AIAA-94-1283-CP

AN ARCHITECTURE FOR REAL-TIME VISION PROCESSING

Chiun-Hong Chien

Intelligent Systems Department
Lockheed Engineering and Sciences Company
2400 NASA Road 1, Houston TX 77058
chien@superman.jsc.nasa.gov

Abstract

This paper proposes an architecture for real time vision processing on parallel processors with physically distributed shared memory, and presents an initial implementation of the architecture on i860-based Mercury Computing Systems. Within the framework of the architecture, each vision function (such as median filtering or contour extraction) is defined as a task or a set of tasks. A collection of these tasks, along with associated data, may be recursively divided into subtasks and processed by multiple processors through the coordination of a task queue server. The task queue server resides in shared memory accessible by all the processors. Each idle processor subsequently fetches a task and associated data from the task queue server for processing and posts the result to the shared memory for later use. In this way load balancing within the parallel processing system can be achieved without a centralized controller, as demonstrated by experimental results.

1. Introduction

It is well known that vision processing involves a tremendous amount of computation. It is even more so for real time vision processing such as vision guided grasping of free-floating objects in space [1], in which processing cannot be carried out in real time without high processing power provided by parallel computers such as Hypercubes [2] or i860-based Mercury Computing Systems [3]. Even with the availability of powerful parallel computers, the real challenge is to find the best strategies for mapping data and vision tasks onto underlying parallel architectures.

A great deal of effort has been directed toward exploring parallelism in pixel-level image processing by taking advantage of its simplicity and data regularity [4]. The success of parallel image processing, however, has not been extended to mid-level feature processing or high-level image understanding. This is due to:

1. simple data partitioning methods do not fit to the mid-level and high-level vision processing, and
2. existing Ethernet-based network discourages dynamic data/task migration.

The advance of VLSI technology in the past decade makes it possible to build tightly-coupled parallel computers with powerful general-purpose microprocessors (such as i860s) interconnected by high bandwidth crossbar switches. Furthermore, research in physically distributed shared memory [5] has reached a stage where the support of physically distributed shared memory model on commercial parallel processing systems becomes a standard, rather an exception. The availability of a powerful parallel processing system with the support of the physically distributed shared memory model has encouraged us to study more powerful methods for data and task partitioning, task allocation, task scheduling, and load balancing, in mid-level feature processing and high-level object recognition and pose estimation. The result of this effort is the design of our proposed architecture for real time vision processing.

The proposed vision architecture has evolved from a vision architecture, known as PARADIGM [6], designed and implemented on NECTAR (a fiber-optics based high-speed network backplane for heterogeneous multi-computers) [7]. PARADIGM was designed to provide mechanisms and primitives that not only allow vision-related parallel programs to be developed with ease, but also maximize program concurrency at both task-level and subtask-level. While developing their programs, users need only focus efforts on problem solving and task partitioning, without

a need to worry about the details of communication and task scheduling. PARADIGM is a distributed system with centralized control. It is composed of a controller and a number of workers which communicate through message passing.

However, with the support of physically distributed shared memory, it deems unnecessary to have a controller for task management (e.g. distribution and scheduling), and it would be less efficient for workers to "communicate" with each other through message passing. Instead of a controller and several communication servers, a task queue server is used both for "coordinating" task execution and for exchanging information. A task queue server is actually a collection of various queues residing in a shared memory buffer accessible by all the processors/workers. Each idle worker subsequently fetches a task and associated data from the task-queue server for processing and posts results to the shared memory buffer for later use. In this way, the proposed architecture is similar to the blackboard architecture employed in Carnegie Mellon's autonomous land vehicle, Navlab [8].

To study the feasibility of using the proposed architecture for real time vision processing, the task queue server has been implemented on an i860-based MC860VS [2]. Parallel algorithms for median filtering and contour extraction have also been designed and implemented on the MC860VS. Timing statistics has been collected and analyzed in order to measure the overhead for task management and to refine the proposed architecture.

