UNSYS digital

International Journal of Robotics and Mechatronics

Application of Multithreading Technique for Multiple Input and Multiple Output Mechatronic System

Nur'Ain Saaidon, Wahju Sediono

Department of Mechatronics Engineering, Faculty of Engineering, IIUM, Kuala Lumpur, Malaysia

Abstract— In this paper, multithreading technique is being applied in the system algorithm and evaluated for real-time color marker based navigation system for image guide surgery. This work was carried out to investigate the effect of multithreading technique on the performance of multiple inputs and multiple out system. The purpose of this implementation is to reduce the execution time taken for the image processing algorithms in usual sequential algorithm. It is also to ensure that while the system is executing multiple process of image manipulations, the system navigation unit, robotics arm and the user interface display does not pause any longer than necessary. From the experiment, the graphical user interface for the system with multithreaded algorithm can achieve up to 27 Hz refresh rate and 36.56 msec per cycle with input values being updated in every 10 msec. The result shows that the system is able to track the tool marker and navigate it to the target marker with certain error tolerance. The result also shows that the system has significantly improved the refresh rate of the system's graphical user interface by applying multithreaded algorithm in the system control code compare to normal sequential single thread algorithm.

Keywords— multithreading, image processing, color object detection, tracking

DOI: 10.21535/ijrm.v4i2.1002

I. INTRODUCTION

Mechatronics is a multidisciplinary field of engineering that combines mechanical, electronics, computer science and control engineering [1]. A typical mechatronic system captures signals from the environment, processes them to generate output signals and transforms them into forces, movements and actions. The most important aspect of mechatronics system is the presence of sensors and microcomputers to ensure smooth functioning and higher reliability in the mechanical systems which makes it different from conventional machines.

Mechatronics system can be very simple such as automated gate control, washer, dryer and variation of other smart home appliances to very complicated system such as automated guided vehicles, robots and aircraft flight control and navigation systems. Automated gate control system is one of the examples of simple single input and single output system which the system may react to the infrared sensor from user remote control and open the gate by applying electrical current to the motor. Nowadays, there are a lot of systems that need to control multiple inputs and required to send multiple output signals to complete its designed tasks. In this work, the system is an automated colour marker based navigation system for image guided surgery (IGS) proposed in [2]. The system uses multiple colour markers as its input and produces multiple output signals to the robotics arm and display monitor.

Figure 1: Image-guided surgery system [Adopted from BrainLab.com]

The advances in medical imaging and computing over the last thirty years enabled the use of navigation system in surgery. Medical navigation assists the surgeon to achieve effective and safe surgery by localizing the target and critical lesion that should be avoided, thus provides the motion guidance throughout the surgical procedures [3]. Generally, the components of IGS comprise a camera, a CPU and a display screen. The camera is used to capture the position and orientation of the markers which are mounted on the surgical instruments. Then, the system will process the information and display the anatomy of the patient accordingly on the display screen for the surgeon. Figure 1 shows the target fixer sets to the patient's knee for total knee replacement (TKR) surgery which is the common procedure of IGS. Using the image from the display monitor, the surgeon can easily guide the other tool such as the grinder to prepare the knee for the implant. As robotically assisted surgery enhances the capabilities of surgeon

Corresponding author: Wahju Sediono
(e-mail: wsediono@iium.edu.my)

This paper was submitted on November 30, 2017,
and accepted on December 30, 2017.

in carrying out open surgery or minimally invasive surgery, the use of robotic system in IGS have become a significant development direction in the surgical field.

Instead of using passive infrared (IR) markers, the experiment system is using colour markers to detect the position of the surgical instruments (see Figure 2). Stereo cameras are used to capture the images with the markers in it. By focusing on the colour as the key detection attribute, the system required more complicated process of image manipulation compared to processing the signal from the IR sensor. The system is considered to have multiple inputs due to the needs of more than one colour to be detected during the system execution.

