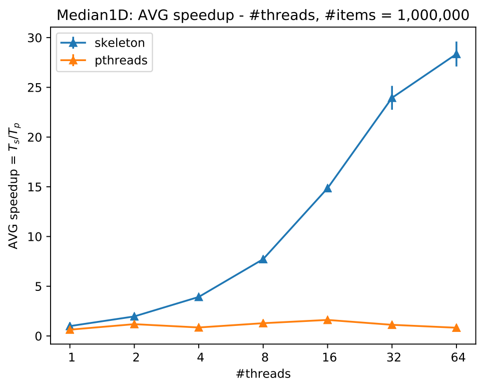

Figure A.4: Median1D on James

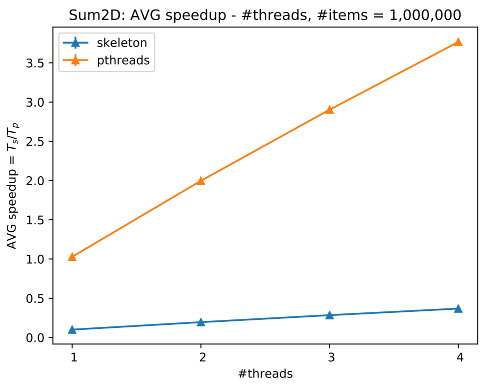

Figure A.5: Sum2D on DICE

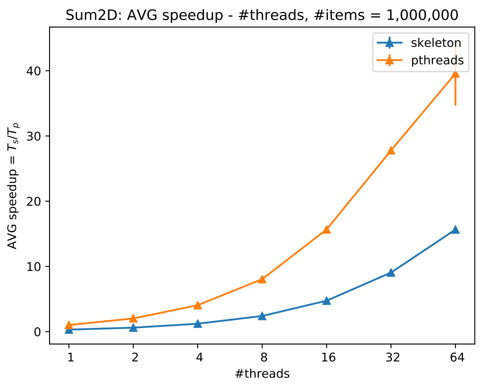

Figure A.6: Sum2D on James

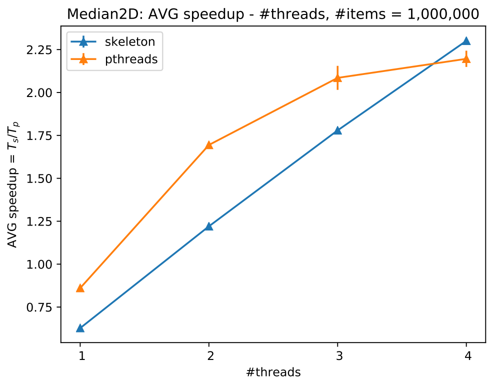

Figure A.7: Median2D on DICE

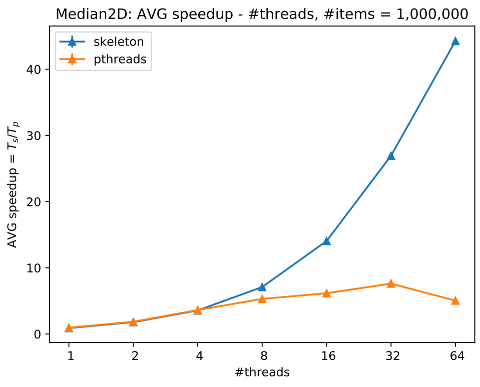

Figure A.8: Median2D on James

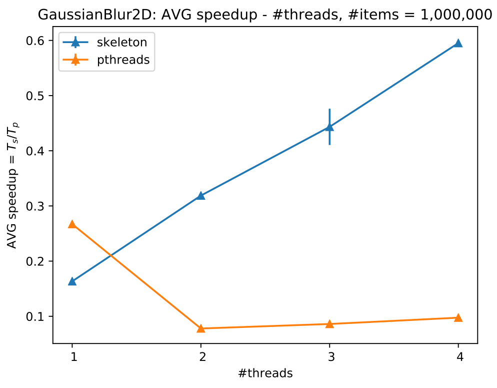

Figure A.9: GaussianBlur2D on DICE

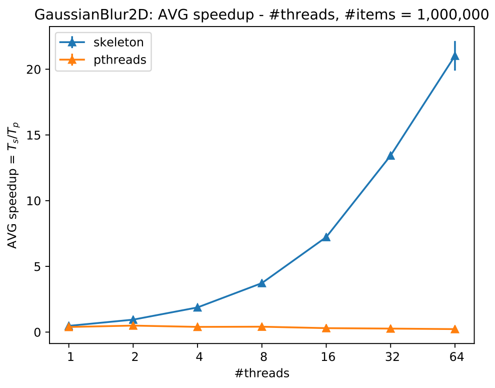