The remainder of this paper is organized as follows. Section 2 gives a brief description of the MC860VS. Discussed in Section 3 are the criteria considered in the design of the proposed architecture. Section 4 presents the design of, and functionality provided in the proposed architecture, which is followed by experimental results from an initial implementation in Section 5. Concluding remarks given in Section 6.

2. Mercury Computing Systems MC860

A MC860 is i860-based array processor (AP). A MC860VS board contains four computing elements interconnected by a six-port crossbar switch. Each computing element consists of a 40 MHz i860 processor, a high-performance data switch, a DMA controller, up and to 16 MBytes DRAM (as shown in Figure 1). The MC860 acts as a peripheral to a host computer such as a Sparcstation or a 68040-based vxWorks platform [3].

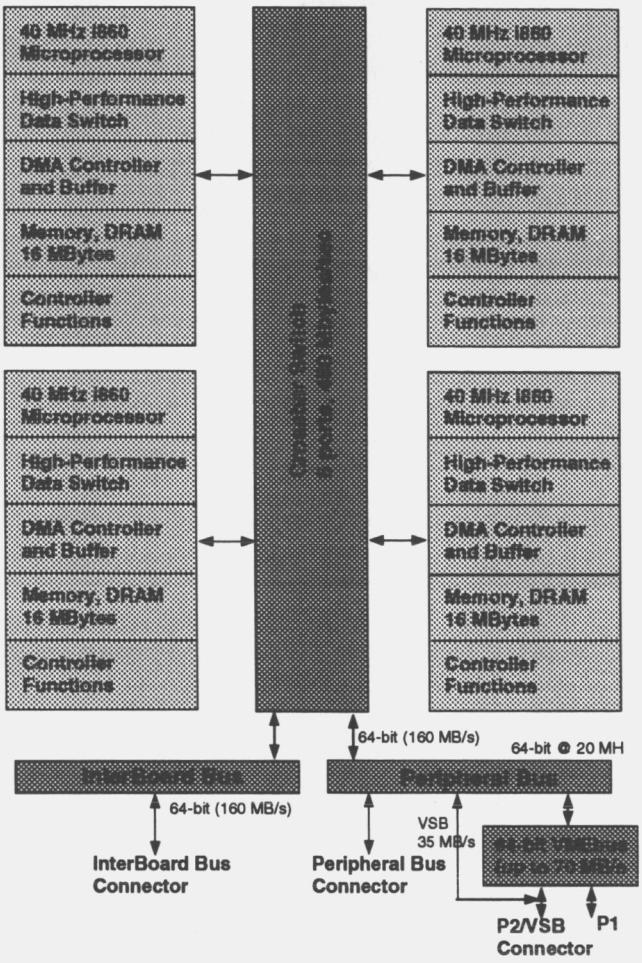

Figure 1: The MC860VS block diagram

The characteristics of computing elements includes

1. Eighty (Sixty) megaflop of computational power for single (double) precision operations.
2. One hundred and sixty megabytes per second transfer rate to AP memory (through a high performance data switch).
3. A 640 Mbytes/sec internal data transfer rate.
4. A 4 Kbytes instruction cache and an 8 Kbytes data cache.
5. A shared memory interface between the host and the MC860.
6. A DMA controller that allows for a memory to memory transfer rate of 160 Mbytes/sec without the involvement of the CPU.

The software supports for MC860s includes

1. An operating system (MC/OS) supports synchronization features such as semaphores and sockets, and others such as multi-tasking, various types of interrupts, and mapping of external memory, among others. It also supports a physically distributed shared memory model for ease of parallel programming.

2. A Scientific Algorithm Library (SAL) and its extended version (ESAL) consist of hundreds of micro-coded primitive functions designed to provide fast memory throughput (and processing speed) by utilizing the data cache (via indirect or direct access) as an extended registers.
3. A rich software development environment in which the user can develop an application in either of three ways as follows:
   - The Application Accelerator (Transparent) approach.
   - The Subroutine Engine approach.
   - The Multicomputer (Attached Processor) approach.