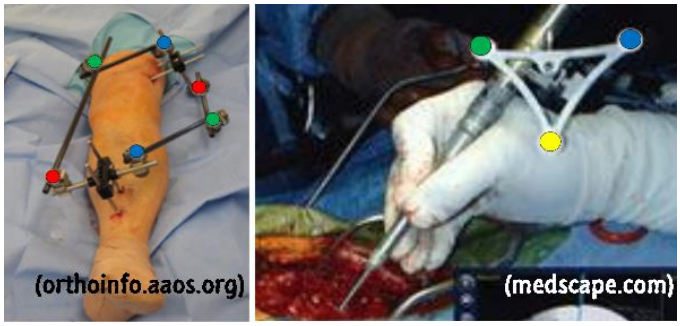

Figure 2: Colour markers on surgical instruments [Adopted from Orthoinfo.aaos.org & Medscape.com]

While current IGS system only displays the processed information and shows it on the monitor to the surgeon, the proposed system also used the information to move the robotic arm. The robotics arm is equipped with another surgical instrument and programmed to pursuit the marker on the patients. Thus, the system is considered to have multiple outputs which are the robotics arm and the display monitoring screen. This system is considered to be real time based application, thus, the processing time per cycle is very important aspect to be considered when designing the system control program.

Image manipulation processing is a good example of the application that benefit from multithreading technique. It involves the process of inspecting image pixels and manipulating them. Image processing can be a time-consuming task based on the matrix structure of the image. Image with high resolution has a bigger number of pixels, which lead to longer processing time for the image [4]. Thus multithreading technique is applied in the system algorithm to reduce the processing time by performing high time-consuming tasks in the background.

II. MULTITHREADING

Multithreading is a type of execution model which allows the main execution thread to detach the long-running or time-consuming task to other execution thread to run in parallel or avoid freezing window for user interaction. Multithreading or parallel programming can be accomplished using the Threading Building Blocks (TBB), offered by Intel, OpenMP (Open Multi-Processing) [5] or using POSIX (Portable Operating System Interface) threads standard [6]. More than ten threads can be generated in a program depending on the capability of the computer processor.

Threading Building Blocks is a standard C++ code library that supports scalable parallel programming. It does not need other special languages or compilers and can be used on any processor or operating system. TBB can run on single core system as well as multicore system using templates for common parallel iteration patterns. TBB allows program designer to achieve increased speed from multiple processor cores without having to be experts in synchronization, load balancing, and cache optimization. The program designer just needs to specify the tasks, not the thread and the library will efficiently map the task onto threads. In general, easier programming, better portability, more understandable source code and better performance and scalability can be expected by avoiding the programming in raw native thread model. [7]. Although TBB offers much easier package, programming with raw native thread model can give more flexibility in system design.

Another method to implement multithreading is by using OpenMP. OpenMP is a language extension involving of pragmas, routines and environment variables for Fortran and C programs. At the beginning, while offering nothing specific for C++ programmer, OpenMP emphases on the loop structures and C code in its programs. The loop structures are the same loop nests that were developed for vector supercomputer. It is an earlier generation of parallel processor that executed incredible amounts of computational work in very tight nests of loops and were programmed frequently in Fortran. Altering those loop nests into parallel code could be very rewarding in terms of the result. In the latter version, OpenMP adding the support for irregular constructs such as while loops and recursive structures instead of exclusively focused on long, regular loop structure. OpenMP gives programmer flexibility to choose either to have static, dynamic or guided scheduling loop iterations compare to TBB.

Another option is by directly used raw thread interface such as POSIX thread (pthreads) or Window threads. Raw threads signify the control of parallelism at its lowest level which offer maximum flexibility but with advanced degree of complexity in term of programmer effort, debugging time and maintenance cost. This method entails the programmer to map up the tasks onto processor cores explicitly which may not suitable for large scale system.

While offering better processing rate, more careful programming designation is required to avoid non-intuitive behaviour during the program execution. The most common non-intuitive behaviours of multithreaded program are racing conditions and deadlocks which related to the shared memory of the program. Multithreaded programming requires access control to the shared data as it is the only way for the threads to communicate. Generally, all threads will use their related variables which are global variables in shared memory in order to make changes in other threads. Shared memory is referred to when two or more unit of parallel task can access data in the same location [8], [9].

Race conditions may occur if commands to read and write data from two different threads are received at almost the same instant. In another word, a race condition happens when two or more threads "race" to access the shared data and at least one of them tried to update the data [10], [11]. It cannot guarantee which thread will get to the data first. As a result, the data can

be unpredictable and errors in reading or writing may arise. The program may crash or shutdown unexpectedly in worst case scenario. Figure 3 illustrates the event of race condition where Thread B overwrites value changed by Thread A due to unsynchronized data access. As can be seen, the last result is not reliable to be used as it is ignoring the process done by Thread A.

