JOURNAL OF GUIDANCE, CONTROL, AND DYNAMICS
Vol. 25, No. 4, July–August 2002

Uncorrelated Observations Processing at Naval Space Command

Shannon L. Coffey,* Harold L. Neal,† and Mathew M. Berry‡
U.S. Naval Research Laboratory, Washington, D.C. 20375-5355

The Naval Space Command (NSC) uses a program called SAD (Search and Determine) to construct satellite orbits from uncorrelated observations. The process involves constructing orbits from pairs of observations that are not associated with a known object. A lot of computer time is required to explore all of the candidate orbits, which can be constructed from the several thousand uncorrelated observations that are maintained. To shorten the run time, a parallel-processing approach was taken. This distributes the processing across multiple computers, which allows significantly more data to be processed than with the previous serial program. This has resulted in a greater number of orbits being discovered than was previously possible, and the run times have been significantly reduced. This revised program is now run in an off-line mode at NSC. It represents the first known parallel-processing application to be used at a Navy Operational Command. The next iteration of the software will come with appropriate database and user-interface modifications to allow its full-scale integration into the operational environment at NSC.

Introduction

THE Naval Space Command (NSC) and the Air Force Space Command are the two U.S. organizations that maintain catalogs of space objects. These catalogs, which are updated daily, currently contain orbital elements for over 9000 objects. The catalogs are updated by differentially correcting the initial conditions for the objects by processing observations from ground-based sensors. The sensors frequently provide observations that cannot be correlated to any object in the catalog. These uncorrelated observations can arise from several sources. Some of the observations are of new objects formed from the fragmentation of payloads or rocket bodies. These particles can sometimes be traced to a parent body. However, frequently they are detected long after the fragmentation event, in which case they show up on the radar as an unknown or uncorrelated observation or otherwise known as Uncorrelated Target (UCT). Uncorrelated observations also come from objects that were once maintained in the catalog, but were later lost. This situation often arises for objects with highly eccentric orbits that spend a large amount of time far away from the Earth where they cannot be detected by sensors.

In the late 1960s NSC developed a program called SAD (Search and Determine) for constructing element sets from observations associated with fragmentation events. For this application knowing the blast point aids considerably in constructing element sets for the fragments. SAD has also been used to process large volumes of uncorrelated targets into element sets. The program is capable of processing observations from any sensor in the Space Surveillance Network, but in practice it is usually applied to data from NSC's network of radar observing stations (known as the Fence). In this paper we focus our discussion on the application of SAD to UCT processing only.

To meet the needs for faster processing of UCTs and for processing larger data sets, we undertook the task of rewriting SAD for parallel execution. To parallelize the algorithm, we implemented a master-slave design using parallel virtual machine (PVM) message passing. Our goal was to reuse as much of the original code as possible to speed development of the software and to preserve the functionality of the program.

In this paper we present the original SAD algorithm and discuss the problems we faced and solutions we chose when we modified it to run in a parallel-processing environment. We present results of testing the program with different configurations of a massively parallel processor, the IBM SP2 at the Air Force's Maui High Performance Computing Center. We also present results from real-world use of the parallel program by NSC.

It is frequently the case that the effort involved in developing an initial version of a program for first use is substantially less than the effort required to finalize the program for an operational environment such as that at NSC. We will discuss some aspects of the final program that is being installed at NSC in the "Implementation at NSC" section.

Observations of Space Objects

The Space Surveillance Network (SSN) of Radars consists of a wide variety of sensors, dish antennas, phased array radars, and the NSC Fence. The NSC Fence is a unique radar system consisting of three transmitters and six receivers. The transmitters conduct a wide beam of radar energy into space. When a satellite passes through this radar beam, the reflected radar signal is bounced down to one or more receiver sites. Radar observations detected at one receiver site are called single observations or "singles." These data are reduced to the form of direction cosines. Sometimes more than one receiver site will get the returned energy in which case a triangulated position of the object in space is produced. This triangulated position, frequently referred to as a TRI, is what is processed by the SAD program. The TRIs are also the data product sent to Cheyenne Mountain to be folded in with the other SSN observations.

The phased array and dish antennas produce a much different data product. They frequently lock on to a target and will track it for several seconds through a pass. When several observations, referred to here as "points," of the same object are produced during one pass, that set of points is denoted as a "track." Generally, the sensor sites will be tracking a known object and will label or tag the observations accordingly. In addition, UCTs are also produced and sent to Cheyenne Mountain and to NSC each day.

For convenience, in this paper we will denote the SSN as those radars from the Space Surveillance Network, excluding the NSC Fence.

The SAD program was designed to operate on Fence triangulated observations. It can incorporate SSN data as well. However to use these data the tracks must be reduced to a single observation.

NSC receives approximately 100,000 individual points from the SSN each day. On average, there are about five points per track. In addition, they receive about 3700 UCT points from the SSN each day. About 70% of these observations are automatically correlated

Received 5 August 1999; revision received 31 August 2000; accepted for publication 31 August 2000.

*Head, Mathematics and Orbit Dynamics Section. Associate Fellow AIAA.

†Aerospace Engineer.

‡Aerospace Engineer, Mathematics and Orbit Dynamics Section. Student Member AIAA.

to known objects; another 20% are manually correlated. This leaves about 10% or about 370 points per day that go into a holding file of UCTs. NSC maintains about 60 days of these unresolved UCTs. At the time of this paper, there were 24,587 SSN UCT points on file at NSC.

NSC receives approximately 150,000 individual direction cosines each day from the Fence. From these observations they receive about 155 UCT triangulated points each day. At the time of this paper, they had 13,191 UCT TRIs on file.

Therefore, NSC normally has about 37,000 UCTs from the Fence and the SSN on file at any one time. The file of UCTs covers a time span of 60 days. This constitutes the set of unknowns that must be processed.

As large as the UCT file is and the enormous processing time it takes to sift through it for new objects, the computer resources required are small in comparison to what may be required in the future. Currently there is a proposal to upgrade the Fence to a higher frequency that would allow it to detect objects as small as 5 cm. If this happens, the Fence may be producing as many as 2.5 million observations per day on 100,000 objects. At first almost all of these new observations will be UCTs. This will present an enormous processing problem, which must be addressed if the full benefit from the upgraded Fence is to be realized.