The Transparent approach allows quick testing. The Subroutine Engine approach is suitable for SIMD types of parallel processing. The Multicomputer approach is feasible for MIMD types of parallel processing, and is the approach used for the implementation of our proposed architecture.

3. Design Consideration

This section describes criteria considered in the design of the proposed architecture. We first identify the characteristics of different vision tasks. Vision processing can be roughly divided into three levels: low-level image processing, mid-level feature processing, and high-level image understanding. Their characteristics are as follows:

Low-level Image Processing:

Data items are pixels, which are uniformly distributed in the image space. Operations include simple local or neighborhood operations (e.g. thresholding, filtering, and edge detection) performed on a large amount of data. The inherent parallelism is fine-grained at a pixel level.

Mid-level Feature Processing:

Data items are 2D features such as points, lines, and regions of which the distribution in the image space are not uniform. There are a medium number of data items. Relations among data items are spatial relationships such as adjacency, overlapping and containment. Operations on these data items include unary operations for computing geometric properties, and binary operations involving spatial relationships. The potential parallelism is medium-grained at either feature level or at a level of subsets of spatially adjacent features.

High-level Image Understanding:

Data items are natural or cultural objects or subparts of these objects, which are not uniformly distributed in the image space. Operations include matching between objects (and their subparts) and possible models. The number of data items in general is small, but the number of possible models may be large. The potential parallelism is at the task level or at the level of subsets of the solution space.

As reported in the literature [4] parallel image processing has been successfully applied to real systems such as Warp and Connection Machine. A few systems (such as the CMU Navlab system [9]) have also been developed by exploiting task-level parallelism. However, some tasks are inherently time-consuming and may become bottlenecks during processing if only task-level parallelism is exploited. This problem can be overcome by supporting both task-level and sub-task level parallelism, in addition to pixel-level parallelism, in our proposed vision architecture. Moreover, the heterogeneous nature of different tasks/subtasks and the variation in processing time raise some crucial issues such as task allocation/scheduling, data/task migration, and load balancing, that are not encountered in low-level image processing and scientific computing. The problem is further complicated by the need to handle spatial features in vision processing (spatial-oriented operations, in particular) due to the dimensionality of spatial features, their complex data structures and tangled topological relationships.

In the past, most parallel algorithms have been designed for a single operation at a time. The underlying assumption was that parallel algorithms for multiple operations could be obtained by pipelining those for single operations. This argument might be true for a sequence of local operations (such as low-level image processing) where data distribution is more or less similar for all the operations. However, the argument does not hold in general cases where data distribution varies with individual operations. In these cases, the overhead involved in data redistribution must be taken into account, and therefore the best algorithm for a single operation may no longer be the best choice when it is used along with other operations. In other words, an architecture for parallel processing must allow us to optimize the performance of a set of heterogeneous tasks as a whole, rather than the performance of each individual operation.

To maximize performance, many operations (i.e. tasks in the context of the proposed architecture) should be executed concurrently as long as no temporal ordering (i.e. dependency) exists between them. The task-level parallelism can be realized using a task-queue mechanism (implemented as a *Task Queue* and a set of *Subtask Queues* in this work). In the task-queue mechanism, a task queue and subtask queues are used to keep all the task, that are subsequently assigned to (or fetched by) any idle processor/worker. Task dependency can be handled by a resource mechanism and will be discussed later. To prevent potential bottleneck, any time-consuming task must be divided into smaller tasks by either dividing the task into subtasks (*task partitioning*), or dividing the data set into subsets (*data partitioning*), or a combination of both. A mechanism for *task partitioning* not only allows a vision system to achieve good load balance, but also makes it possible for users to design a complex parallel/distributed program in a hierarchical modular fashion. That is, a program can be recursively divided into modules, submodules, and primitive operations.

A straightforward *data partitioning* method is to partition the data set into many subsets and distribute the subsets to each processor using a task queue mechanism. However, in the domain of vision and spatial oriented processing, partitioning the data regardless of their spatial relationship usually results in the scattering of spatially adjacent data items among the processors which increases the overhead in data migration for tasks that involve operations on spatially adjacent data items. The overhead may be reduced by using shared memory, rather than message passing via sockets, for communication. There are other issues involved in partitioning spatial-oriented data. Readers are referred to [6] for a more detailed discussion.