Figure A.10: GaussianBlur2D on James

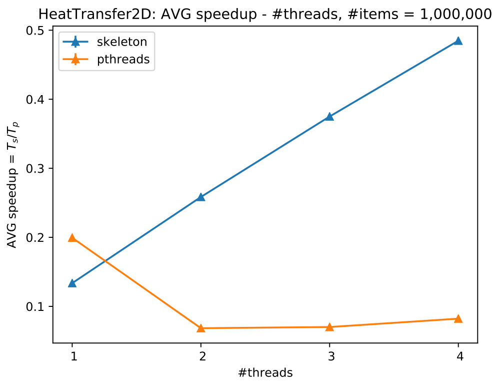

Figure A.11: HeatTransfer2D on DICE

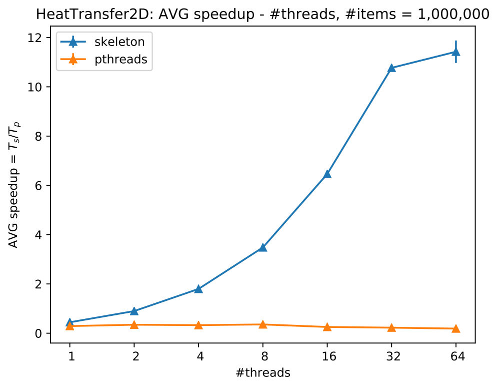

Figure A.12: HeatTransfer2D on James

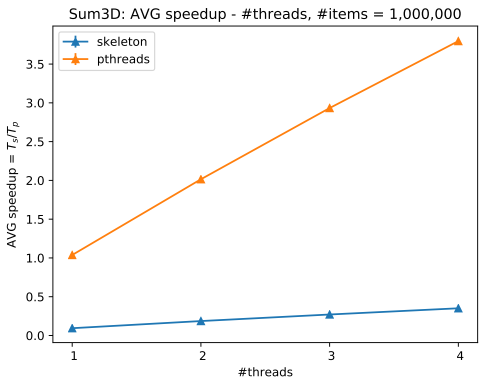

Figure A.13: Sum3D on DICE

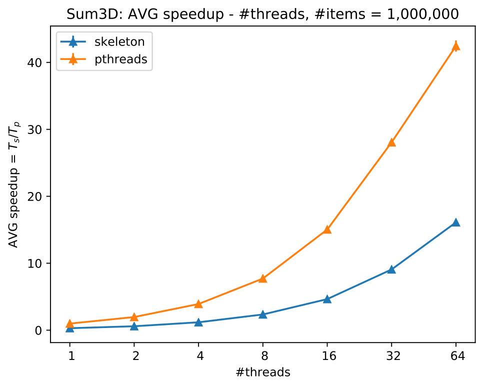

Figure A.14: Sum3D on James

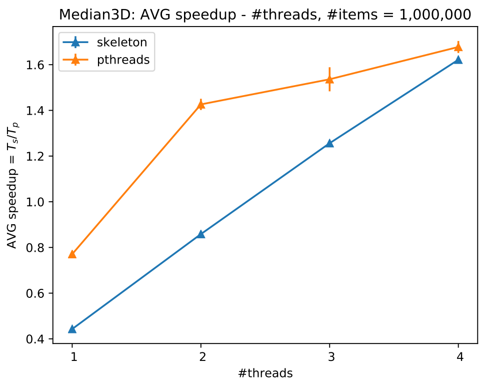

Figure A.15: Median3D on DICE

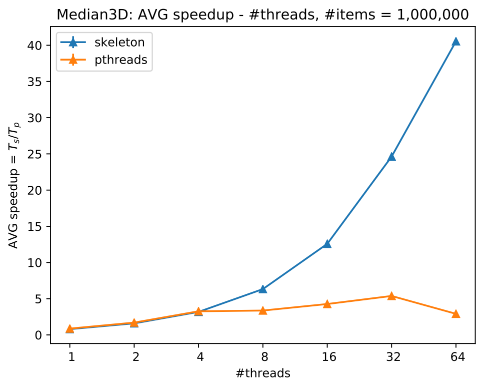

Figure A.16: Median3D on James