SAD Algorithm

The essence of the algorithm for SAD, when it is used for UCT processing, is to form every possible pair of observations from the uncorrelated observation set and then determine what orbits could contain these two observations. When SAD is used for processing fragmentation events, the blast point of the fragmentation is used as the first observation in every pair. In such a case, SAD does not consider any pair of observations that does not include the blast point, and so the number of pairs to process decreases tremendously and the computer resources needed are greatly reduced. In this paper we focus on the more demanding problem of uncorrelated observation processing. For uncorrelated observation processing the observations' set is usually made up of Earth-centered X-Y-Z observations. These observations are created by triangulating direction cosine data from the NSC Fence to create one X-Y-Z observation per orbital pass.

Creating Candidate Orbits

The SAD algorithm begins by sorting the observation set according to time. Each observation is checked to make sure it falls within an analyst-supplied time range. The analyst can also specify that duplicate and near-duplicate observations be deleted. The program then forms the Cartesian product of the observation set with itself. The resulting set of pairs of observations are called starter pairs. We refer to the earlier observation, with respect to time, in each starter pair as the starter point. The time between the two observations in the starter pair $\langle \Delta t \rangle$ must fall inside an analyst-specified range.

The orbital elements corresponding to each possible orbit through the starter pair are obtained through a modified solution to Lambert's problem.[1] The solution has been modified so that the orbital elements obtained are consistent with the analytic propagator PPT3 (Ref. 2) that is used to propagate the elements. PPT3 is based on the Brouwer theory and takes into account zonal gravity harmonics through fifth degree. The modifications to the solution involve correcting for secular drifts in the argument of perigee and the right ascension of the node between the times of the two observations in the starter pair. The mean motion calculated in the solution is the Brouwer mean motion, not the Keplerian mean motion.

For each starter pair there are several possible solutions to Lambert's problem, which contain the two observations. For example, the number of revolutions N that the satellite made about the Earth between the two observations is unknown; for each value of N the solution is different. In SAD the analyst places limits on the range of allowable values for N by specifying the minimum and maximum orbital periods $T_{\min}$ and $T_{\max}$ for the orbits that SAD will attempt to construct. Given the time $\langle \Delta t \rangle$ between the two observations, SAD calculates the limits on N such that the period of the orbit is in the specified range. The limits on the number of revolutions such that the orbital period falls in the interval $[T_{\min}, T_{\max}]$ are extended by two on both sides to guarantee that all orbits near the analyst-specified limits will be considered. Thus, the limits on the number of revolutions are

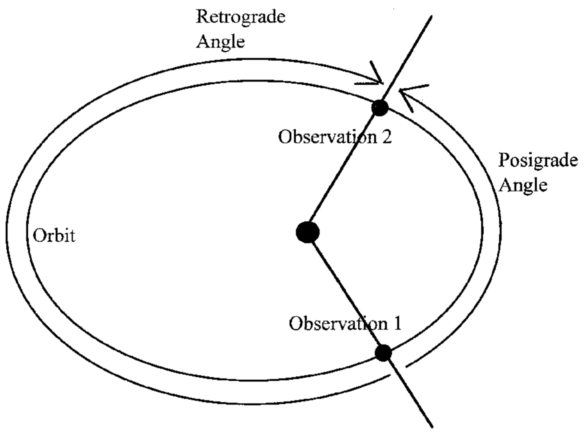

Fig. 1 Residual angle after $N - 1$ rotations.

$$N_{\min} = \Delta t / T_{\min} - 2, \qquad N_{\max} = \Delta t / T_{\min} + 2$$

where both limits are rounded down to the nearest whole number. $N_{\min}$ has a minimum value of one. For a given N there are two possible directions of motion for an orbit, either posigrade or retrograde; after $(N - 1)$ complete revolutions the last partial revolution is through the angle from the first observation to the second. In the case of a posigrade orbit, the angle is measured positively from the first observation to the second. In the case of a retrograde orbit, the angle is measured in a negative sense from the first observation to the second (Fig. 1). Finally, for a given N and a given direction of motion, there are two solutions to Lambert's problem, based on the two roots of the equation for the semilatus rectum involved in the solution. Therefore, considering all possible N and directions of motion, there are

$$4(N_{\max} - N_{\min} + 1)$$

solutions for a given starter pair. The SAD algorithm loops through all of these possible solutions, generating a candidate element set for each one. The algorithm consists essentially of three do-loops. The innermost loop alternates the two possible solutions for the semilatus rectum for a given N and direction of motion; the next loop alternates between the two directions of motion; and the outermost loop varies through N, the possible number of revolutions between the observations.

Many of the candidate element sets are rejected immediately. For some choices of N, direction of motion, and semilatus rectum root, no element set can be determined because no numerically valid solution can be computed. Some of the element sets will be physically impossible; for example, some orbits will have a negative altitude at perigee. Other element sets can be rejected because they do not fall within the orbital period interval specified. If a candidate element set is not rejected, it is processed next by the orbit refinement loop.

Orbit Refinement Loop

The orbit refinement loop begins with a simple least-squares differential correction of the candidate element set using the PPT3 analytic propagator. Only the two observations in the starter pair are used in this initial differential correction. This step is necessary so that the element set reflects the contributions of long- and short-period terms in the analytic propagator. The differential corrector iterates up to 11 times or until the element corrections are sufficiently small. Using the corrected element set, SAD then attempts to identify additional observations that correspond to that element set. This is done by having SAD predict when a satellite with this orbit would cross the NSC Fence, in other words, when this satellite should create an observation. SAD checks all of the observations in the observation set after the starter point (and those within some time span before the starter point) against two criteria to find observations that may correspond to the predicted Fence crossings. The criteria are the following: 1) the time of observation must be near the time of one of the predicted Fence crossings, and 2) the observation must lie near the orbital plane defined by the corrected element set.

If there are no observations besides the starter pair that meet these two criteria, the element set cannot be further refined. However, if additional observations do meet the criteria, the differential correction is repeated. For this differential correction all of the observations that meet the two criteria are used, but the two observations in the starter pair are weighted much more heavily than the other observations. This time, the differential corrector will iterate until the corrections are smaller than a predefined tolerance up to a maximum of 10 iterations. If at least four observations are used in the differential correction, a drag decay parameter is solved for and included in the analytic propagation. The individual X, Y, and Z components of the observations must meet a tolerance condition before they are used in the differential correction. For the first two iterations of the differential corrector, the tolerance is set large enough to accept all observation components. After that, the tolerance T_{DC} is given by

$$T_{\mathrm{DC}} = \mathrm{Max}\left\{2(\text{average rms error of previous iteration}),\ 2P_g a\sqrt{F_t}\Big/R_e\right\} \tag{1}$$

where a is the semimajor axis of the orbit, R_e is the Earth's radius, F_t is an analyst-specified tolerance parameter between one and nine, and P_g is given by

$$P_g = \mathrm{Max}\{1,\ 100[1.075 - (a/R_e)(1 - e)]\}$$

where e is the eccentricity and P_g is used to loosen the tolerance for orbits that have a very low perigee altitude. When Eq. (1) is used for T_{DC}, the tolerance is actually a measure of the average rms error of the differentially-corrected fit.