In summary, the proposed vision architecture should be designed to:

1. support both task/subtask-level parallelism,
2. use task/subtask queues for task management,
3. use a resource mechanism, along with multiple subtask queues, for task scheduling,
4. hide the details of communication and task allocation/scheduling from users,
5. employ shared memory for communication,
6. provide a mechanism for composing parallel vision programs.

A detailed description of the task queue server is given in the following section.

4. Task-Queue Server

The proposed real-time vision processing architecture is similar to the black board architecture. A physically distributed shared memory buffer "accessible" by all the processors is used for storing and fetching tasks (for execution). The block diagram of the proposed vision architecture is shown in Figure 2. It consists of a master, a taskqueue server, and a number of workers which communicate with the master and with each other through a physically distributed shared memory buffer. The master has a set of user defined functions for data/task partitioning, and for post processing (such as merging of partial results from workers). Each worker is facilitated with a set of user defined functions for task execution.

The heart of the proposed architecture is a task-queue server residing in the shared memory buffer. It consists of a task queue, an internal task queue, a number of subtask queues (one associated with each worker/processor), and a reply queue. The task queue stores a sequence of tasks to be executed. These tasks may be recursively divided into subtasks (based on functionality) and queued in the internal task queue. For each task in the internal queue, there is a corresponding task handler (in the module operated by the master) responsible for partitioning the task into a set of smaller subtasks (via *data partitioning*). These subtasks are then "evenly" placed into the subtask queues subject to certain constraints, such as the locality of data on which the tasks will be executed or resource requirements. (e.g I/O devices). The constraints are used to determine the *executors* of the tasks. The reply queue is for storing information regarding the completion of subtasks.

In the following, we shall give more detailed description about important functionality provided by the proposed architecture, including task partitioning, task scheduling and allocation and communication.

Task Partitioning

The principal responsibility of each task handler is to partition tasks. To maximize concurrency (and hence performance), a time-consuming task should be partitioned into a number of smaller subtasks so that the load of the task can be distributed equally over the workers. There are two techniques to partition a task: *task partitioning* and *data partitioning*.

With *task partitioning*, a task is partitioned into a number of smaller tasks which are usually of different

