Real-Time Computational Needs of a Multisensor Feature-Based Range-Estimation Method

Raymond E. Suorsa, Banavar Sridhar and Terrence W. Fong*
NASA Ames Research Center
Moffett Field, CA 94035
suorsa@windchime.arc.nasa.gov

Abstract

The computer vision literature describes many methods to perform obstacle detection and avoidance for autonomous or semi-autonomous vehicles. Methods may be broadly categorized into field-based techniques and feature-based techniques. Field-based techniques have the advantage of regular computational structure at every pixel throughout the image plane. Feature-based techniques are much more data driven in that computational complexity increases dramatically in regions of the image populated by features. It is widely believed that to run computer vision algorithms in real time a parallel architecture is necessary. Field-based techniques lend themselves to easy parallelization due to their regular computational needs. However, we have found that field-based methods are sensitive to noise and have traditionally been difficult to generalize to arbitrary vehicle motion. Therefore, we have sought techniques to parallelize feature-based methods. This paper describes the computational needs of a parallel feature-based range-estimation method developed by NASA Ames. Issues of processing-element performance, load balancing, and data-flow bandwidth are addressed along with a performance review of two architectures on which the feature-based method has been implemented.

1 Introduction

The design of intelligent low-altitude guidance systems for helicopters requires information about objects in the vicinity of the flight path of the vehicle. The sensor system on the helicopter must be able to detect objects such as buildings, trees, poles and wires during flight. A complete obstacle-detection system may consist of an active ranging sensor and passive ranging using electro-optical sensors. A comprehensive overview of this problem can be found in [1, 2].

Several techniques have been proposed for range determination using electro-optical sensors [3, 4, 5]. These techniques use optical flow resulting from the relative motion between the sensor and objects on the ground together with the helicopter state from an inertial navigation system to compute range to various objects in a scene. One algorithm of interest can detect, track and estimate range to image features (i.e., patches of an image with common statistics or spatial structure) over time from a multisensor system mounted on a vehicle moving with arbitrary six degrees of freedom [6].

This paper is a continuation of a previous work which described, at length, a parallel version of our feature-based range-estimation method (hereafter referred to as *Optflow*) [7]. Within that work *Optflow* was ported to a distributed computer (based on a network of eight workstations) and a multithreaded shared-memory multicomputer (a Silicon Graphics IRIS 4D/480).

In this paper we follow the two paths of shared and distributed memory once again but with two different architectures: the iWarp systolic mesh, and Silicon Graphic's latest generation of shared-memory multicomputers, the Onyx. In this light we take a closer look at the performance necessary for real-time computation of *Optflow*. Section 2 of this paper presents a brief overview of the fundamentals of the feature tracking algorithm, focusing on the parallel constructs and load balancing scheme. Section 3 describes the performance

*Recom Technologies, Inc. San Jose, CA. *terry@ptolemy.arc.nasa.gov*

of the the iWarp distributed memory multicomputer and the Silicon Graphics Onyx. This is followed by a concluding section which summarizes performance issues and important architectural differences.

2 Feature Tracking Algorithm

The range-estimation algorithm is a modified version of area-based matching. The algorithm tracks small regions of motion imagery (features) over time and uses an extended Kalman filter to estimate the feature's range. Knowledge of sensor velocity, angular rate and the estimated range from the Kalman filter is used to limit the search space.

The matching is currently performed using normalized correlation [6]. This method, though computationally intensive, gives accurate results and is immune to linear changes in scene brightness. The major computational portion of the feature-based tracking algorithm is in performing the normalized correlations necessary to form a correlation surface. A correlation surface is generated by correlating the pixel array of a feature in a past image with candidate pixel arrays defined by a search window in a current image. The search window is constructed with knowledge of the imaging parameters, sensor geometry, and vehicle state [7]. The peak of the correlation surface indicates the location of a feature at the new time, and can be used by an extended Kalman filter to estimate the spatial coordinates of the ground object which corresponds to the feature [3, 6].

The feature tracking mechanism begins with feature selection by partitioning the master image using a *cell grid*. Each cell is a square pixel area with an odd number of pixels, $2n + 1$, to a side. Each grid cell is scanned to see if an image feature is present. Features can therefore be detected only with the spatial accuracy (within the image plane) of the grid resolution, but they are tracked within the image plane with subpixel resolution. A cell size of 11×11 pixels gives good overall performance, balancing matching accuracy versus spatial resolution [6]. A 512×512 pixel image would therefore be divided into 46×46 cells with 6 pixels remaining along two of the edges. For this implementation, feature selection is based on intensity variance within a grid cell.

2.1 Virtual Processing Regions

Many field-based techniques exhibit regular computational structure at each pixel in the image plane. Such algorithms can achieve near linear speedup on a single-instruction-multiple-data (SIMD) massively-parallel-processor (MPP) [8]. As field-based algorithms become more complex, performing indirect addressing and trigonometric functions at each pixel, MPPs such as the Connection Machine-2 (CM2) have fallen short and indicate the need for more the powerful local processors found in multiple-instruction-multiple-data (MIMD) machines [8].