After this differential correction is completed, SAD uses the resulting element set to repeat the orbit refinement loop: the Fence crossings are predicted, the observation set is searched for observations corresponding to those crossings, and differential correction is performed with the resulting observations. The search for observations corresponding to Fence crossings is done from scratch with the corrected element set; with the exception of the observations in the starter pair, the observations used in the preceding differential correction do not automatically carry over to the next one.

This orbit refinement loop is repeated up to 15 times provided the following conditions are met:

1) After the initial differential correction at least one observation besides the starter pair is found that corresponds to a predicted Fence crossing.

2) The number of observations corresponding to Fence crossings increases from the earlier orbit refinement, or the tolerance T_{DC} used on the last iteration of the differential correction decreases from the tolerance used on the last iteration of the differential correction of the earlier orbit refinement.

3) After the third orbit refinement there are at least three or four observations (depending on analyst-specified settings) corresponding to predicted Fence crossings.

4) After the third orbit refinement the tolerance T_{DC} used in the differential correction is less than

$$5P_g a\sqrt{F_t}\Big/R_e$$

If all of these conditions are met, the element set is saved, although it may be overwritten on the next pass through the orbit refinement loop. After the program exits from the orbit refinement loop, we will have the best representation of the candidate orbit. Often however, these conditions are not met for a particular candidate orbit, in which case nothing is saved.

Considering all of the combinations of number of revolutions, orbit direction, and so on, there could be up to $4(N_{\max} - N_{\min} + 1)$ candidate orbits for each starter pair. Typically, however, far fewer than half of these will be processed in the orbit refinement loop because the others fail physical or analyst-specified limits. After each valid candidate orbit has been refined, the resulting element set is compared against the best element set produced thus far for that starter pair. Only the better of these two element sets is saved. In this case the better orbit is the one with the greater number of observations corresponding to predicted Fence crossings. If both have the same number of observations, the better orbit is the one with the lower (more stringent) tolerance during the last iteration of the differential correction.

After all of the candidate orbits for a starter pair have been processed, the best one is printed, along with the observations that were used during the differential correction. However, it is often the case that there is no candidate orbit that met the four conditions specified earlier. In that case there is no valid orbit for this starter pair, so that nothing is output. Processing then continues with the next starter pair.

Managing the Observation Set

In the basic SAD algorithm just presented, the observation set would remain unchanged throughout the run of the program. This would allow some of the observations to be used to create or refine several different element sets. However, in reality, each observation can belong to only one element set. If we have a high degree of certainty that an element set found by SAD truly accounts for the observations used to create and refine it, it makes no sense to allow SAD to form additional element sets with those observations. Therefore, SAD removes the observations used to create and refine such element sets from the observation set. We give the name superior orbit to an orbit corresponding to such an element set. In SAD we define a superior orbit as one meeting these conditions: 1) at least five observations (or six depending on analyst specifications), including the starter pair, are used during the orbit refinement; and 2) the rms error between the final differential correction fit and the observations is less than 1 km.

The observations belonging to a superior orbit are removed from the observation set at the time that the element set is printed by means of a flag vector, which indicates which observations are to be used and which are to be ignored in subsequent processing. When a superior orbit is found, the processing of all succeeding starter pairs will be affected. Because the removed observations will not be used in any succeeding starter pairs, nor can they be used as observations corresponding to Fence crossings during the orbit refinement loop, the number of orbits found may be significantly reduced.

Because the observation set changes during the course of the program execution, the order in which the starter pairs are processed is important. We would, therefore, prefer to process high-quality starter pairs first. The quality of a starter pair depends on the quality of the two observations in the starter pair. For triangulated X-Y-Z observations we define the quality of an observation as the number of stations that were used in the triangulation which created the observation. The quality values for X-Y-Z observations can be between two and six; a higher value means a higher quality observation. By adding up the quality values for the two observations in the starter pair, we obtain the quality sum for the starter pair.

The starter pairs are processed in order from the highest quality sum to the lowest. (Note, however, that quality values do not have any effect on which observations are selected when the program searches for observations corresponding to Fence crossings during the orbit refinement loop.) In the program this is implemented by looping through the observations set one time for each possible quality sum. On each pass through the loop, the only observations that will be used as starter points are those that have quality values such that potential starter pairs made with this starter point might have the desired quality sum, given the range (two through six) of possible quality values for the second observation in the starter pair. For example, for the pass through the loop processing quality sums of 10, only observations with quality values of four, five, or six will be chosen as starter points. Once a starter point has been chosen, it is simple to form starter pairs with only those observations which are greater than or equal to the desired quality sum.

In Fig. 2 we present a pseudocode outline of the SAD algorithm, including the handling of quality sums and superior orbits.

```
Initialize constants
Read and sort observation set
Loop on quality sum: from highest to lowest
  Loop on starter point: from beginning to end of observation set--only those which
  can yield this quality sum
    Loop on second observation in starter pair: from the observation after the
    starter point to end of observation set--only those that yield quality sum
      Loop on number of revolutions between obs in starter pair: from $N_{min}$ to $N_{max}$
        Loop on direction of orbit: posigrade and retrograde
          Loop on two choices of semi-latus rectum used to solve Lambert's problem
            Obtain initial element set from solution to Lambert's problem
            Perform a differential correction using only starter pair
            Orbit refinement loop: perform up to 15 times
              Predict fence crossings
              Search for corresponding observations
              Perform a differential correction with those obs
              Save element set if it meets conditions and is the best for this
              solution to Lambert's problem
            End orbit refinement loop
            Save best element set out of all possible orbits for this starter pair
          End semi-latus rectum loop
        End orbit direction loop
      End number of revolutions loop
      Output best solution for this starter pair if it meets conditions
      Remove observations associated with superior orbits
    End starter pair loop
  End starter point loop
End quality sum loop
```

Fig. 2 Pseudocode outline of the serial version of SAD.

Modifications to the observation set and to the order in which starter pairs are processed create serious problems in parallelizing the code because we wish to process several starter points concurrently. It would appear that this would not be possible if the processing of the *n*th starter point might be affected by what happens during the processing of the $(n - 1)$th starter point. We discuss our approach to this problem in the next section.