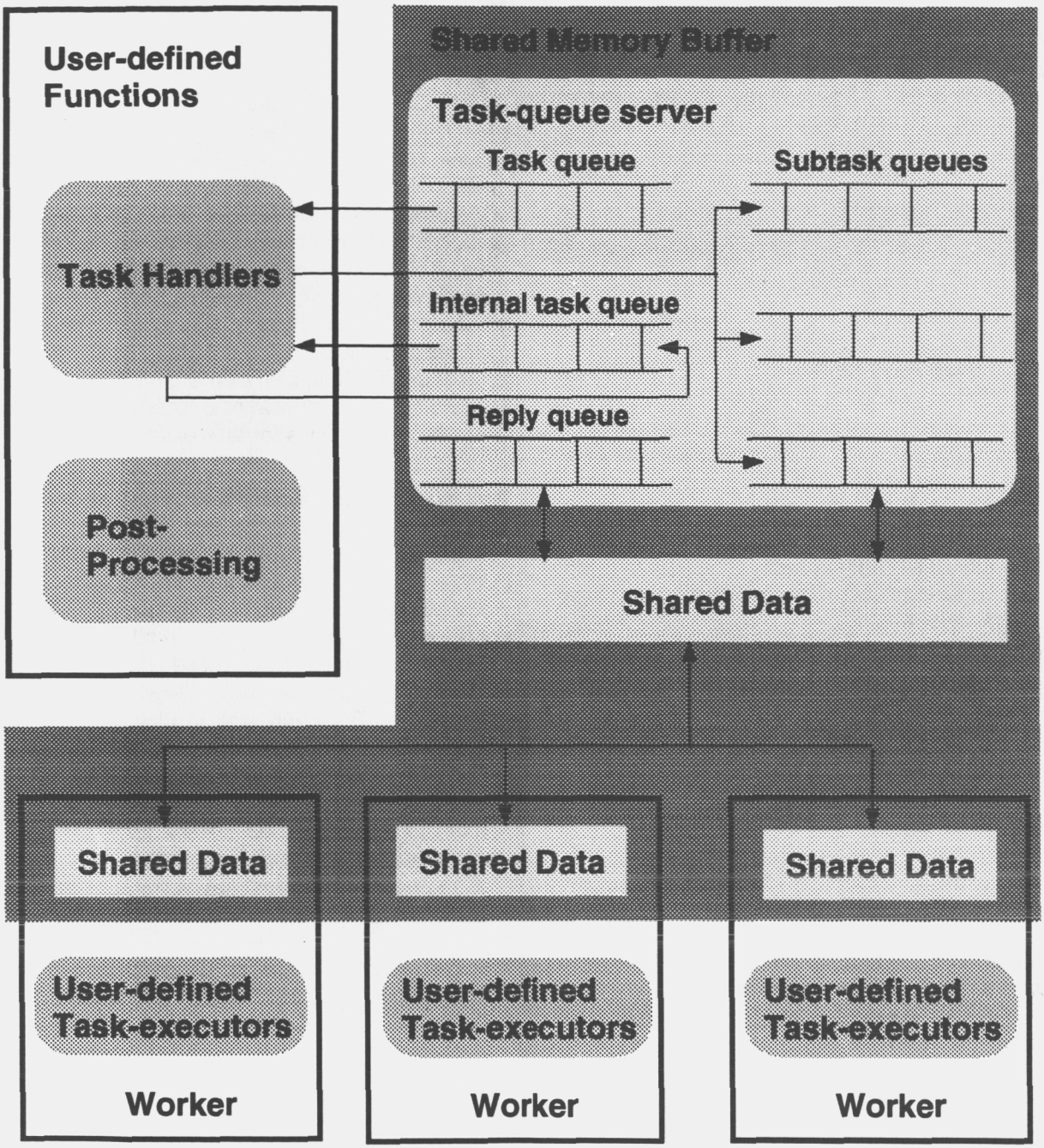

Figure 2: The block digram of the proposed vision architecture

functionality and must be executed in a certain order. For example, a task to extract occluding contours from a noise image may be divided into subtasks for noise reduction, segmentation and contour extraction. The task for contour extraction may not be scheduled until the other two are completed.

Instead of partitioning tasks by functionality, *data partitioning* divides the input data into subsets, which are then processed by workers concurrently and independently. For example, given an image, the task of performing median filtering on the image can be partitioned by dividing the image into subimages, each of which is processed independently by a worker.

To facilitate task and data partitioning, the proposed architecture provides several primitives, including *Create_Task*, *Create_Subtask*, and *Broadcast_Subtask*. The function *Create_Task* creates a new task that is placed in the Internal task queue, waiting to be scheduled. The function *Create_Subtask* creates a

new subtask. A task in the internal task queue may recursively call *Create_Subtask* to create a set of subtasks, that are placed in one or more subtask queues, waiting to be fetched by the associated workers for processing A subtask can be specified as either *worker-dependent* or *worker-independent*. If a subtask is *worker-dependent*, it must be executed by the specified worker. If it is *worker-independent*, it can be executed by any worker, although the specified worker is preferred. A subtask can be marked as *worker-dependent*, when other workers do not have the resource (including data) needed for handling the subtask.

The function *Broadcast_Subtask* allows a subtask to be created and broadcast to all the workers. For example, the task *Task_Exit* is broadcast to all the workers at the end of processing for freeing resources (including memory, sockets, and semaphores).

Task Scheduling

As mentioned earlier, task allocation is carried out by using multiple subtask queues to keep spatial locality, and task scheduling is achieved by using a resource mechanism and a *Depend_On* call. It is important for a distributed system to detect when tasks need competing resources and to schedule tasks to resolve conflicts. In the proposed vision architecture, a resource can be associated with a physical entity (e.g. a display device) or a virtual entity (e.g. data structures). Resources are created with a capacity and tasks can be registered as using a number of resources. A task can be executed only when it needs no resource or the needed resources are available.