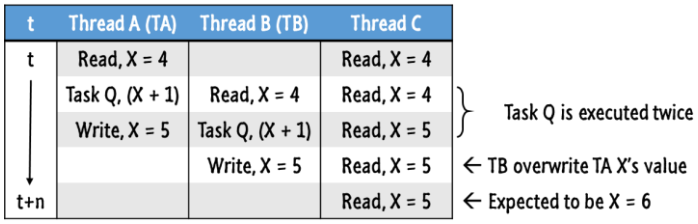

Figure 3: Race condition in multithreaded program

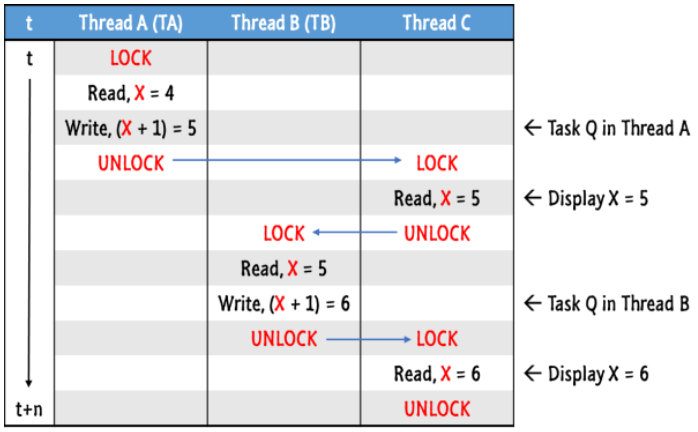

Figure 4: Thread synchronizing with mutex lock

Thus, synchronization between the threads is very important and must be taken into consideration when designing the algorithms in order to have a reliable thread-safe program. The most common solution for race condition is thread synchronizing via 'mutually exclusive' (mutex) locks. Mutex lock limits access of other threads while a thread is using the shared data. Basically, the thread locks the mutex before accessing the shared data and when it is done, it unlocks the mutex [12] as show in Figure 4.

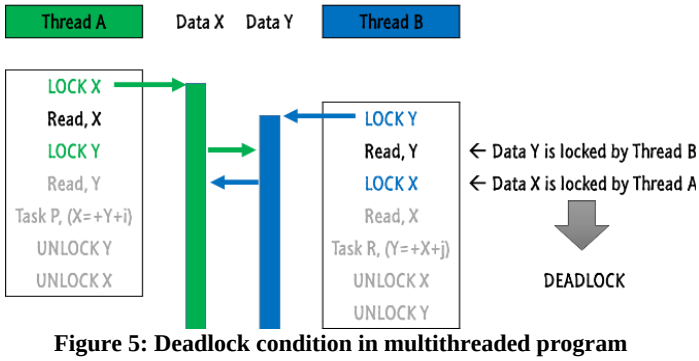

Figure 5: Deadlock condition in multithreaded program

However, the use of synchronized tool can cause a different concurrency fault called the deadlock condition [13]. While race condition makes the programs to do unexpected event, deadlocks can cause the program to just hang infinitely. Deadlock is a situation when one thread waits for a lock from the other thread while holding the lock that is needed by that other thread. The waiting cause perpetual blockage of the program, thus, stops program progress as shown in Figure 5. The program will not take any new input or generating any output and may require to be restarted manually. [13], [14]. Therefore, more consideration on the order of the lock is necessary while designing multithreaded program. Deadlock can be prevented by making the lock statement short in code and time which should last in nanoseconds, not milliseconds.

III. Colour-based Navigation System

The experimented navigation system consists of PC as the main controller, stereo camera, and robotics arm. The images from the camera will be processed through Image Processing Unit (IPU) of the system to filter out the related colour of the markers. The markers are then attached to the tool and target fixer are being identified from the processed image. As the target's markers are moving, the system will detect the movement and send the respective position to the robotics arm to move the tool marker accordingly. The display monitor will shows the position of the tool and target marker accordingly. Figure 6 shows the navigation system's block diagram and Figure 7 shows the flow chart of the system.