Parallelizing the Algorithm

A number of programs originally implemented on serial computers can benefit from parallel processing. In some cases execution improvements can be realized only by extensive redesign of the underlying algorithms. This is especially true of programs that require extensive communication of low-level data between processors. SIMD (single instruction, multiple data) computers, such as the Connection Machine CM-5, were originally designed for these types of parallel applications. In other cases a problem may require applying the same process to multiple objects, with no exchange of information between the individual processors. Often in such applications existing software can be reused without extensive revision. Generally, the biggest effort is in designing a method to coordinate the program execution on multiple processors. Although SIMD computers have been used for such problems, the execution time on these computers can be less than optimal because all processors on the machine must be synchronized at the instruction level. For problems where some objects require far less processing time than others, strict synchronization results in some processors being underutilized, and so a more loosely coupled processing environment is more efficient. For our project we combined elements of both approaches. Each processor can handle a different starter point; however, the processors must communicate when a superior orbit is found that changes the observations set that they all use.

To implement a parallel version of SAD, we must have a way to control the many processors that will be executing our algorithm and a way to communicate between the processors. These processors could be a network of heterogeneous workstations or the nodes of a massively parallel processor such as the IBM SP2. For this purpose we chose to use PVM, a public-domain set of message-passing libraries and utility programs.[3] PVM is available for numerous platforms so that our program can be easily ported to different computer systems. One advantage to PVM is that it allows us to design some level of fault tolerance into our program, which we discuss next.

Master-Slave Design

To facilitate a parallel version of SAD, we chose a master-slave parallel design paradigm. The master program, which is what the analyst starts when he wants to run SAD, coordinates the execution of a number of slave programs. The master reads in the analyst-specified parameters and the observations, which it sorts according to time. This code was unchanged from the serial version of SAD. The master starts up the slave programs, one for each node in our virtual machine. The virtual machine is simply the collection of processors on which we will be executing our program. The master can autodetect the number of nodes available in our virtual machine so that the analyst does not need to specify how many nodes to use. The master program broadcasts the observation set to all of the slaves. Then, the master sends to each slave the value of the quality sum being processed and a unique starter point for each slave program to process. As in the serial version, the order in which the starter points are sent out is dependent on what quality sum is being processed. The master program then waits for messages from the slaves. If it receives a message containing an element set, along with the observations used to create and refine it, the master prints them. Once it receives a message that the slave has finished processing its starter point, the master sends the slave a new starter point to process.

The slave programs are duplicates of each other. The relation between the master, slaves, and the Mission System at NSC is illustrated in Fig. 3. All communication with the NSC mission system is through a common source of information, an Oracle database.

Each slave program begins forming all of the starter pairs that contain its unique starter point and which yield the desired quality sum. From this point in the algorithm, processing continues just as it did in the serial version. When a slave program finds the best orbit for a starter pair that meets the conditions to be printed out, the orbit is passed back to the master program, along with its supporting observations. The slave program continues processing its starter point, while the master program prints out the orbit. Once a slave has finished processing its starter point, it notifies the master program, which then sends the slave a new starter point to work with. By letting the slaves request more work when they finish, we achieve an effective load balancing, or division of labor, among all of the slaves in the virtual machine. Although it requires a little more communication between the master and the slaves, this method is more efficient than partitioning all of the starter points among the nodes at the beginning of processing. In that scheme any node that has finished its set of starter points would be left idle while the other nodes still have work to do. This allows the slaves to work at their own pace in accordance with their local workload. None of the computers in the virtual parallel processing system are dedicated to SAD; they all perform other tasks, which is the normal case for general purpose computers.

Changes from the Serial Version

The greatest dilemma in designing the parallel version of SAD was how to handle superior orbits. In the serial version of SAD, the discovery of a superior orbit affects the processing of all succeeding starter points because the observation set is adjusted by removing the observations linked to the superior orbit. In the parallel program several starter points are processed concurrently, so that when a superior orbit corresponding to a particular starter point is found in one slave program some of the observations supporting that superior orbit may have been already processed as starter points by other processors. To mimic the serial version of SAD, all starter points after the one for which a superior orbit was found would have to be reprocessed with the adjusted observations set. This would require greater complexity in the code to take care of the necessary bookkeeping, and the reprocessing could severely diminish the efficiency of the parallel code, especially when a large number of nodes are present in the virtual machine. Therefore, we sought another way to deal with this problem.

Through testing of the SAD program, we discovered that superior orbits were relatively rare. Out of several hundred orbits produced from a typical observation set, there are generally less than a dozen superior orbits. It would appear then that removing the observations associated with superior orbits is not of significant importance in the algorithm. This observation led us to reexamine the purpose of superior orbit handling in the original algorithm.

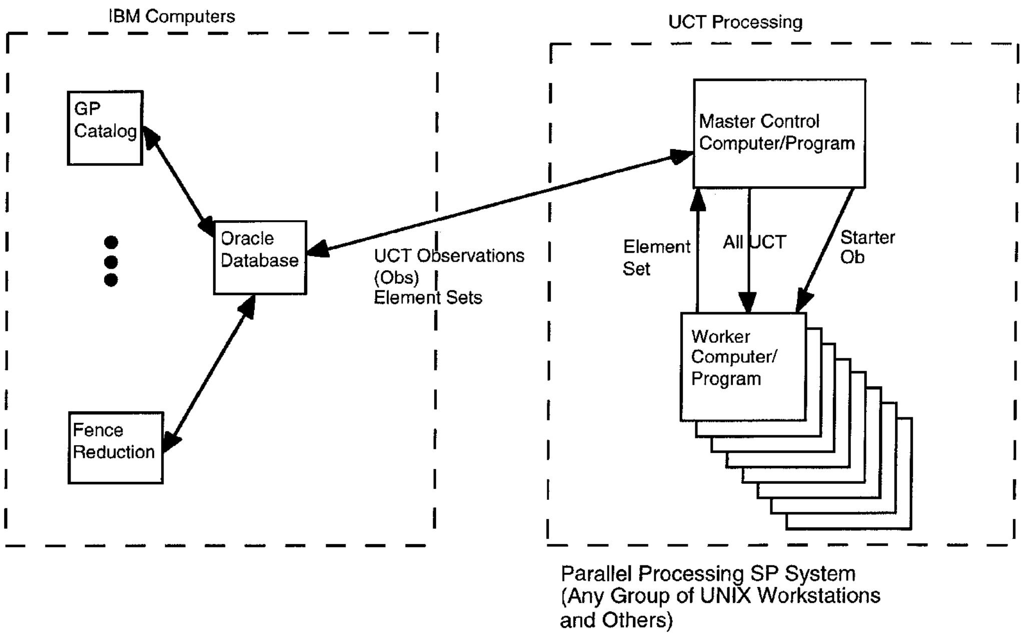
IBM Computers
GP Catalog
Oracle Database
Fence Reduction
UCT Observations (Obs) Element Sets
UCT Processing
Master Control Computer/Program
Element Set
All UCT
Starter Ob
Worker Computer/ Program
Parallel Processing SP System (Any Group of UNIX Workstations and Others)

Fig. 3 Parallel-processing master slave organization.