A problem with our feature-based algorithm, which is shared by more complex field-based techniques, is that the solution of the optical flow (and in our case also the estimation of range) does not exhibit regular computation within the image plane. Computational complexity rises dramatically in areas of imagery densely populated by features. To combat the problem of imagery-dependent complexity, two software abstractions were created and described in [7]. The first is the *autonomous tracking unit* (ATU) and the second is the *virtual processor region* (VPR).

Once a feature is detected within a grid cell, an ATU is spawned to track the feature over time and estimate the range to the Earth-fixed object which gave rise to it. An ATU is a task-parallel feature tracker: it is autonomous in the sense that it tracks it's feature through the image plane, while scene content evolves, regardless of other features. If a feature leaves the image plane or otherwise becomes untrackable then the ATU dies. As motion imagery evolves, ATUs will track the optical flow within the image. Thus an ATU will generally flow outward from its initial location in the image (assuming a forward-looking sensor in forward motion).

Since the ATU is designed to be autonomous, its data requirements, in terms of image plane locality, will vary with motion imagery. Such task autonomy makes it possible for two ATUs, spawned near each other in the image plane, to diverge from each other over time.

The implication of this is that task parallelism alone is not sufficient to efficiently map the algorithm

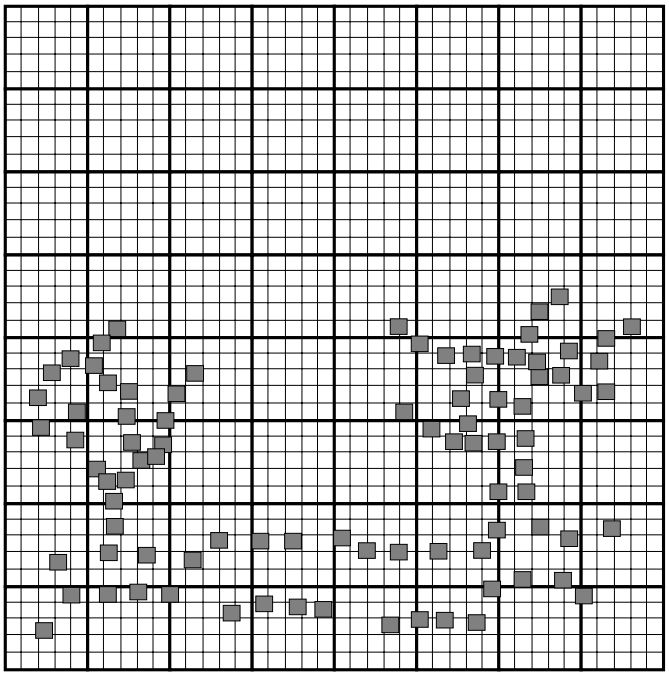

Figure 1: Image plane partitioning.

onto simple parallel hardware. To overcome the data locality requirements, a higher level of abstraction is introduced above the level of the ATU. This abstraction, the virtual processor region (VPR), adds spatial locality restrictions to each ATU within the image space.

Fig. 1 illustrates the idea behind virtual processor regions. The textured squares represent the location of ATUs within a master sensor image plane. The ATUs are arranged to illustrate the tracking of two trees and several ground features. The image is divided into 8×8 VPRs (heavy lines). Each VPR is responsible for maintaining a rectangular arrangement of grid cells. In this example each VPR is allocated 5×5 grid cells (thin lines). The boundaries for the VPRs are the same as for the underlying cell grid. The maximum number of VPRs is equal to the number of grid cells.[1]

The VPRs represent separate regions within the image plane that can be allocated to a processing element (PE). In the example of Fig. 1 there are 64 VPRs which can be distributed among up to 64 processing elements in a task/data parallel fashion. Each PE processes the ATUs (textured squares) and performs feature detection in untracked grid cells (white squares) which are contained within its assigned VPR. Since the VPRs are spatially allocated their image data requirements are fixed. Therefore, as an ATU tracks a feature across a VPR boundary, the VPR passes the ATU to the appropriate neighboring VPR before the next image set is acquired. Currently each VPR is allocated enough image pixels surrounding the VPR proper such that ATUs can be tracked into a neighboring VPR's image space without the need for interprocessor communication on distributed memory machines.[2] This is an interim solution allowing for more flexibility in the algorithm such that a wider range of architectures may be considered. If an architecture has very cheap nearest-neighbor communication then ATUs would be designed to be transferred between VPRs during the tracking phase of the algorithm. If the tracked features have inter-image shifts greater than can be handled by the extra pixels sent along with a VPR, then interprocessor communication during the tracking phase will be necessary.