By using the resource mechanism properly, synchronization between tasks can often be realized. For example, producer-consumer tasks can be coordinated using the resource mechanism in the following manner. First, the information to be produced by the producer task is registered as a resource of no capacity. When the producer is finished, the capacity of the resource is increased. Therefore, the consumer task, which is registered as needing the resource, will not be scheduled until the producer task is done.

In addition to resource conflicts, there are also task dependencies between different tasks. Task dependency is handled by a function, *Depend_On* (with two tasks as parameters), that constrains a task to be scheduled only when the other task is finished. The function *Depend_On*, together with the resource mechanism, allow task-level synchronization to be specified.

5. Experimental Results

The initial version of the proposed architecture has been implemented on i860-based MC860VS with eight i860 processors, each with 16M memory. In the initial testing, the taskqueue server physically resides on the i860 processor where the master is running. Up to seven workers are running on the same number of i860 processor. The objectives of the initial implementation and testing are as follows:

1. To learn more about the characteristics (i.e strength and limitation) of i860-based MC860VS in the context of real time vision processing.
2. To study the feasibility of using the proposed architecture for real time vision processing.
3. To measure the overhead involved in using the task queue server for parallel processing. The amount of overhead will in turn be a guide line for determining granularity of parallelism used for real-time vision processing.
4. To study the efforts involved in composing parallel programs for vision processing using functions provided by the proposed architecture.

To achieve these objectives, parallel algorithms for several different types of vision operations have been implemented on the MC860VS including algorithms for median filtering and for contour extraction. Timing statistics for running these parallel algorithms on MC860VS have been collected for analysis.

Median filtering is an pixel-level operation, and is easily parallelizable. A typical approach is to divide the image (to be filtered) into N subimages. Each of the N subimages, along with the median filtering operation forms a subtask to be processed by a worker. No explicit merging operation is required when all the subtasks are processed.

On the other hand, contour extraction involves conversion of data structures. i.e. the conversion of a pixel-level representation (image) into a feature or features (contours). One of the approaches to parallel contour extraction is to divide the image into a number of subimages, and to extract a partial contour from each subimage. It is not a "regular" operation in a sense that processing time on each subimage depends on the complexity/shape of the contour in the subimage. It is not a local operation, either, since an explicit merging operation is required to merge the partial contours extracted from the subimages to obtain the complete contour(s).

Experiments for collecting timing statistics were repeated for different numbers of i860 processors (from one to four), for different numbers of subtasks (i.e. 1, 2, 3, 4, 6, 8, 12, 16, and 24, respectively).

Figure 3 shows timing statistics on parallel median filtering (on 256x256 images). For cases where only a single i860 was used, processing time remains relatively the same regardless of the number of subtasks. It implies that the overhead for task management is negligible (when the granularity of parallelism is in the order of tens of milli-seconds) if all the subtasks are processed by a single processor. For cases where two i860s were used, processing time was unusually high (annotated as A2 in Figure 3) when the median filtering operation was divided into three subtasks. This was due to load imbalance. That is, one i860 had one subtask to process while the other had two. The same argument is applied to unusually high processing time annotated as A3 and A4 in Figure 3. It is interesting to point out that processing time at C2 in the figure is slightly higher than B2 and D2. This can be explained as follows. The size of each image on which median filtering was performed is 256x256. The numbers of subtasks associated with B2 and D2 are 8 and 16, respectively. That is, the image was partitioned into subimages of the same size in each of the two cases. Load balance was achieved in these cases. On the other hand, the number of subtasks associated with C2 is 12 by which 256 is not dividable. As a result, load balance was more difficult to achieve since some subimages had larger sizes than the others and took longer time to process.

Load unbalance can also be observed at B3 and D3. The numbers of subtasks in these two cases are 8 and 16, respectively, which are not dividable by 3, the number of processors.