SAD produces far more orbits than the analyst will add to the catalog. The analyst takes the output from SAD and selects the best orbits, using other software tools and his or her experience, and adds those to the catalog. If an orbit should truly be added to the catalog, it should show up in the next SAD run anyway if it is not added to the catalog this time. So, the analyst is more interested in getting a group of orbits from which to select the best, rather than the exact same output that the serial version of the algorithm would produce.

Therefore, we have some leeway in handling superior orbits. However, we still wish to remove the observations associated with superior orbits from the observation set. Leaving them in will result in extra orbits in the output: each time one of the observations associated with a superior orbit is used as a starter point, another version of the orbit may be produced in the output, although the orbital elements and the associated observations will vary from one incarnation to the next. (It is also possible to get multiple versions of the same orbit for nonsuperior orbits.)

Our solution is to update the observation set on the master and all of the slaves as soon as possible after it changes. In the slave program in which the superior orbit is found, the orbit and its associated observations are sent back to the master program (as are all printed orbits), and the observation set is immediately updated. Therefore, the processing of the rest of the starter pairs containing this particular starter point will be the same as in the serial program. When the master receives an orbit, it checks to see if it is a superior orbit. If so, it updates its observation set by setting flags in its active observation vector. The vector is then broadcast to all of the slave nodes. The slave nodes will not receive this vector, however, until they have finished processing the starter point on which they are working. So, the change on all of the slaves except the one with the superior orbit will be effective with the next starter point that they process. The starter points that are being processed or have already been processed on those slaves are not reevaluated with the updated observation set.

One implication of updating the observation set in this manner is that if we run the parallel version of SAD multiple times with the same observation set, the number of orbits and their solutions will vary from one run to the next. This occurs because the starter points that have been processed when the observation set is updated will be different on each run, as a result of differences in the operating conditions on the nodes in the virtual machine. We discuss the effect of using different configurations of the virtual machine on the results in the next section. This behavior makes it very difficult to test and debug the software, but it is acceptable from the analyst's point of view: any version of the output set will meet his needs.

The most significant rewriting in the code to create the parallel version of SAD was to insert the message passing between the master and slave programs. In some ways this process can be thought of as putting the loop which forms the starter pairs, examines all of the possible solutions to Lambert's problem, and performs the orbit refinement, inside its own subroutine. This subroutine, however, exists as a separate slave program, and the calling parameters are passed by message passing. Because this subroutine did not exist as a separate program in the serial version, we had to spend some time making sure that all common blocks and other variables that are used in this part of the program have the same values in the slave program as they do in the serial version of the program. This requires that some of the initial setup performed in the master program, such as setting up physical constants, be done on all of the slaves as well. This initial setup needs to be done only once on each slave, however, because the slave programs remain active until all processing has been completed.

The second most significant change was to combine the loop that cycles through the different quality sums and the next innermost loop, the loop selecting starter points. In the parallel version these become one loop that cycles through the starter points several times, incrementing the quality sum when the end of the observations set is reached, until all quality sums have been processed. Although this sounds like an inconsequential change, it has a significant effect on the performance of the parallel version of SAD. Consider what happens if these two loops remain separate. Suppose that the number of valid starter points at a particular quality sum is greater than the number of nodes in the virtual machine. (This is typically the case.) The master program will send out the starter points until all slaves are busy. One of the remaining starter observations can be sent out whenever a slave notifies the master that it has finished with its assigned starter point. However, once the last starter point at this quality sum has been passed out, the slaves must sit idle after they finish their starter point until all of the slaves have finished. Typically, there are a few starter points that take much longer than the rest to process, so that the idle time can amount to a significant waste of resources. By combining these two loops into one loop, the slave nodes do not have to sit idle. Instead, they can begin processing starter points from the beginning of the observation set at the next lower quality sum. In this way the only time the slaves need to sit idle is at the very end of the program when there are no more quality sums to process.

In Figs. 4 and 5 we present pseudocode outlines for the master and slave programs, respectively. The structure of the inner loops in the slave program is much like the structure of the inner loops in the

```
Initialize constants
Read and sort observation set
Start up slave programs
Broadcast observation set to slaves
Set quality sum to highest level
Initial loop on starter point until all slaves are busy
    Send each slave a different starter point to process
End initial starter point loop
Loop while any slave is busy
    Check for failure notification
    Receive slave message
    If message type is element set
        Print out element set and associated observations
        If element set is for superior orbit
            Update vector flag indicating which observations are to be used
            in the observation set
            Broadcast new vector flag to all slaves
        End if
    Else if message type is idle slave
        Send next starter point (or starter point from failed slave, if any) to idle slave
        If no starter points left, decrement quality sum (unless already at lowest sum) and
        send first valid starter point from beginning of observation set
    End if
End busy slave loop
Notify slaves to shut down
```

Fig. 4 Pseudocode outline of the master program for parallel SAD. Communication with the slave programs is indicated by italics.

```
Initialize constants
Receive observation set from master
Message processing loop: Receive message from master
    If message type is end
        Exit slave program
    Else if message type is flag vector
        Update the observation set using the flag vector from master
    Else if message type is starter point
        Loop on second observation in starter pair: from the observation after the
        starter point to end of observation set--only those that yield given quality sum
            Loop on number of revolutions between obs in starter pair:
            from $N_{min}$ to $N_{max}$
                Loop on direction of orbit: posigrade and retrograde
                    Loop on two choices of semi-latus rectum in solution to Lambert's problem
                        Obtain initial element set from solution to Lambert's problem
                        Perform a differential correction using only starter pair
                        Orbit refinement loop: perform up to 15 times
                            Predict Fence crossings
                            Search for corresponding observations
                            Perform a differential correction with those obs
                            Save element set if it meets conditions and is the best for this
                            solution to Lambert's problem
                        End orbit refinement loop
                        Save best element set out of all possible orbits for this starter pair
                    End semi-latus rectum loop
                End orbit direction loop
            End number of revolutions loop
            Send best solution for this starter pair to master if it meets conditions
            Remove observations associated with superior orbits
        End starter pair loop
        Notify master slave is idle
    End if
End message processing loop
```

Fig. 5 Pseudocode outline of a slave program for the parallel version of SAD. Communication with the master program is indicated with italics.

serial version of SAD in Fig. 2. This is an indication of the amount of code we were able to reuse in designing the parallel version of SAD. Communication between the master and slave is noted in these figures with italicized text.