A.2 Efficiency

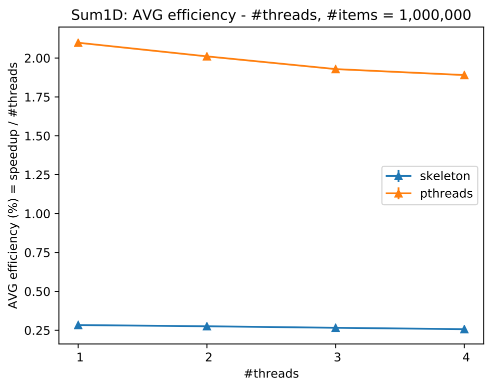

Figure A.17: Sum1D on DICE

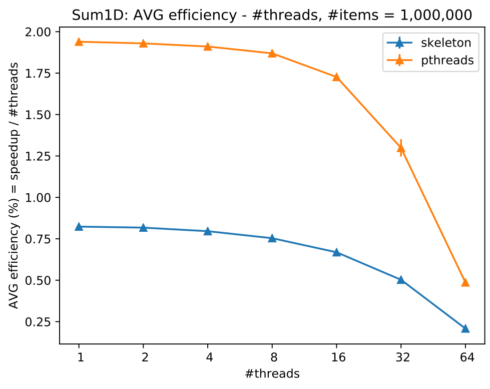

Figure A.18: Sum1D on James

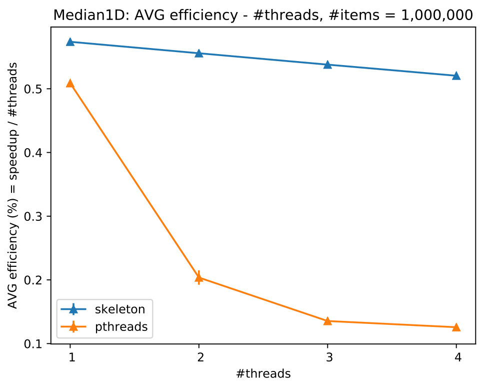

Figure A.19: Median1D on DICE

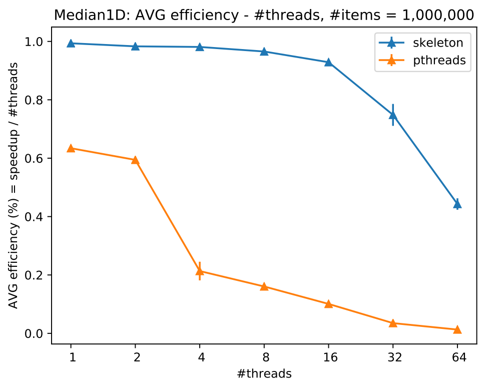

Figure A.20: Median1D on James

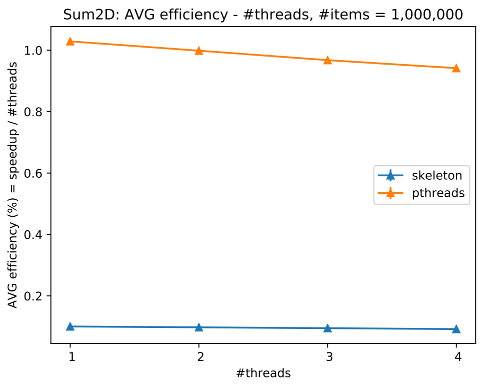

Figure A.21: Sum2D on DICE

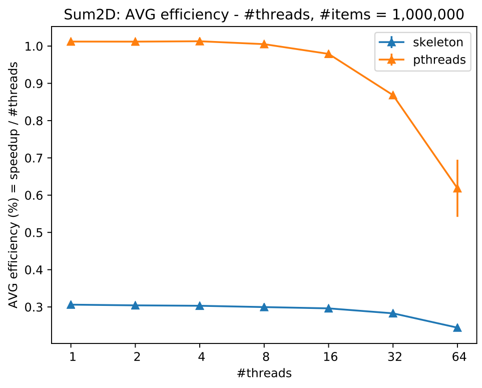

Figure A.22: Sum2D on James

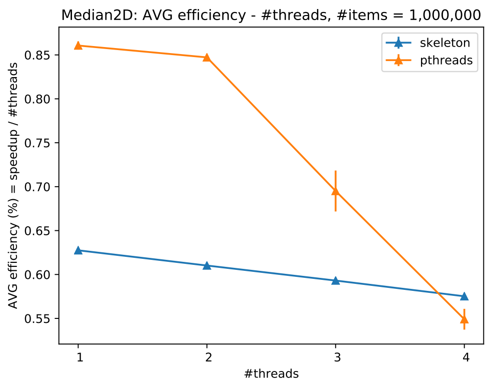