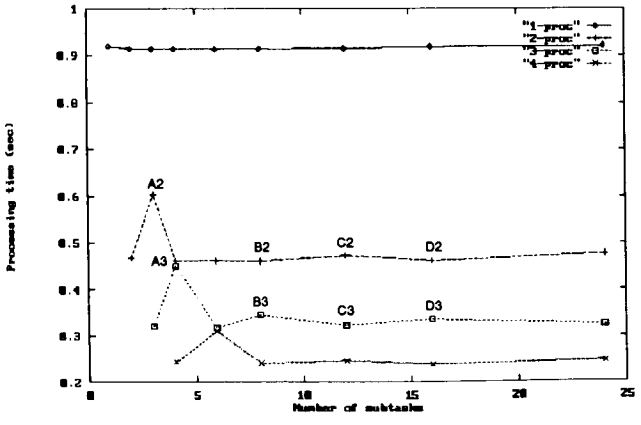

Figure 3: Timing statistics of running parallel median filtering on MC860VS

A different characteristic in timing statistics on parallel contour extraction (as shown in Figure 4) can be observed. For example, in the cases where only a single i860 was used for contour extraction, processing time increased with the number of subtasks. The increase in processing time is not due to the overhead in task management, but due to that in merging the partial contours extracted by each subtask to obtain the complete contour. The overhead is so significant that not much speedup could be obtained by increasing the number of processors beyond three. For the cases where the number of processors is three (or four), the processing time decreases, reaches a minimum, and then increases as the number of subtasks increases. This is due to the fact that a large number of subtasks (running on a relatively small number of processors) will smooth out variation in (and so decrease) processing time for extracting partial contours, but increase time for the merging operation.

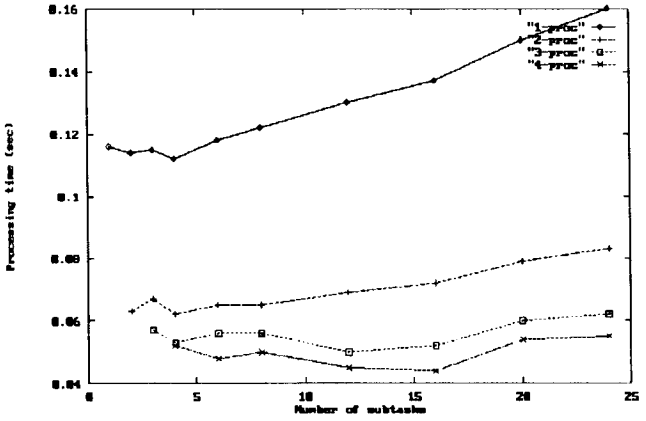

Figure 4: Timing statistics of running paralle contour extraction on up to four MC860VS

To investigate the overhead due to task management in a multi-processor environment, another experiment was conducted to measure speedup factors by running parallel median filtering on one to seven i860 processors. In this experiment, the number of subtasks was set equal to the numbers of processors so that load imbalance would not affect speedup calculation. The results are shown in Figure 5. The solid curve indicate the upper bound for speedup. In an ideal case where there is absolutely no overhead in task management and communication, the processing speed is supposed to increase linearly with the number of processors utilized for processing. The dash curve shows actual speedup. It can be seen that speedup is nearly linear when the number of processors is less than four. The performance of the system gradually degrade as the number of processors increases. The degradation is probably due

to (1) contention among processors for accessing various queues in the task queue server, and (2) overhead in inter-board communication.