Fault Tolerance

After initial testing of parallel SAD, we discovered that there were occasionally reliability problems with some nodes failing before an entire run of the program finished. This prevented the program from completing successfully because the master would wait indefinitely for results from the failed nodes. Therefore, we added some fault tolerance to the software. With the public domain version of PVM, the master program can request to be notified when a node executing a slave program fails. We programmed the master program to keep track of what starter points have been sent to each slave and check periodically for node failures. When a node fails, the master reissues the starter point from the failed node to the next slave node that becomes free. This can lead to some duplication in the output, if the slave on the node that failed had already sent back orbits to the master before it died. Obviously, this fault tolerance will not work if the master node fails, but it does guarantee that if the master program finishes one can be sure that all starter pairs have been processed. It is not, however, a guarantee that the program will finish for all possible catastrophic system circumstances. Our experience has been that failures do not occur for reasons related to the PVM or SAD. Generally they result from people turning off the computer or other programs bringing the computer down.

Analysis

As we noted in the preceding section, the output of the program can vary from run to run because of the handling of superior orbits. In this section we examine the effects that different configurations of the virtual machine can have on the output and run time of SAD.

The hardware used for this testing was the 400-node IBM SP2 at the Maui High Performance Computing Center (MHPCC). Up to 128 nodes can be used in a virtual machine, and we have some control over the method of hardware communication between the nodes.

The data set used for testing was an actual observation set that NSC processed with parallel SAD. The set contained 3848 X-Y-Z TRI observations, which is approximately 60 days worth of uncorrelated observations. After the set was sorted, duplicate observations were removed and a strict 60-day time frame was enforced (via parameters input by the analyst), the resulting set contained 3711 observations.

In Fig. 6 we plot the run time for processing this observation set vs the number of nodes in the virtual machine. The run time here is the actual wall clock time for the process to complete. Because the nodes of each virtual machine on the IBM SP2 at MHPCC are dedicated to run one user program at a time, this is an accurate measure of the time needed to complete the program. We ran multiple tests for each node set. Except in a few cases, the run time was remarkably consistent for a given number of nodes in the virtual machine. The run time varies inversely with the number of nodes in the virtual machine.

The method of communication between the nodes depends on the communication protocol used and the hardware connection between the nodes. On the Maui SP2 there are three combinations of protocol and hardware that can be used[4]: 1) Internet Protocol communication over the ethernet network between the nodes, 2) Internet Protocol communication over the High Performance Switch network between the nodes, and 3) User Space Communication Subsystem over the High Performance Switch network between the nodes.

These configurations are listed in order of increasing communication performance. At the time parallel SAD was written, the public domain version of PVM could not use the User Space Communication Subsystem. The analyst can specify which of the first two configurations to use via the batch file used to submit the program. Figure 6 contains run times for both of the two configurations SAD could use. We found no significant difference in the run times between the two configurations, despite the greater throughput possible with the High Performance Switch. This is because parallel SAD falls into the class of "embarrassingly parallel" applications.

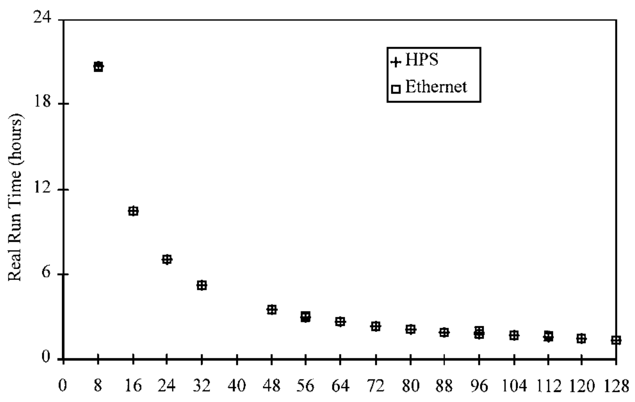

Fig. 6 Run time vs number of nodes in the virtual machine.

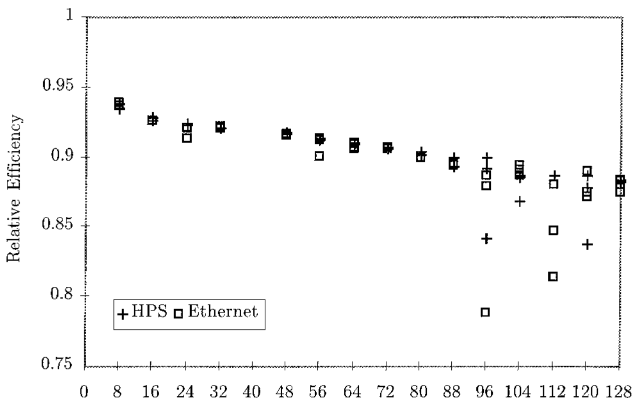

Fig. 7 Relative efficiency vs number of nodes in the virtual machine.

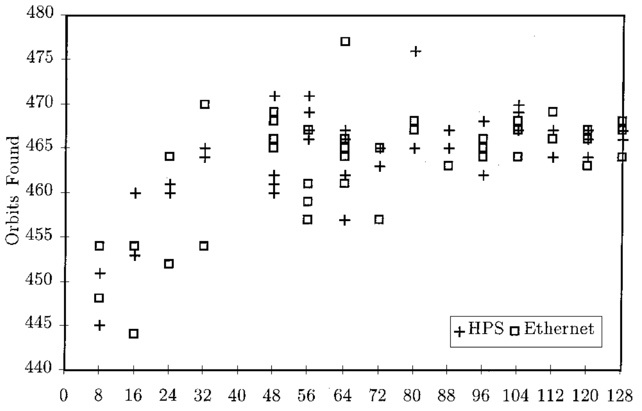

Fig. 8 Number of orbits output vs number of nodes in the virtual machine.

Communications accounts for an insignificant portion of the program run time when compared to the amount of computation, so that using a better form of communication does not result in a noticeable reduction in run time.

The relative efficiency of a program is defined[5] as

$$E_{\text{relative}} = T_1 / P T_P$$

where P is the number of nodes in the virtual machine, T_1 is the time to execute the parallel program on one node, and T_P is the time to execute the program on P nodes. In Fig. 7 we plot the relative efficiency for our test case vs the number of nodes in the virtual machine.