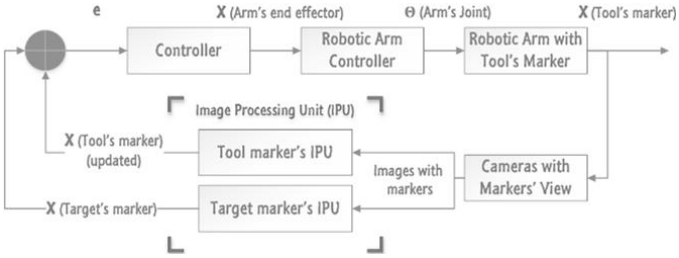

Figure 6: Proposed navigation system's block diagram

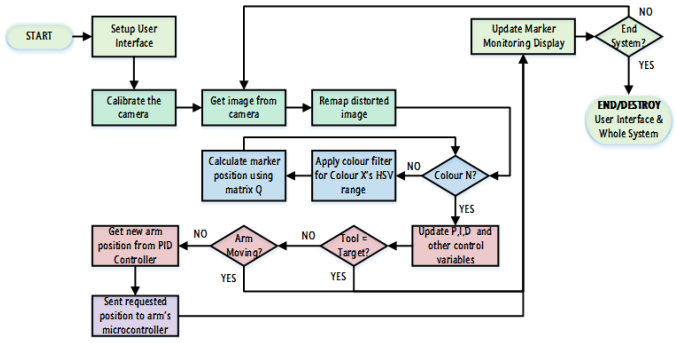

Figure 7: Proposed system's flow chart

Referring to Figure 7, the system will start by setting up the user interface and executing camera calibration which is the input device of the system. The user interface consists of the display panel of the detected markers images, variable initialization panel and a various options of interactions which are available while the system running. After the camera calibration process, the system will start processing the input images for the navigation tool.

The new image retrieved from the camera will be processed in image processing unit for a number of processes to determine

the position of the markers. The hue range values are set to detect the colour markers. The ranges of the HSV value for each of the colour are pre-set in the system. The process of filtering will be repeated N times for N number of marker colour needed. For example, if the tool marker has three different colours and the target-fixer has three other different colours as shown in Figure 2, the same image will undergo six times colour filtering processes. After each of the markers is detected, the actual position of the tool tip and target point will be calculated to define the error value which is the distance between the points.

In this work, a simple controller is used to control the robotic arm in tool navigation. If the position of the tool tip and target point is within user's preset tolerance range, the system will skip the controller unit in order to avoid unnecessary compensation movement. The system will also skip the controller unit if the arm is still moving in order to allow the arm completes the movement for the previous goal position. If the position is not in an acceptable range and the arm already reaches the previous goal position, the error value in X, Y, Z form will be provided to controller unit to determine the next goal position of the arm.

Then, the system will update the live feed of display image and position graph for monitoring purpose. The new arm position will be sent to the robotics arm's Arduino microcontroller board in order to move the tool tip to the target point. Then, the display panel will be updated and the system will start the next cycle by getting the new images from the camera to be processed.

Normally, the system's algorithm will be designed sequentially to the process following the system's flow chart. The sequential algorithms allow remarkably fast and easy development and implementation for the program design. However, the run-time performance can be noticeably low as processing an image multiple times increased the computing time significantly if there are multiple colours to detect.

Alternatively, the parallel processing allows much faster performance with the complexity of implementation and availability of hardware resources [5]. Especially in a real-time application, it is necessary to design the algorithm with fast computing to ensure the reliability of the system's output. Parallel processing can be achieved by implementing multithreading technique in designing the system's algorithm.

As mentioned before, image manipulating processes is very time consuming. Following the system design, the image needs to be filtered out multiple times depending on the number of desired colours for detection. The use of graphical user interface and robotic arm manipulation unit in the system needs more time to refresh the position of the marker. While processing the image throughout multiple filters and image enhancements, the user interface may freeze and the robotic arm may halt at an undesirable position. Thus, to avoid this kind of situations, the system is designed by adopting multithreading technique of execution.

Using the multithreading technique, the process of maintaining the user interface unit, the control of the robotics arm unit and the process of image manipulation are executed in parallel. Each of the units is assigned to different threads and is running simultaneously. Figure 8 shows the processing flow diagram for multithreaded algorithm.