Fig. 2 depicts the feature tracking method as a flow chart using the definitions of autonomous tracking units and virtual processing regions. The feature tracking algorithm based on virtual processing regions

[1] In the case of 512×512 pixel images with 11×11 pixel cells there can be as few as one VPR or as many as 2116 VPRs.

[2] Each VPR is currently given a ten pixel wide image strip from each of its neighbors. The size of this strip is based on the highest inter-image shift expected during tracking. This is not directly applicable to a shared-memory implementation.

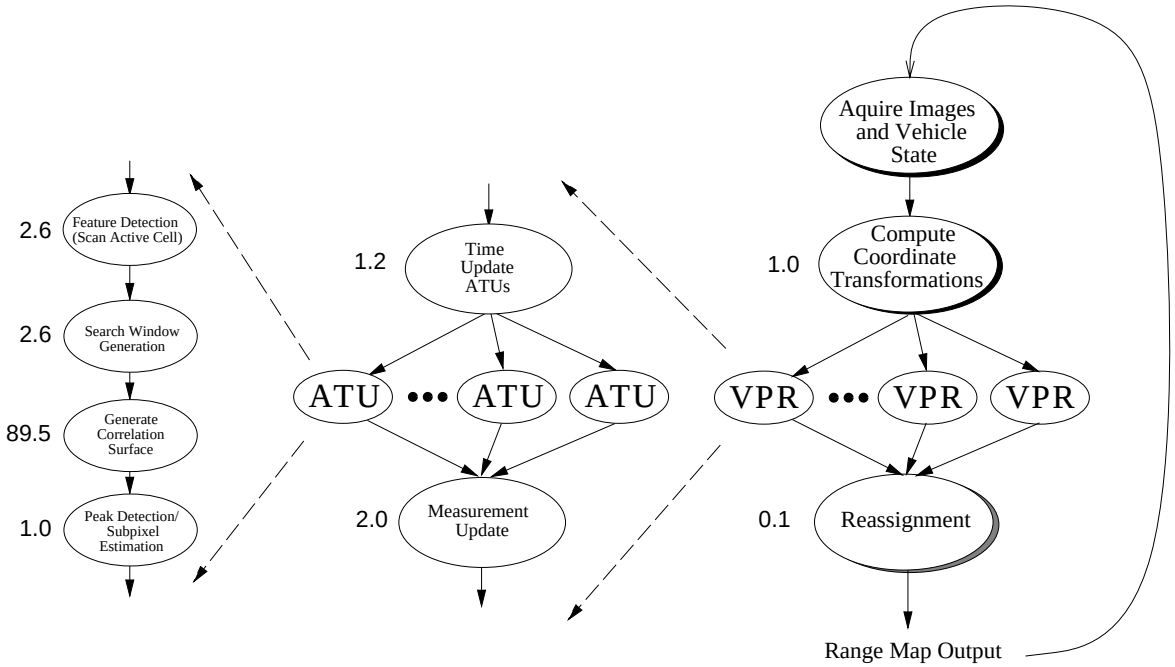

Figure 2: Feature tracking algorithm flowchart.

exhibits Single Program Multiple Data (SPMD) parallelism. Each VPR can be processed in parallel as soon as its input data have been supplied. The VPR however can exploit further parallelism at the ATU level if it contains more than one grid cell.[3] Each ATU/grid cell can be processed in parallel. The data requirements for each ATU are implicitly supplied by the parent VPR. The ATU in turn is composed of a series of serial matrix-like operations which exhibit vector-like parallelism. The number next to each ellipsoid in Fig. 2 indicates the aggregate percentage of total computation needed by each function.

The motivation behind the ATU/VPR construct is flexibility. The feature tracker can be configured to use as few as one or as many as a couple thousand VPRs. Changing the number of VPRs obviously affects the ATU per VPR ratio. On a highly parallel machine (with several thousand processors) each processor would be assigned either an empty grid cell or an active ATU; there would be no need for VPRs. Since the algorithm has been designed to be ported to different architectures, the VPR construct is necessary to reduce the number of task/data parallel units to the optimal number for a given architecture and load balancing scheme.

2.2 Load Balancing

If scene content is such that features are not distributed uniformly (which is most often the case), then a load balancing technique will be needed to make efficient use of every processing element in a system. From previous research we have found that a load balancing technique called static scheduling has performed well for this problem [7]. This scheduling algorithm is presented below.

The major computational load of each VPR is in performing the correlations necessary for feature tracking. If the time to scan a cell for a new feature is t_d, the time to generate a correlation surface is t_c and the time to perform measurement and time update is t_f, then if the ith VPR has A_i untracked cells and B_i ATUs (tracked features), the computation time for the ith VPR, τ_i, can be approximated by

$$\tau_i \approx A_i t_d + B_i(t_c + t_f + t_d) \tag{1}$$

[3] Parallelism below the VPR level has yet to be explicitly exploited.