The relative efficiency is a measure of the amount of time that the slaves are doing useful work. The fact that the relative efficiency decreases only slightly for large numbers of nodes in the virtual machine indicates that the parallel version of SAD is scalable—able to effectively utilize whatever number of nodes it is given. The small decrease in efficiency for large virtual machines can be partially attributed to the increased likelihood that two or more nodes will request work from the master at the same time. In such a case some of the slaves will be idle until the master has time to service them. Part of the decrease in relative efficiency may be as a result of the increased communication load that occurs with a large virtual machine.

The serial version of SAD finds 371 orbits from this data set. The average number of orbits found with the parallel SAD from this observation set was 465, a 25% increase. Only five of these orbits qualified as superior orbits. If the observations associated with superior orbits were not removed from the observation set, the program found 499 orbits. This is the upper limit on the number of orbits the program will find for any configuration of the virtual machine. In Fig. 8 we show the number of orbits output vs the number of nodes in the virtual machine. The configuration of the communication method had no significant effect on the number of orbits found. There is a slight trend toward fewer orbits being found when we include fewer nodes in the virtual machine. When there are fewer nodes in the virtual machine, fewer of the starter points succeeding the starter point that had a superior orbit can be processed before the observation set is updated. This results in fewer duplicate incarnations of the superior orbit and an output more similar to what would be produced by the serial version of SAD.

Table 1 Results of operational runs of parallel SAD by NSC

Date of run (yymmdd)	No. of obs	Orbits found by SAD	Orbits accepted by analyst	No. of lost objects	No. of new objects	No. of nodes	Run time, hr
950512	3657	693	19	4	11	8	NA
950525	3737	489	28	5	10	64	2.07
950609	3835	587	43	11	14	64	2.91
950810	4174	463	26	6	10	64	3.48
950915	4897	697	60	5	35	64	3.11
951005	4972	854	29	4	20	64	3.50
951019	6231	1050	81	9	40	128	2.59

Application

For operational use we implemented parallel SAD on the Air Force's IBM SP2 parallel computer located at Kihei, Maui, Hawaii. The program has been in operation since May 1995. For a typical execution the analyst at NSC sends the observation set via file transfer protocol to the SP2 and then logs on to one of the interactive nodes of the SP2 via a U.S. Department of Defense (DoD) terminal access communication (TAC) connection, which was a dedicated DoD dial-up communication system. The analyst submits a standard batch file that puts the program in the LoadLeveler queue for an allocation of nodes. The batch file sets up the number of nodes to use, the configuration of the communication systems, and the location of the observation set. Generally, the analyst will rarely have to change the batch file. The program typically will receive its allocation of nodes within 24 hours after the batch file has been submitted. The necessary PVM setup is then done automatically, and the program begins running. After the program has finished, the analyst again FTPs to the SP2 to retrieve the results.

In Table 1 we present statistics for some actual runs of SAD by NSC. Now that the program can be run for a full 60 days of uncorrelated observations the number of orbits found is greatly increased, not only because more orbits are contained in a longer time span, but also because the possibility of finding enough observations during orbit refinement to support a particular orbit has increased. Typically the program produces several hundred orbits. The analyst postprocesses this mountain of data using other software to find the best orbits, which will be inserted into the catalog. This will generally result in several dozen new orbits for the catalog. One of the chief benefits of parallel SAD is that the 60 days of data can be postprocessed all at once. With the serial version of SAD, the same amount of data would involve several postprocessing sessions, one for each of the several smaller SAD runs needed to process that much data. The number of orbits inserted into the catalog is highly dependent on the amount of time the analyst has to postprocess the SAD results. Some of these orbits may not be sustained for long time periods and may subsequently become lost objects that might be discovered again at a later time.

In Table 1 we present information from an analysis of the orbits that the analyst added to the catalog. By comparing the element sets added to the catalog with a historical database, the analyst can determine how many of the orbits were once tracked, then lost, and then rediscovered with SAD, as well as the number of completely new orbits. The remaining orbits (e.g., 13 = 28 − 5 − 10 for the 950525 run of SAD) are objects that belong to an existing object that NSC already has in the catalog.

The capability to process much more data with the parallel version of SAD has increased the program's proficiency in finding certain types of orbits. Many orbits, including orbits with perigee in the southern hemisphere, semisynchronous orbits, and low-inclination orbits, are only rarely observable by the NSC Fence, sometimes resulting in only one observation per day or less. When SAD (the serial version) could only process five days of data, there were not enough observations from such orbits to create and refine an element set. With 60 days of data, it is far more likely that there will be enough observations for parallel SAD to support an element set.

Implementation at NSC

We present a great deal of detail in this section on how this program was implemented at NSC to illustrate the concept that one program can function efficiently as a parallel-processing program and as a serial program with very little overlap and yet with maximum reuse of code between the two modes.

After the initial experience with running SAD at Maui, it was decided to implement the program at NSC. A set of workstations used as personal computers by various staff members was allocated to form a virtual parallel-processing system. This version of the software turned out to be a new development. The version of the software that was used for the earlier program was based on an old version of SAD that had run on the NSC Cyber computer. Since 1995 the serial program had been significantly changed to accommodate new coding standards and to run on the new NSC distributed processing environment based on IBM workstation computers. Thus we started from scratch with the new version of the program and incorporated the programming modifications we had developed for the earlier version. There were two problems we had to address in this new program: one was the incorporation of a new user interface that conformed to display requirements of the NSC operational system and the other was a desire to run the program on a single computer as well as on multiple computers.

Several ways were thought of to run the parallel program on a single computer. We discuss several possible implementations, each has its advantages and disadvantages.

The master and slave Programs can coexist on a single computer with the PVM messages passed between the master and slave processes within this computer. A disadvantage to this method is the overhead associated with running two processes that really do only one job and the unnecessary message passing between the internal processes. One of the advantages to this is that there is only one system of programs to develop and maintain. Moreover, there is virtually no overhead associated with running within the PVM environment. Still a disadvantage is that PVM must be running to support the master and slave programs.

A second method is to develop and maintain two completely separate programs, a serial program that has no parallel-processing code in it and the fully capable parallel program. The advantage to this approach is that there is no complex software required to enable one program to perform both functions. The obvious disadvantage is the extra effort required to maintain the two programs. Although both programs could share low-level modules, the configuration control over two almost identical programs can be difficult to maintain.

A variation of the preceding method is to encompass in one program the capability of running in a serial mode or a parallel mode. This lessens the problem of code maintenance because only one program would be maintained. Also, if a change is made to the SAD algorithm, only one executable needs to be recompiled.

We accommodated NSC's request by using the following approach. First, the bulk of the work of SAD was isolated into a new subroutine SADWRK. This subroutine takes a starter ob as input and will process that point against all applicable secondary obs. This subroutine was created because this code will now be linked into two different programs: once in the slave program as before and now also in the master serial program. SADWRK is executed from the master program only when the master is functioning in serial (one computer) mode. Otherwise SADWRK is called from the slave program.