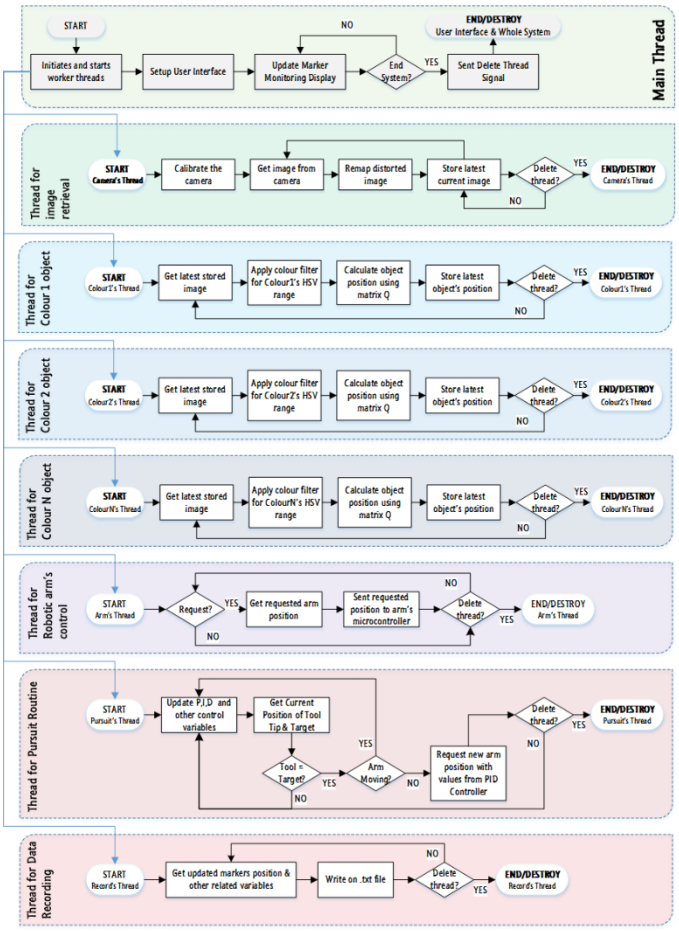

Figure 8: Block diagram of algorithm with multithreading technique

This system has been designed and coded using QT framework. QT framework is a programming platform that offers simple and easy to use API for graphical user interface (GUI) application. However, the libraries in QT framework are not limited to the GUI application, as it also includes the extensive library on event handling and threading related routines. QThread class in QT provides a platform-independent way to manage thread by simply using the ***start()*** function to start worker threads in the main thread. QT offers several classes to support the synchronization between the threads such as QMutex, QReadWriteLock, QSemaphore and QWaitCondition classes in order to avoid race condition or deadlock in the program.

The main thread which is in charge of the user interface related functions will initialize and start the worker threads as shown in Figure 8. The Image retrieval's thread will execute the camera calibration at the beginning of the system execution. It will then continue to retrieve new image from the camera and stores it as a global variable for further processes.

Since the image processing operations are time consuming and do not require any intervention from the user while running, each of the colours filter will be executed in different worker threads. The original image will be stored as a global variable that can be accessed by each of the threads. Each thread will

copy the image, filter it and process the image for its respective colours which correspond to its own specific HSV value range. Using this technique, multiple colour filters are running simultaneously in the background without the need to wait for others to finish. Other than image processing threads, different threads are also assigned to the user interface related class and robotic arm manipulation class. All threads will be destroyed if the main thread sends out the end program signal.

Since there are a lot of variables needed in more than one thread at the same time, the synchronization between the threads is crucial while applying multithreading technique. All the passing variables are set as the global variable and protected by the lock features provided by the QReadWriteLock class from QT library. This lock will prevent the related variables from being access by other threads if the value of the variable is being changed. However, multiple of threads can read the value of the variable at the same time if they are not locked by the write's lock. This is useful as different image processing thread will have a different moment of updating the current variables while the recording thread will store all of the markers' position for the record in every 10 seconds. Other than the record thread, the position of the markers will also be retrieved by the user interface's threads for display purpose and pursuit threads for navigation purpose.