If the number of features per VPR does not change too rapidly during steady-state feature tracking, then it would be possible to perform static scheduling for the current frame time based on each VPR's estimated computation time τ_i from the previous frame.

Given that the master image is divided into M VPRs, static scheduling attempts to distribute all M VPRs from the current frame onto a set of N processors so as to minimize completion time. It is required that $M > N$ so that the scheduler has the resolution to properly distribute the load. More precisely: Given N processors, a deadline D and an M element set, $\mathcal{X}$, of VPRs each with estimated computation time τ_i, select a disjoint partition of $\mathcal{X} = \mathcal{X}_1 \cup \mathcal{X}_2 \cup \cdots \cup \mathcal{X}_N$ such that

$$\max \left\{ \sum_{i \epsilon \mathcal{X}_j} \tau_i : 1 \leq j \leq N \right\} \leq D \tag{2}$$

The estimated time necessary to process all VPRs with a uniprocessor is

$$T = \sum_{i=1}^{M} \tau_i \tag{3}$$

Thus the best case deadline D possible, given N processors, is T/N.[4] This is known as the Multiprocessor Scheduling Problem [9]. The challenge of static scheduling is to choose the partitions $\mathcal{X}_j$ in a computationally efficient manner such that the maximum processing element computation time approaches T/N.

3 Implementation Results

The results from previous research were inconclusive whether a distributed-memory or shared-memory multicomputer would reach real-time performance sooner. What we could conclude, though, was that a system was needed that could sustain at least from 1.5 to 2.0 Gflops with fast inter-processing-element input/output (I/O). To this end we have ported the algorithm to two architectures which might be able to support this performance level in a system portable enough for installation on a helicopter. The architectures we chose were the iWarp distributed-memory multicomputer and the latest Silicon Graphics Onyx multicomputer. Issues relating to porting *Optflow* to the iWarp (by far the larger of the two tasks) are summarized here. For those readers who wish a detailed explanation see [10].

3.1 Distributed-memory machine

The iWarp is a distributed-memory multicomputer and is the product of a joint development effort between the Intel Corporation and Carnegie Mellon University. The system supports both tightly and loosely coupled parallel processing and was designed to support high-performance signal processing via balanced communication and computation [11]. The iWarp is distinguished from other distributed-memory multicomputers by three significant features: high-bandwidth internode communications, systolic processing, and multiplexed physical connections [12].

Under the iWarp architecture, processing elements, or "nodes", are connected in a 2-D toroidal mesh "array". Each node is a RISC like processor theoretically capable of 20 MIPS and 20 MFLOPS.[5] Each node also has a communication agent able to support four duplex 40 Mbytes/second channels, 320 Mbytes/second aggregate, to other nodes in the array. Input/output for the array is handled by special interface nodes. At present, the only interfaces are the "Sun Interface Board" (SIB), which provides communications and proxy services to a Sun workstation, and the "iWarp Serial Interface" (iSIO). The latter provides 60 MB/sec transfer rates and is intended as a general-purpose interface to high-speed devices such as frame grabbers, video, and disk [13]. A schematic diagram of the iWarp architecture is shown in Fig. 3.

[4] This assumes, of course, that 100% of the code can be parallelized.

[5] There is a hardware bug in the current release of the chip (c-step) which limits the maximum floating point performance to 10 MFLOPS.

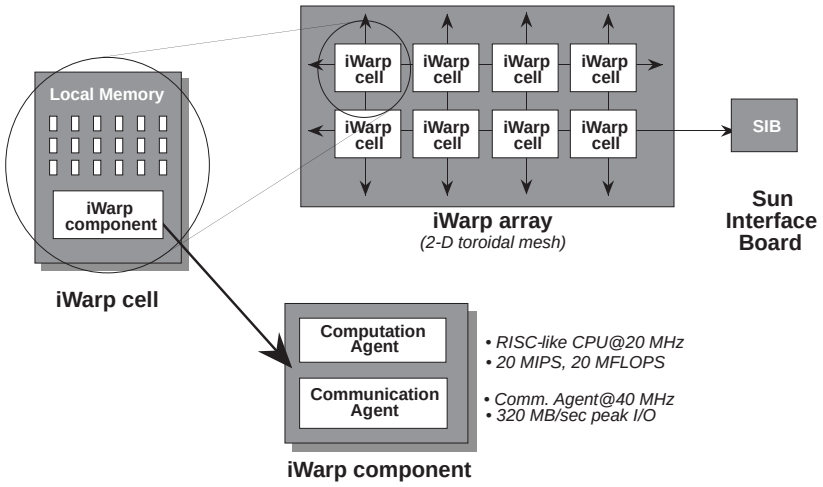

Figure 3: Schematic diagram of the iWarp distributed-memory multicomputer.