Creating this subroutine provided some challenge. We first identified the largest block of code that was common to both the serial program and the slave program and isolated in the subroutine SADWRK. Each of the variables used in the SADWRK had to be checked by the programmer to see if it was unique to the subroutine or had to be passed in by the calling program. Forty variables ended up being passed into the new subroutine.

The next step was to create a logical variable serial that would guide the execution path of the program. Serial is true unless parallel is given as a call line flag or if the job is scheduled by the NSC scheduling process. NSC has a system of scheduling jobs where the operators can request that a job be run at a certain time or at certain times on a group of scheduler machines, instead of running the job on their own workstations at the present time. Because it is usually very large jobs that are scheduled and because it is known that the scheduler machines will always have access to PVM, all scheduled SAD jobs are set to run in parallel mode.

When the program runs in serial mode, the code follows the same path as when it runs in parallel but if statements are used to bypass certain PVM calls. For instance, running in serial mode the subroutine that starts the PVM daemons is not called, nor is the subroutine that spawns slave programs on other machines. The serial code essentially acts as the master in parallel mode, but without ever communicating to a slave. A starter point is picked in both modes, but what is done with this point is different. In parallel mode the starter point and other initial information is sent to the slave. The slave program receives this information and then passes it into the SADWRK routine. However, in serial mode the starter point is passed directly into the SADWRK routine.

One of the interesting things about SAD is that whenever an element set is found the program exits all of the way back up to the user interface, where the element set is stored and then goes back down into the working part of the program. This must be done because SAD takes so long to run, and if the program exits prematurely results would otherwise be lost. This is a particular concern with SAD because it can frequently run for several days. This behavior provided a challenge when modifying the program to run in either serial or in parallel mode.

When running in parallel, the slave calls SADWRK, which finds the element sets. SADWRK sends each element set directly back to the master subroutine. When the master gets an element set, it passes the element set to the user interface. The top-most SAD program consists of the user interface and the master program controller. The master subroutine, when it exits to the user interface, has to save its environment (the values of all of its variables) so that it can resume execution with the same environment after return from the user interface. Meanwhile the slave continues to run SADWRK looking for more element sets. The slave process, being an independent process, is never exited and, therefore, does not have to save its environment.

However, when running serially, SADWRK must pass control back to the calling master routine whenever it needs to pass the element back to the user. SADWRK cannot continue to run because SAD is running as a single process. When running in serial mode, SADWRK must save its place (local variables) and then exit to the master that, in turn, exits to the user interface. Local variables in SADWRK are saved much the same as the master environment in parallel mode. To handle these two possible cases, if statements had to be placed into the SADWRK routine. After the element set is discovered, there is an if statement to determine if the program is in serial or parallel mode. If it is parallel, a series of PVM calls are made to pass the element set to the master. The master will then exit to the user interface as just described. The slave will continue to run, processing other starter points. If it is serial, the program will exit out of SADWRK with a flag set so that it will know an element set has been found.

The parallel mode of SAD involves a lot of bookkeeping to keep track of all of the slaves. The master program keeps track of how many slaves were active and how many slaves had been sent starter points. To get the serial mode to function properly, some of this bookkeeping had to be simulated. Basically, when the program is executed in serial mode, the program thinks that it has one slave that it needs to pass starter points to. But instead of passing it by PVM, it simply passes the information in an argument list into the SADWRK subroutine. SAD increments a processing list number when it calls the subroutine so that it will know that it now must process what the subroutine found.

When messages come from PVM, they are marked with a message tag so the master can know what sort of message it is receiving. This way the master knows if it is going to receive an element set to process, or if the slave is simply saying that it did not find one and now wants a new starter point. To get the serial mode to function the same way, the message tag was made into one of the call line parameters in the SADWRK subroutine. So when the subroutine comes back, SAD can use the same code to examine what the subroutine has found. If an element set was not found, it decrements the processing list number back down to zero and provides the next starter point, so that on the next loop it will pass the new starter point to the subroutine. If an element set was found, the processing list number is set back to zero because SAD must go back into SADWRK to finish processing the starter point. However, the starter point is not changed, and the value of the secondary point is saved because when it goes back into the subroutine the next time it must pick up where it left off with that starter point, not start with a new one.

Conclusions

The parallelization of SAD provides a lesson on how to reuse existing serial software in a parallel-processing environment with minimal development cost. The level of effort expended by the Naval Research Laboratory was approximately two to three man-months.

For operational Commands this effort opens the prospect of using external resources for many operational problems. Most operational Commands are hesitant to use outside resources, citing problems of availability and reliability. However, for this problem these are not significant issues. The average turnaround time for a run of SAD on the IBM SP2 at Maui is certainly acceptable. Although the use of this program has significantly enhanced productivity, the limiting factor on its use is now the manpower to process the results, rather than computational resource limits. Operational Commands have many such problems that can substantially benefit from using more computational resources than the Command can allocate locally, but which can be obtained through networking. By parallelizing programs judiciously with PVM or other multiplatform tools, the software can easily be moved to other computing resources when they become available, even a local resource or a heterogeneous combination of local and remote resources if desired. This facilitates a long-term strategy for minimizing software development costs in which Commands can meet the processing needs of today with outside resources while developing the computational resources needed for the future.

Acknowledgments

This project was developed under contracts with the NSC. We gratefully acknowledge support from Steven Knowles, Neal Melson, Edna Jenkins, Deborah Perini, William Synan, and other staff members of NSC.

Through the use of the Maui High Performance Computing Center, this research was sponsored in part by the Phillips Laboratory, Air Force Materiel Command, U.S. Air Force, under cooperative agreement number F29601-93-2-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Phillips Laboratory, the U.S. Government, the University of New Mexico, or the Maui High Performance Computing Center.

References

[1]Prussing, J. E., and Conway, B. A., *Orbital Mechanics*, Oxford Univ. Press, New York, 1993, Chap. 4.

[2]Schumacher, P. W., and Glover, R. A., "Analytical Orbit Model for U.S. Naval Space Surveillance: An Overview," AAS/AIAA Paper 95-427, Aug. 1995.

[3]Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V., *PVM—Parallel Virtual Machine—A Users' Guide and Tutorial for Network Parallel Computing*, MIT Press, Cambridge, MA, 1994.

[4]IBM Corp., *SP2 Administration Guide*, Kingston, NY, 1994.

[5]Foster, I., *Designing and Building Parallel Programs*, Addison Wesley Longman, Reading, MA, 1995.