There are several classes that have been developed to accommodate the control element of the system as can be seen in Figure 9. IOCamera class is used to calibrate the camera when the system started at the first time and it will retrieve new images of the markers during the navigation process. ImageProcessing class is responsible for all the image manipulation processes including the colour filtering process. IOArduino class controls data transfer to robotics arm microcontroller. Servo and Arm class are in charge for robotics arm manipulation and positioning control. MainWindow and MyQGLViewer class are representing the control of the user interface functions which mainly in display and variables manipulation for online monitoring.

IV. Experimental Setup

The algorithms are tested in Intel Core i5 processor workstation with 8 GB of RAM memory and Windows 8.1 based operating system. Figure 10 shows the hardware setup of the system experiment. In this designed algorithm, up to five different coloured markers with circular shape were selected for detection and tracking. The hue range values are set to detect the colour of green, yellow, red, violet and blue. The ranges of the HSV value for each of the colour are pre-set in the system.

For the experiment, only one marker is used to specify the position of the tool and another marker for the target-fixer on the navigation system experiment in order to minimize the performance error. The green marker, identified in real life, is acting as the tool marker. The target-fixer marker movements are pre-recorded and replayed during the navigation process. There is a high possibility of collision between the robotic arm and the target marker. Thus, the simulated target point is used in which the movements of the marker are independent from the navigation control.