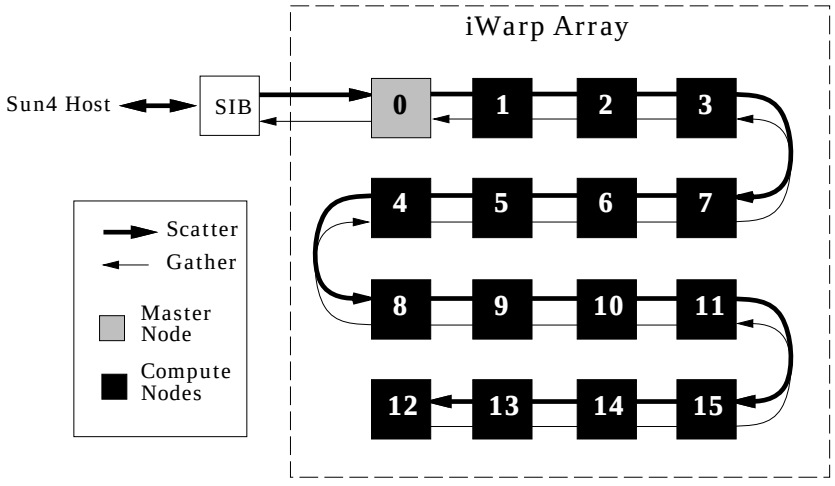

Figure 4: Communication topology for the iWarp implementation of *Optflow*.

Several constraints impacted the implementation of *Optflow* on the iWarp. Foremost were restrictions resulting from current iWarp systems. At present, there are relatively few installations (i.e., 30-40 sites) and most have 8 or fewer nodes. Moreover, almost all machines have limited node memory, typically 512 Kbytes static RAM per node.

A distributed-memory implementation of *Optflow* has been constructed for the Oak Ridge National Laboratory (ORNL) iWarp, the largest system available to us. The system consists of 16 nodes with 2Mbytes of static RAM per node. The SIB supplies the single input/output interface on the system. Although the communication bandwidth provided by the SIB is not sufficient for real-time use, it was considered acceptable for development purposes. A high-performance video interface, which will utilize the iSIO, is currently under development and will meet real-time requirements.

One of the limitations of the iWarp is that the existing iWarp software tools restricted the development process. The standard development environment is limited, consisting of single-node compilers (i.e., C, FORTRAN) and internode communications libraries. A rudimentary facility is also provided for array-to-host communications. This environment is immature relative to other current distributed parallel systems (e.g., iPCS/860, CM-5) and does not provide abstractions beyond the node level.

The iWarp implementation of *Optflow* uses the coarse-grained, message-passing computation model based on VPRs covered in detail in [10]. For this implementation, we chose to construct a managed scatter/gather network using a pair of unidirectional snake networks as shown in Fig. 4. A sending node is located at one end of both networks. The networks traverse the array and make turns at the edges. Receiver nodes are located at each node and have taps into both networks. One network is then used for sending data from the master node to the compute nodes (i.e., scatter) and the other for receiving computed results (i.e., gather).

For each processing loop, the master node obtains, from the host, the image set and vehicle state information to be processed. The master node uses the state information and sensor geometry to create transformation matrices which all the compute nodes will need. The master node then partitions the set of VPRs based on load-balancing criteria, and determines the data package for each compute node (this is the *preprocessing* stage). The data is then distributed to the compute nodes via the scatter network (the *send* stage). Once all the data has been placed onto the scatter network, the master node spins while the compute nodes process the VPRs (this is the *wait* stage). After computation has been performed, the compute nodes return results via the gather network to the master node (the *receive* stage). The master node performs ATU reassignment and cleanup activities (the *postprocessing* stage), and then the loop begins again.

The ORNL iWarp, even though it had 2 Mbytes per node, could not process an entire 512×512 image. In [10], a 512×128 (i.e. a wide image) was used to test the iWarp version of *Optflow*. It was discovered that a bug existed in the image distribution code that executes on the master node. This bug would slow the communications bandwidth between the master node and the compute nodes by an order of magnitude, while distributing image data, when the VPRs were a fraction of the width of the image. Thus, by using a 512×128 image, only 11 VPRs were available to test true speedup. This number of VPRs however was not suitable to test speedup and load balancing on a 16 node system.

In order to test the speedup and load balancing using all 16 nodes, while bypassing the bug, and keeping within the memory limits, a simple modification was made. A 128×512 image was used (i.e. a tall image) instead. This gave the master a maximum of 45 VPRs to distribute to 15 compute nodes. With this configuration, a speedup graph could be generated, bypassing the image distribution bug.

3.1.1 iWarp Performance

Figure 5 is a plot of the speedup for the iWarp version of *Optflow* using the tall images, 128×512, described earlier. A linear speedup, using three to fifteen compute nodes, should give a speedup of five times if there were no nonparallel code, load imbalance, or communication delays.