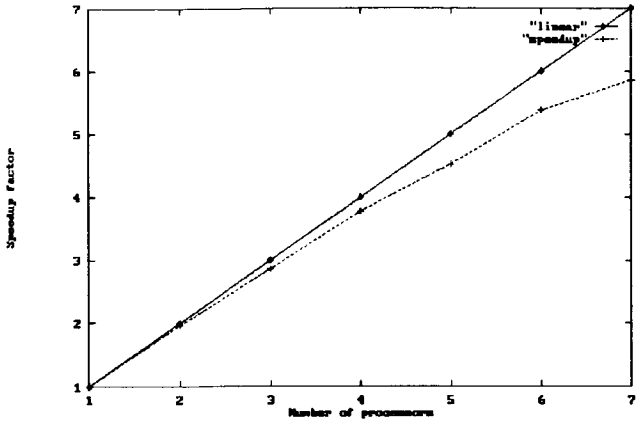

Figure 5: Speedup factors of runing parallel median filtering on MC860VS

6. Concluding Remarks

The task queue server and parallel algorithms for two vision operations have been designed and implemented on an i860-based MC860VS. Timing statistics has been collected to analyze the overhead for task management (including queue-access control and inter-board communication). For local operations which do not require merging operations, such as median filtering, nearly linear speedup can be obtained for a small number of processors and processor utilization (efficiency) gradually degrades as the number of processors increase. For any global operation which requires an additional operation to merge partial results, such as contour extraction, it may not be a good idea to divide the operation into a large number of subtasks.

Based on experimental results from initial implementation, it is expected that the proposed vision architecture will provide a convenient mechanism for composing efficient parallel and distributed programs (vision programs in particular). However, it should be pointed out that a good architecture is necessary but not sufficient for achieving real-time vision processing. For example, the first target application of the vision architecture is vision guided grasping of free flying objects in space by the ExtraVehicular Activity Helper and Retriever (EVAHR) [1]. In order to assist real-time grasping, EVAHR's vision module is required to provide poses (i.e. the orientations and locations) of to-be-grasped-objects to its arm/hand controller at 10 Hz (i.e. 0.1 second per pose estimation) [10]. Median filtering and contour extraction are part of preprocessing before pose estimation can be performed. Experimental results seem to indicate that it may take more than 0.14 seconds to perform only median filtering with 7 processors. This problem can be alleviated by applying median filtering only to subimages from which accurate information needs to be extracted. In other words, real-time vision processing cannot be achieved without efficient (sequential and parallel) vision algorithms and a good parallel vision architecture (along with powerful parallel processors).

References

[1] Grimm, K., Erickson, J., Anderson, G., Chien, C. H., Hewgill, L., Littlefield, M. and Norsworthy, N. "An experiment in vision-based autonomous grasping within a reduced gravity environment," in *Proc. SPIE Conf. on Cooperative Intelligent Robotics in Space III*, Nov. 1992, Boston, MA.

[2] Hypercube, Intel Corportation.

[3] *Training guide for the MC860 Attached Processor*, Mercury Computing Systems.

[4] Weems, Charles "The DARPA image understanding benchmark for parallel computers," Tech. Report 90-98, Dept. of Computer and Infor. Science, Univ. of Massachusetts, 1990.

[5] Bershad, B. N. and Zekauskas, M, "Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors," Tech. Report CMU-CS-91-170, School of Computer Science, Carnegie Mellon Univ., 1991.

[6] Chien, C. H, and Lin, L. J. "PARADIGM: an architecture for distributed vision processing," in *Proc. of IEEE Conference on Pattern Recognition*, June 16-21, Atlantic City, NJ., pp. 648-653.

[7] Arnould, E., Bitz, F, Cooper, E., Kung, H. T., Samson, R., and Steenkiste, P. "The design of Nectar: A network backplane for heterogeneous multicomputers," TR-CMU-CS-89-101, School of Computer Science, Carnegie Mellon, Jan. 1989.

[8] Shafer, S., Stentz, A. and Thorpe, C. "An architecture for sensor fusion in a mobile robot," Tech. Report CMU-RI-TR-86-9, Robotics Institute, Carnegie Mellon Univ., 1986.

[9] Thorpe, C., Hebert, M., Kanade, T., and Shafer, S. "Vision and navigation for the Carnegie-Mellon Navlab, *IEEE Transaction on Pattern Analysis and Machine Intelligence*, Vol. 10, 3, May 1988.

[10] Chien, C. H "Multi-view based pose estimation from range images," in *Proc. SPIE Conference on Cooperative Intelligent Robotics in Space III*, November 1992, Boston, MA.