Figure A.23: Median2D on DICE

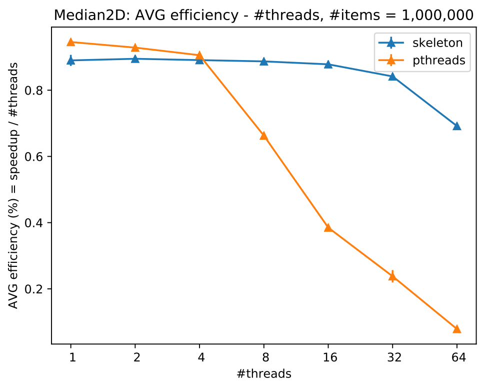

Figure A.24: Median2D on James

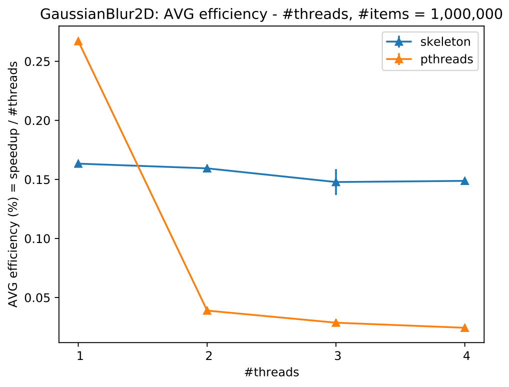

Figure A.25: GaussianBlur2D on DICE

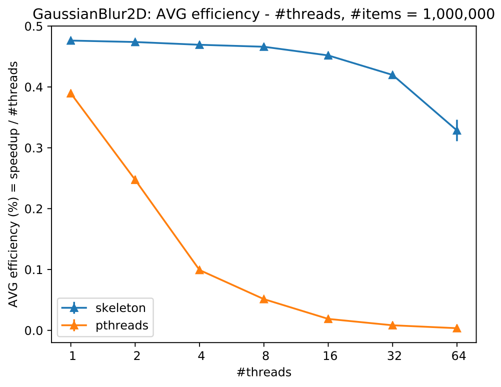

Figure A.26: GaussianBlur2D on James

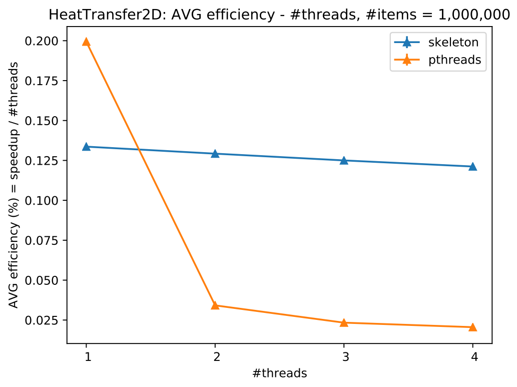

Figure A.27: HeatTransfer2D on DICE

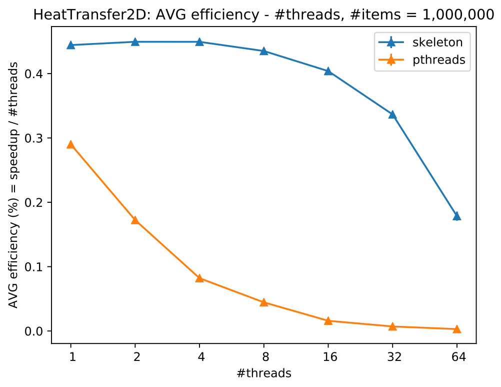

Figure A.28: HeatTransfer2D on James

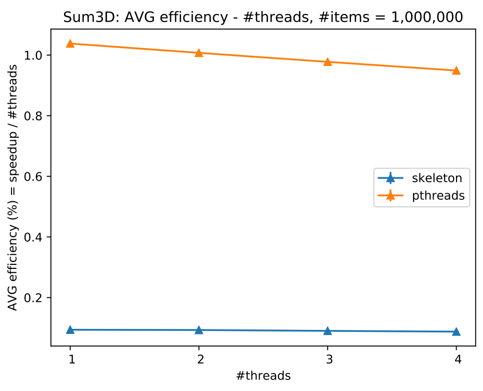