Table 1 presents a summary of the iWarp performance on parallel and nonparallel sections of the code during the 20th image pair of the test sequence. The test image sequence is described in [7]. The columns labeled Pre&Post (pre and post processing), Send&Recv, and Wait are tasks for which the processing times are measured on the Master Node. The columns labeled Ave (average time to compute all VPRs for all the compute nodes) and Max (maximum node compute time) are computation times measured by the compute

Number of Compute Nodes	Master Node			Compute Node		Percent Balanced	Loop Time
	Pre&Post	Send&Recv	Wait	Ave	Max		
3	0.09912	0.05605	2.50002	2.50424	2.51133	99.7	2.65519
4	0.09381	0.05641	1.90790	1.87812	1.92174	97.7	2.05812
5	0.09588	0.05658	1.50261	1.50251	1.52212	98.7	1.65508
7	0.09131	0.05688	1.29479	1.25228	1.33744	93.6	1.44298
9	0.09933	0.05738	0.98762	0.93918	1.00588	93.3	1.14435
12	0.09791	0.05826	0.78113	0.68304	0.81891	83.4	0.93729
13	0.10506	0.05845	0.66150	0.62658	0.70054	89.4	0.82503
15	0.10273	0.05905	0.63379	0.53704	0.63344	84.7	0.79557

Table 1: Compute times for the 20th frame pair, in seconds, for *Optflow* on the iWarp using 45 VPRs.

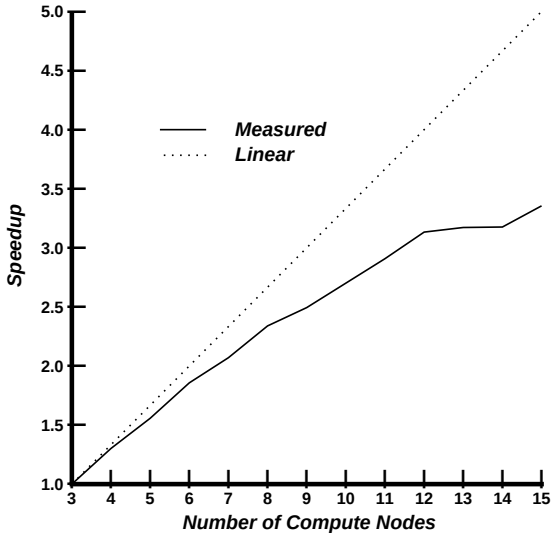

Figure 5: Measured speedup for the iWarp array, excluding host I/O, averaged over 20 image pairs.

nodes. The column labeled Percent Balanced is the ratio of the Ave compute time to the Max compute time. This gives a rough measure of how well the load balancer is functioning. Finally the column labeled Loop Time is the sum of the first three columns and corresponds to the time to process the 20th frame pair, excluding host I/O.

The speedup calculated using numbers from the the Ave column will give $2.50424/0.53704 = 4.66$ which corresponds nearly to the theoretical maximum (i.e. no nonparallel overhead and perfect load balancing). If we take into account load imbalance we can calculate speedup using the Max column, $2.51133/0.63344 = 3.96$. And finally, the true speedup to process the 20th frame pair, including nonparallel code, load imbalance, synchronization delays, and communication/computation overlap we find $2.65519/0.79557 = 3.34$, 67% of theoretical.

The maximum performance for the iWarp can be calculated as the number of features per second computed. The 20th image pair had 417 features, thus $417/0.79557 \approx 524$ features/second. This divided among the number of compute nodes gives almost 35 features/second per node using 15 nodes (52 features/second per node using 3 nodes)[6]

[6] All measurements in features/second within this paper are truncated to an integer value.

Comparison of Onyx and IRIS				
# Model	Nodes	Features/second	(Features/second)/node	Speedup
Onyx	4	763	190	3.71
Onyx	3	538	179	2.77
Onyx	2	385	192	1.95
Onyx	1	196	196	1.00
IRIS	8	553	69	6.88
IRIS	4	290	72	3.77
IRIS	1	74	74	1.00

Table 2: Performance comparison of the four processor Onyx versus a 4D/480.

3.2 Shared-memory machine

The shared-memory machine tested was a four processor Silicon Graphics Onyx. We chose the Onyx due to the promising speedup numbers (6.88 for eight processors) that was achieved by an SGI IRIS 4D/480 in previous research [7]. We were not able to gain access to an eight processor Onyx for direct comparisons to the 4D/480 but the data gleaned from the performance runs on the Onyx using the speedup information from it and the 4D/480 should allow us to make a lower-bound prediction on performance.

The Onyx system tested had four 100 MHZ MIPS R4400 Processors, and an interconnection backplane bandwidth of 1.2 Gbytes/second [14]. The Onyx predecessor, an IRIS 4D/480, has eight 40 MHZ MIPS R3000 Processors and a interconnection backplane bandwidth of 64 Mbytes/second.