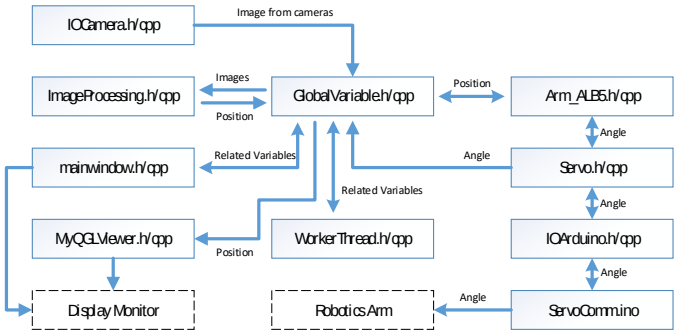

Figure 9: System controller's data flow diagram

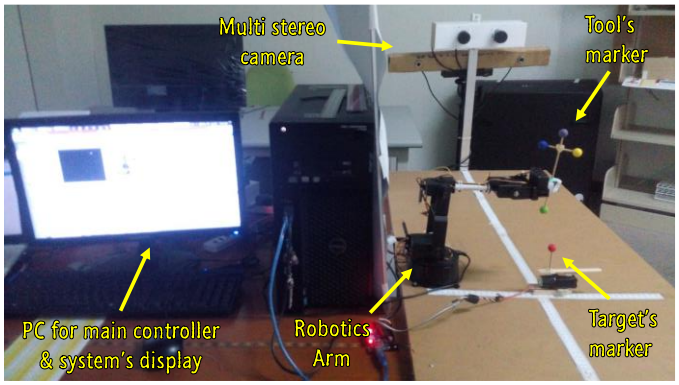

Figure 10: System's hardware setup

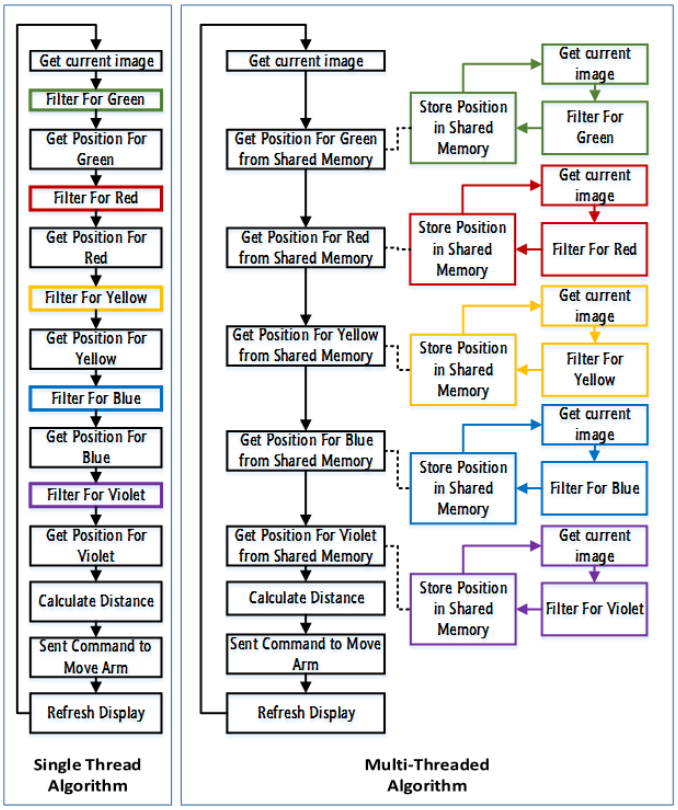

Figure 11: Methods of updating colour markers position

As the main objective of this work is to investigate the effect of multithreaded algorithm on the performance of the system, five different colours will still be processed although only two will considered as the tool and the target marker. Different approaches of collecting input have been tested for all five colour markers as shows in Figure 11.

Figure 11 shows two different methods to update the position of five different colour markers used for the system. In single thread algorithm, the positions are extracted and updated after one another sequentially. In multithreaded algorithm, main thread only retrieves the values from the shared memories without executing any image processing task. In this approach, the algorithm for extracting marker's positon from the image is designed to run independently without attachment to any other threads. Each colour has its own thread and the position of the marker is extracted continuously unrelatedly of other colour filters or process in the system as shown in Figure 11. Independency of the display thread is also important to ensure the user interface and display unit does not freeze while the program is running.

Table 1: Comparison for Different Number of Threads Approaches

	GUI Refresh Rate (Hz)	Average Time per Cycle (msec)	Point Distance Error (mm)
Single Thread Algorithm Single Colour	13.22	75.64	51.91
Single Thread Algorithm Multi Colour	4.33	231.16	108.83
Multithreaded Algorithm Single/Multi Colour	27.36	36.54	44.47

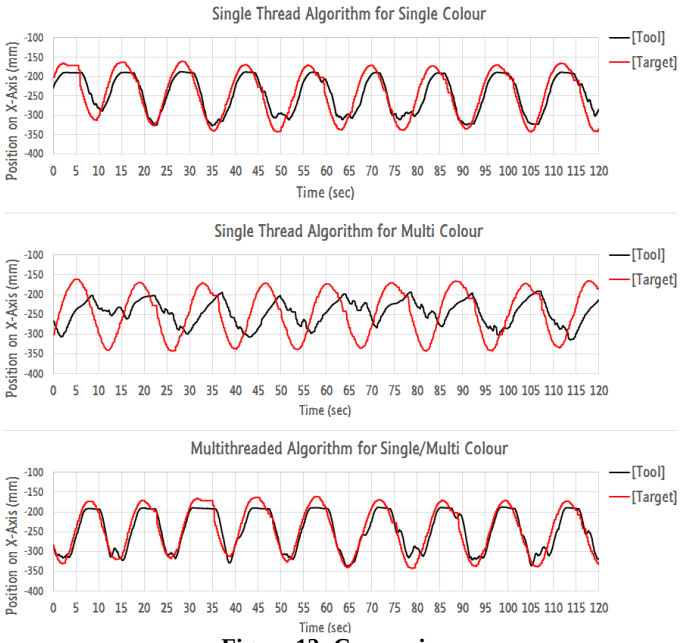

Figure 12: Comparison on different number of threads approaches

Table 1 and Figure 12 show the result of three different approaches of updating markers' position. GUI refresh rate for multicolour object detection is improved significantly with multithreaded algorithm compare to single thread algorithm. Even for detecting single colour, using multithreaded algorithm is faster than using a single thread. When the system is fixed to detect single colour using single thread algorithm, it takes about 75.64 msec to refresh the display unit and take about 231.16 msec for five colours. However, it only needs 36.54 msec in average for multithreaded algorithm regardless of the number of colours to detect. By using this method of threading, the rate will be similar for single or multicolour since the main thread only needs to access shared memory to get the updated values.

Other than the obvious way to improve the system performance which is selecting more advance input and output device with higher precision, improving the update rate of the system can also improve the performance of the navigation control. Slow update rate will result in inaccurate system feedback. The tracking can be lagging as shown in Figure 12, worsening the system's performance by contributing to the points distance error. Though the system is using the same controller, the result may depend on the updating rate of the system. Faster update rate producing more accurate tracking result.