Figure A.29: Sum3D on DICE

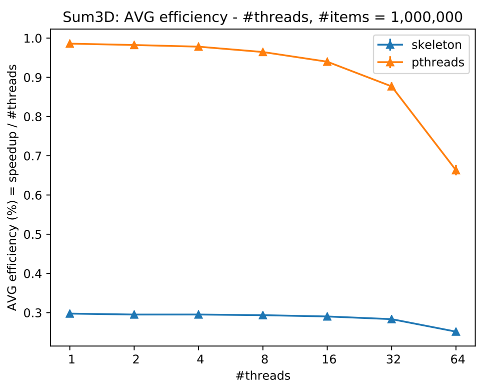

Figure A.30: Sum3D on James

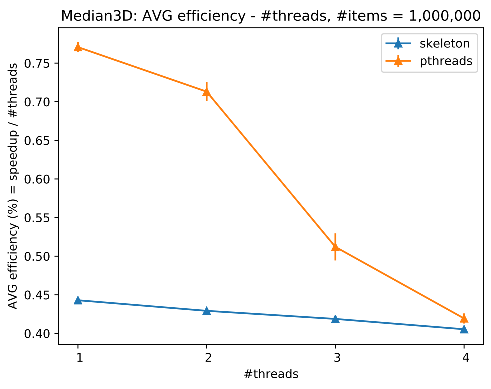

Figure A.31: Median3D on DICE

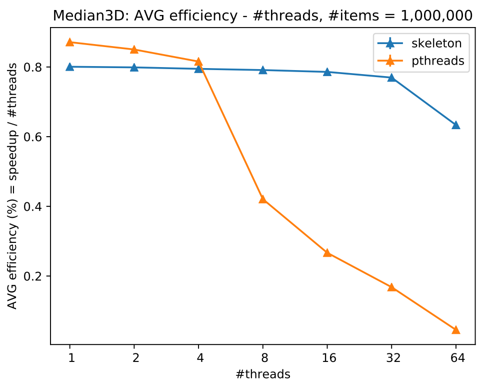

Figure A.32: Median3D on James

A.3 Cost

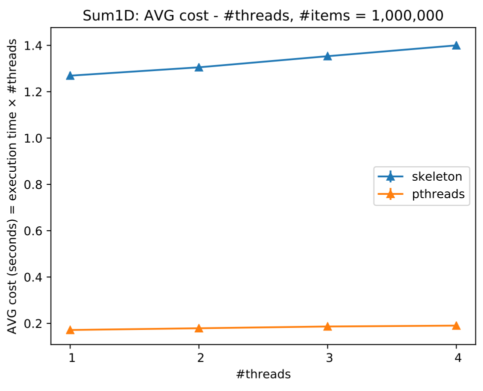

Figure A.33: Sum1D on DICE

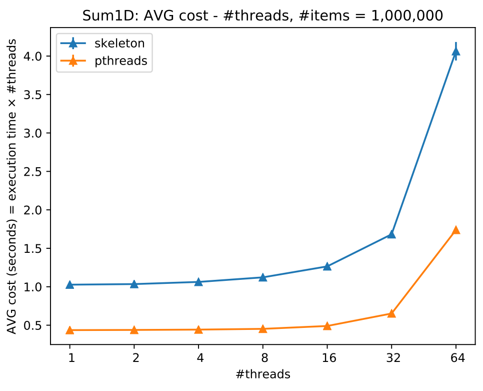

Figure A.34: Sum1D on James

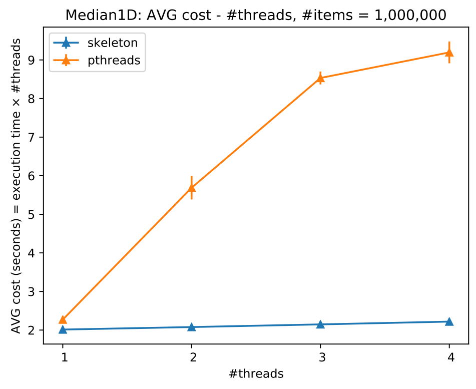

Figure A.35: Median1D on DICE