The implementation of *Optflow* on the Onyx was of minimal effort compared to the that needed for the iWarp version. The Onyx has a multithreaded operating system (OS) which lends itself to easy coding of data-parallel tasks. A coarse-grained thread approach was chosen to parallelize *Optflow* on this architecture.[7] In this approach N threads are generated for an N-processor machine. The number of VPRs should exceed the number of processors such that the load balancer will have enough resolution to approach the T/N lower bound. For this implementation we chose the number of VPRs to be 64 (same as the previous version for the 4D/480). The static load balancer generates an index map whereby each thread can address the appropriate set of VPRs from shared memory. The assignment of thread to processor is done automatically by the OS. The OS distributes one thread per CPU since each thread will be heavily loaded. Since we did not have direct access to the Onyx, the development time necessary to carry out per-thread time measurements, equal in modular resolution to the iWarp, was not available.

3.2.1 Onyx performance

Figure 6 is a plot the speedup curve from one to four threads. The coarse-grained thread approach achieves good speedup on the Onyx. With so few processors, though, all we can say for certain is that the Onyx speeds up as well as its predecessor.

Table 2 presents a summary of the performance increase the Onyx has over its predecessor. The Onyx achieves a 2.6 times increase in speed over the 4D/480 using the same number of processors. It should be noted here that since the Onyx tested had only four processors, the OS could not "get out of the way" by allocating four processors totally to *Optflow*. The OS on the eight processor 4D/480, however, was able to do this while benchmarking *Optflow*'s performance on four of them. In short, when the number of heavily loaded threads is equal to the number of processors, a degradation appears in the high end of the speedup curve due to the presents of the executing IRIX kernel. This assumption may be the cause for the 3.77 to 3.71 decrease in speedup from the eight-processor 4D/480 to the four-processor Onyx.

The Onyx is able to process approximately 190 features/second per processor with four processors (196 features/second per processor with one processor). An estimate may be made of the performance of an

[7] The original design of the multithreaded version of *Optflow* could only use eight threads due to limitations in the OS for the IRIS 4D/480.

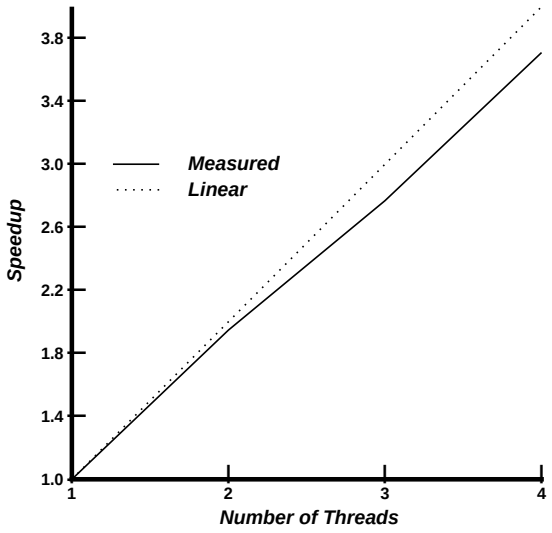

Figure 6: Measured speedup for the four processor SGI Onyx, excluding disk I/O, averaged over 20 image pairs.

eight-processor Onyx using the above data. If we assume it speeds up at least as well as a 4D/480, and that it will get a performance boost of at least 30% by going to a 150 MHZ processor,[8] then an eight-processor 150 MHZ Onyx system should be able to estimate range to $(6.88/3.77) \cdot (190/2) \cdot 1.3 \approx 225$ features/second per processor, or 1800 features/second. An unrestricted operational system[9], one in which the algorithm is limited to at least 1000 active ATUs per frame, with a minimal frame rate of 10 frames/second, would need a compute system capable of processing $10,000$ features/second (assuming pipelined I/O). Therefore an eight-processor Onyx may only be 5.5 times too slow to meet requirements for real-time processing.

Onyx systems with more processors (than we where able to test) are promised. Future systems will have available a new superscalar processor, the TFP, optimized for floating point operations [14]. Performance speculations for Onyx systems with more than eight processors or those with the new TFP processor cannot be justified until more advanced systems are available to us.

4 Conclusions

In our earlier research we could not conclude which architectural design, shared or distributed memory, would offer the most performance for *Optflow*. All we could give was a rough idea, in MFLOPS and Mbytes/second, of what level of performance a parallel processor needed. The results of this paper highlight the impact, on *Optflow*'s performance, of the different hardware designs tested.

The performance of *Optflow* on the iWarp suffered in three different areas. First, 45 VPRs is not adequate to distribute to 15 nodes. The load balancer does not have the resolution to smoothly distribute to load generated by the test image sequence. Therefore the high end of the iWarp speedup graph becomes very choppy. Second, is the compiler and debugging environment are rudimentary. Hours were spent unrolling loops by hand and swapping integer operations for floating point and vise versa so that the iWarp's LIW (long instruction word) could be utilized by the compiler to keep the integer and floating point units busy. Finally the iWarp cell itself does not have the installed base to mandate the resources at Intel to keep its

[8] 150 MHZ will be the default for these machines.