V. Conclusions

This work was carried out to investigate the effect of multithreaded algorithm on the performance of multiple input and multiple out system. In this work, algorithm design for real time detection of colour markers used in the medical navigation system is being designed and evaluated. Colour image processing algorithms for multi-colour object detection were designed using C++ language in Qt on Windows platform. Image processing routine for extracting the position of the markers required a long time compared to other tracking input devices. The presence of display monitoring and robotics arm unit also increase the processing time cycle. Thus, multithreading technique is implemented in the system algorithm. The purpose of this implementation is to reduce the execution time taken for the image processing algorithms without affecting the robotics arm and the user interface display.

The graphical user interface for the system with multithreaded algorithm can achieve up to 27 Hz refresh rate and 36.56 msec per cycle with input values being updated in every 10 msec. The result shows that the system is able to track the tool marker and navigate it to the target marker with certain error tolerance. The result also shows that the system has significantly improved the refresh rate of the system's graphical user interface by applying multithreading technique in the system control code compared to normal sequential single thread algorithm.

ACKNOWLEDGMENT

The work was financially supported by Malaysian Ministry of Education (MOE) through Fundamental Research Grant Scheme (FRGS).

REFERENCES

[1] J.-W. Lee, "Investigation of mechatronic education in South Korea," *Mechatronics*, vol. 20, no. 3, pp. 341–345, Apr. 2010.

[2] N. Saaidon, W. Sediono, and A. Sophian, "Altitude Tracking Using Colour Marker Based Navigation System for Image Guided Surgery," in *Proceedings - 6th International Conference on Computer and Communication Engineering: Innovative Technologies to Serve Humanity, ICCCE 2016*, 2016.

[3] H. Liao, H. Ishihara, H. H. Tran, K. Masamune, I. Sakuma, and T. Dohi, "Precision-guided surgical navigation system using laser guidance and 3D autostereoscopic image overlay.," *Comput. Med. Imaging Graph.*, vol. 34, no. 1, pp. 46–54, Jan. 2010.

[4] A. Kamalakannan and G. Rajamanickam, "High Performance Color Image Processing in Multicore CPU using MFC Multithreading," *Thesai.Org*, vol. 4, no. 12, pp. 42–47, 2013.

[5] B. Cyganek, "Adding parallelism to the hybrid image processing library in multi-threading and multi-core systems," *Proc. - 2011 IEEE 2nd Int. Conf. Networked Embed. Syst. Enterp. Appl. NESEA 2011*, 2011.

[6] S. Matuska, R. Hudec, and M. Benco, "The comparison of CPU time consumption for image processing algorithm in Matlab and OpenCV," *Proc. 9th Int. Conf. ELEKTRO 2012*, pp. 75–78, 2012.

[7] J. Reinders, *Intel Threading Building Blocks Outfitting C ++ for Multi-Core Processor Parallelism*, First Edit. Sebastopol, CA: O'Reilly Media, Inc., 2007.

[8] M. Kwiatkowski, M. Sankowski, and D. Lukwinski, "Computational performance improvements of multiple hypothesis tracking algorithm," *Proc. Int. Radar Symp.*, no. 1, 2014.

[9] A. R. Micheal McCool, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*. Waltham, MA: Elsevier Inc., 2012.

[10] S. He, S. Li, Y. Chen, and D. Guo, "Uncertainty Analysis of Race Conditions in Real-Time Systems," *Proc. - 2015 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS 2015*, pp. 227–232, 2015.

[11] D. Hutchins, A. Ballman, and D. Sutherland, "C/C++ thread safety analysis," *Proc. - 2014 14th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2014*, pp. 41–46, 2014.

[12] C. Yan, "Race condition and concurrency safety of multithreaded object-oriented programming in Java," *IEEE Int. Conf. Syst. Man Cybern.*, vol. 6, 2002.

[13] A. Jyoti and V. Arora, "Debugging and visualization techniques for multithreaded programs: A survey," *Int. Conf. Recent Adv. Innov. Eng. ICRAIE 2014*, pp. 7–12, 2014.

[14] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter, "Sound Static Deadlock Analysis for C/Pthreads (Extended Version)," pp. 379–390, 2016.