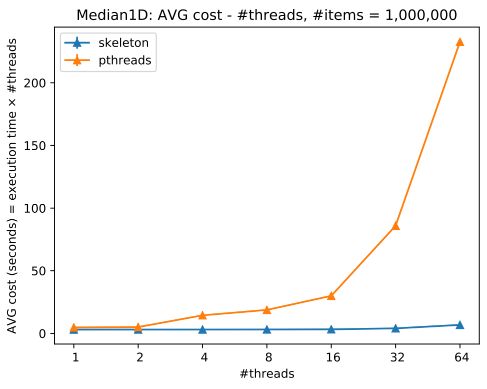

Figure A.36: Median1D on James

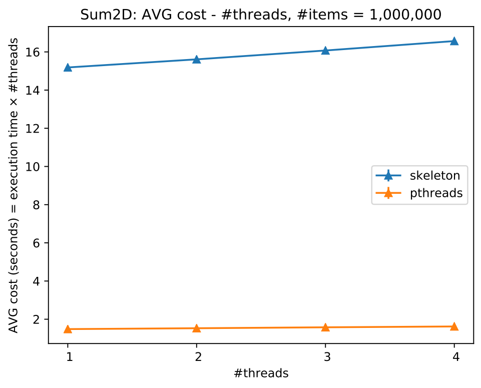

Figure A.37: Sum2D on DICE

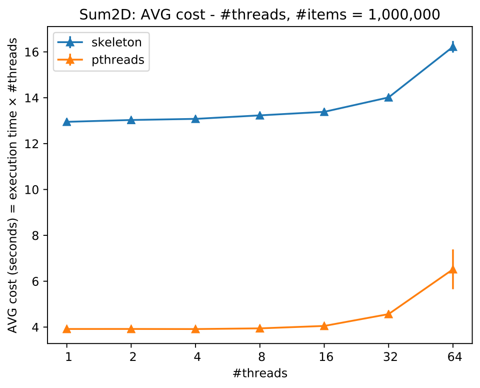

Figure A.38: Sum2D on James

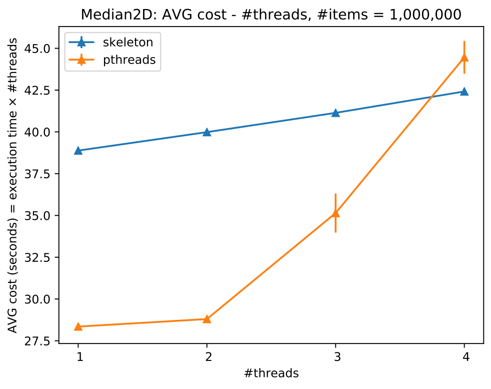

Figure A.39: Median2D on DICE

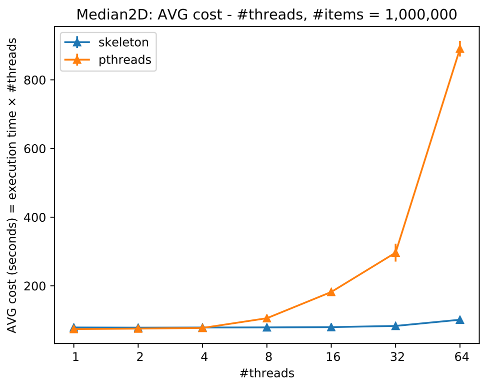

Figure A.40: Median2D on James

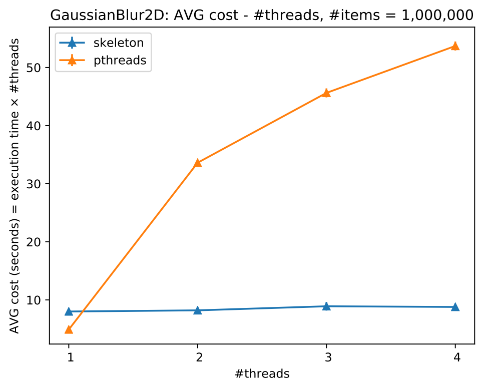

Figure A.41: GaussianBlur2D on DICE

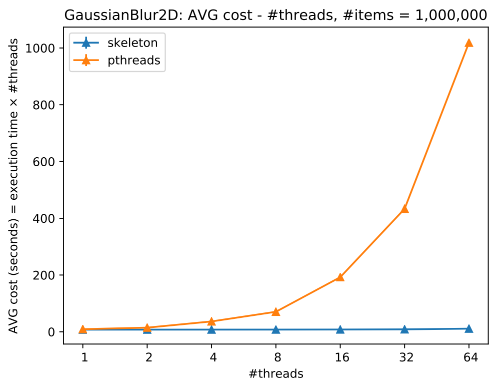