[9] We have found that in real helicopter imagery often only 400 to 600 features will be present. Laboratory imagery (used to benchmark systems) is atypical, with as many as 1500 features, due to the absence of "sky" and other featureless objects. The justification for needing at least 10 frames/second can be found in [15, 16].

raw computational performance competitive with popular architectures such as the MIPS R4400.

A criticism can be made that *Optflow*, in its managed scatter/gather implementation, was not optimal for a toroidal systolic mesh. This is almost certainly true, but for any parallel architecture to perform well in a wide variety of codes, it must not be too communication-model-specific. A large enough iWarp array may be able to achieve real-time performance for *Optflow*, though it will take a great deal of effort reorganizing the code to reach such a goal.

The Onyx was able to outperform a 16 processor iWarp with only three 100 MHZ R4400s. If such large disparities between raw processor performance are to continue, then commercial compute servers will reach our goal of real-time performance the soonest.

Thanks

The authors would like to express sincere appreciation to Judson Jones and the Oak Ridge National Laboratory for use of the ORNL iWarp machine and to Ken Harris and Paul Delzio of Silicon Graphics for their timely response to our request to benchmark a new Onyx machine.

References

[1] B. Sridhar, V.H.L. Cheng, and H. Swenson. Automatic guidance of rotorcraft in low altitude flight. In *AGARD 53rd Symposium of the Guidance and Control Panel on Air Vehicle Mission Control and Management*, Amsterdam, Netherlands, October 1991.

[2] V.H.L. Cheng and B. Sridhar. Technologies for automating rotorcraft nap-of-the-earth flight. In *Proceedings of the AHS 48*th *Annual Forum*, pages 1539–1554, Washington, D.C., June 1992.

[3] B. Sridhar and A.V. Phatak. Simulation and analysis of image-based navigation system for rotorcraft low-altitude flight. In *Proceedings of the AHS Meeting on Automation Application for Rotorcraft*, Atlanta, GA, April 1988.

[4] P. K. Menon, G. B. Chatterji, and B. Sridhar. Vision-based optimal obstacle avoidance guidance for rotorcraft. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference*, New Orleans, LA, August 1991.

[5] Y. Barniv. Velocity filtering applied to optical flow calculations. Technical Memorandum 102802, NASA, Ames Research Center, Moffett Field, CA, August 1990.

[6] B. Sridhar, R.E. Suorsa, and B. Hussien. Passive range estimation for rotocraft low altitude flight. *Journal of Machine Vision and Applications*, 1993. Technical Memorandum 103899, NASA Ames Research Center. October 1991.

[7] R.E. Suorsa and B. Sridhar. A parallel implementation of a multisensor feature-based range-estimation method. Technical Memorandum 103999, NASA Ames Research Center, Moffett Field, CA 94035, February 1993.

[8] P. Nesi, A. Del Bimbo, and D. Ben-Tzvi. Algorithms for optical flow estimation in real-time on Connection Machine-2. Technical Memorandum DIS-RT 24/92, University of Florence, Via S. Marta 3, 50139 Firenze, Italy, August 1992.

[9] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completness*, chapter 3, pages 63–65. W.H. Freeman and Company, San Francisco, CA, 1979.

[10] T.W. Fong and R.E. Suorsa. Real-time optical flow range estimation on the iWarp. In *Proceedings of the SPIE International Symposium on Intelligent Information Systems*, Orlando, Fl, 1993.

[11] S. Borkar, R. Cohn, G. Cox, T. Gross, H.T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Proceedings of Supercomputing 88*, pages 330–339, November 1988.

[12] T.W. Fong. The iWarp guide: An introduction to the iWarp at NASA Ames. Technical memorandum, NASA Ames Research Center, Moffett Field, CA 94035, 1992.

[13] B. P. Hine III and T. W. Fong. Evaluation of the Intel iWarp parallel process for space flight applications. In *Proceedings of the 1993 AIAA Aerospace Design Conference*, 1993.

[14] Silicon Graphics, Inc. Symmetric Multiprocessing Systems. Technical Report, Silicon Graphics, 2011 N. Shoreline Boulevard, Mountain View, CA 94043, 1993.

[15] P.N. Smith, B. Sridhar, and B. Hussien. Vision-based range estimation using helicopter flight data. In *Proceedings of the 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 202–208, Champaign, Illinois, June 1992.

[16] B. Sridhar, P.N. Smith, R.E. Suorsa, and B. Hussien. Multirate and event driven Kalman filters for helicopter passive ranging. In *Proceedings of the 1st IEEE Conference on Control Applications*, Dayton, Ohio, September 1992.