Figure A.42: GaussianBlur2D on James

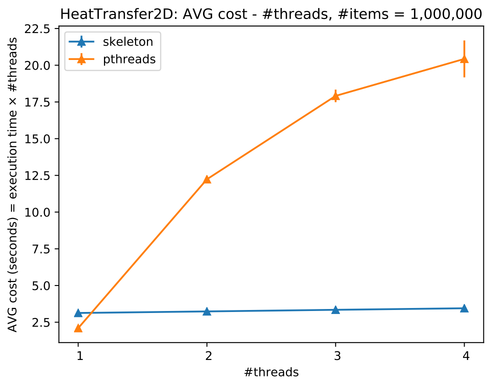

Figure A.43: HeatTransfer2D on DICE

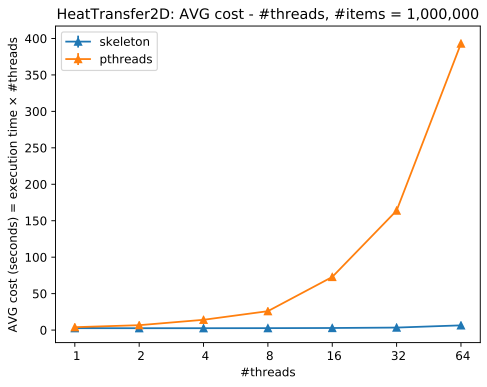

Figure A.44: HeatTransfer2D on James

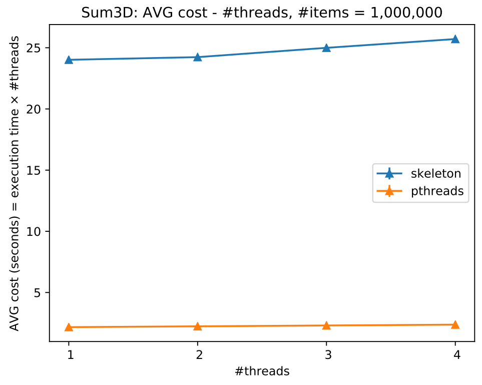

Figure A.45: Sum3D on DICE

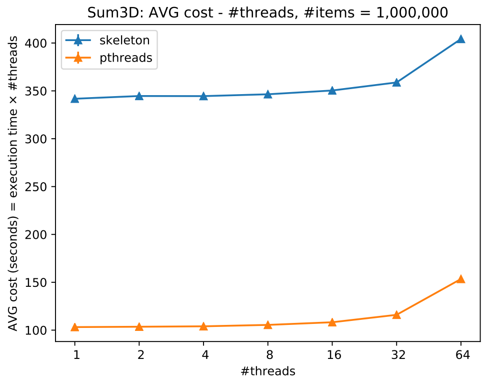

Figure A.46: Sum3D on James

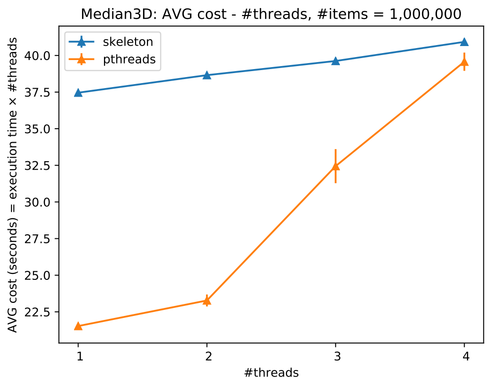

Figure A.47: Median3D on DICE

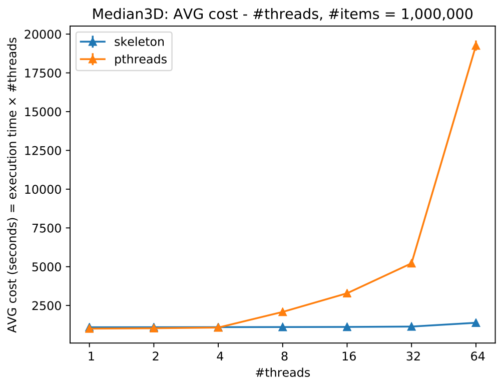

Figure A.48: Median3D on James