```
\documentstyle[11pt]{article}
\textwidth=6.5in
\textheight=8.5in
\oddsidemargin=0.0in
\topmargin=0.0in
\parskip=0.05 in
\newcommand{\bigo}[1]{{\bf O}($#1$)}

\begin{document}
\bibliographystyle{plain}
\pagestyle{plain}
\centerline{\bf {\LARGE Parallel Split-Level Relaxation\footnote{This work was
supported in part by NSF Grants MCS-8221750, DCR-8506393, and DMC-8502115}}}
\vskip 0.35 in
\centerline{\large Tom Henderson and Ashok Samal}
\vskip 0.2in
\centerline{Department of Computer Science}
\centerline{University of Utah, Salt Lake City, UT 84112}
\vskip 2.0 in
\centerline{\underline{\bf {\Large Abstract}}}
\vskip 0.35 in

The goal of high level vision is to identify a set of regions in a given
image.  This has been called by various names: the scene labeling problem
\cite{BallardBrown82}, the consistent labeling problem
\cite{HaralickShapiro79-I}, the constraint satisfaction problem
\cite{Mackworth77}, Waltz filtering
\cite{Waltz75}, the satisfycing assignment problem \cite{Gaschnig79}, etc.

There are several approaches to solve this problem, including backtracking,
graph matching, and relaxation.  A new method called {\it split-level
relaxation}, which is based on discrete relaxation was proposed in
\cite{HendersonSamal86,HendersonSamal86a}.  It takes care of multiple semantic co
nstraints, by
considering each of them independently.  The problem is known to be
NP-complete, so it takes a long time to solve this problem.  With
the advent of multiprocessors, it is now imperative to see if the problem can
be solved faster in the average case.

In this paper we give a framework for solving the scene analysis problem in a
parallel processing environment, using split-level relaxation.
Experiments done on a multiprocessor show that indeed it is
advantegeous to use multiprocessors to solve this problem.

\newpage

\section{Introduction}

In this paper we give a framework for solving the scene analysis problem in a
parallel processing environment, using split-level relaxation.  Initial
experiments are done on a multiprocessor to show its suitability.  The rest of
the paper is organized as follows.  In section 2, the consistent labeling
problem is formally defined.  The traditional ways to solve the problem are
also described.  The scene analysis problem, as used in the field of computer
vision is formulated as a CLP (consistent labellig problem) in section 3.  The
{\it split-level} relaxation
process is described in section 4.  In section 5 the features of the BBN
Butterfly multiprocessor which were used for our results are discussed.  There
are several ways one may extract parallelism from the split-level relaxation
algorithm.  They are discussed in section 6.  The details of the
implementation and the results are given in section 7.  Finally, the
conclusion and future research directions are briefly discussed in section 8.
```

\section{The Consistent Labeling Problem}

Before the problem is formally defined, a few terms are explained.

\begin{itemize}

\item A {\it unit}, ${\it u_i}$, is an item that has to be assigned a value or meaning. For example, in scene labeling, it could be a region obtained after segmenting the image.

\item A {\it label}, ${\it l_i}$, is the value or the meaning that is with be associated to a unit. In the same context it could, for example, be grassland, water, etc. Usually there is only a fixed set of labels (called the domain of the unit) that can be assigned to a unit.

\item A {\it unit-label}, ${\it L_i = (u_i,l_i)}$, is a pair consisting of a unit and its associated label.

\item A {\it labeling}, ${\it {\bf L}}$, is a set \{$L_1, L_2, \ldots , L_i$\} of unit-labels. A labeling is {\it complete} if there is a label for all the the units; otherwise it is called a {\it partial} labeling.

\item A unit can be assigned any value from its domain. In general, however, there is are restrictions on the labels a set of units can have simultaneously in order to be consistent. We call these the {\it constraints}. They can be n-ary in general, but we restrict ourselves here to binary constraints only. The formulation can be extended to the general case easily. Thus, a constraint set {\bf R} is a set, \{$r_1, r_2, \ldots , r_k$\}, where each r_i is a pair of unit-labels. If ({\it $(u_i,l_i),(u_j,l_j)$}) $\in$ {\bf R}, it means the unit u_i can not have the label l_i, when u_j has the label l_j.

\item A labeling ${\it {\bf L}} = \{L_1, L_2, \ldots , L_k$\} is {\it consistent} iff $(L_i,L_j) \not\in {\bf R}, \forall\ i,j \leq k$.

\end{itemize}

Using the above definitions, the consistent labeling problem, can be formulated as follows. Given,

\begin{itemize}

\item A set of units, \({\bf U} = \{{\it u_1, u_2, \ldots , u_n}\}. \) Without loss of generality we assume that all the ${\it u_i}$'s are distinct.

\item Each unit ${\it u_i}$ has an associated domain ${\it D_i}$, which forms the set of possible labels for the unit. We define \({\bf D} = \{{\it D_1, D_2, \ldots , D_n}\}. \) Often, however, all the units take the values from the same domain, in which case \({\it D_1 = D_2 = \ldots = D_n = D} \).

\item A constraint relation {\bf R} which defines the constraints between the units and the labels.

\end{itemize}

the goal of the consistent labeling problem is to find a {\it complete, consistent} labeling. Thus,

\vspace*{0.15in}
\centerline{ {\bf CLP} $\Longleftrightarrow$ ({\bf U,D,R}) .}
\vspace*{0.10in}

There are several variations to this formulation. For example, instead of

obtaining a complete labeling one might want the {\it best} labeling according to certain criteria. Another variation is to ask for the maximal labeling if a complete labeling is not possible. Sometimes all the consistent labelings may be required. The nature of the problem, however, remains unchanged.

\subsection{Complexity of CLP}

It has been proved that CLP is NP-complete \cite{Montanari74}. Hence the algorithms that solve this problem are necessarily exponential in the worst case. So, it is not important to compare the performance of the various algorithms in the worst case. What is more important is to compare their performance in the average case. Even here, it is difficult to analyze the algorithms. As Knuth \cite{Knuth75} points out, it is difficult to analyze these algorithms without actually running them. Even then, the results may not be valid in a totally different problem domain. Gaschnig \cite{Gaschnig79} took an experimental approach to study the performance of the algorithms. Similar results are also presented in \cite{HaralickElliot79}.

\subsection{Solutions to CLP}

The consistent labeling problem can be solved in several ways. The simplest solution is the {\it generate-and-test} method. Here all possible labelings are enumerated and then the ones which are consistent are selected as the solutions. Clearly this method is very naive and inefficient when either {\bf U} or D is a large set. For example, if $\left|{\bf U}\right|$ = 10 and $\left|D\right|$ = 10, then the number of possible configurations is 10^{10}. In many cases the labels assigned to the first few units make the whole labeling inconsistent due to the nature of the constraints. If detected early, saves a lot of computation.

The {\it standard backtracking} method takes advantage of the above observation. A single unit is assigned a label from its domain. Then another unit is chosen and it is given a label such that the new partial labeling is still consistent. If at some point it is not possible to find such a label, the process is backed up to the last unit that was assigned a label. It is given the next possible label and this process continues. If all the units are labeled without violating any constraint we have found a solution. If however we run out of labels at some point during this process, there is no solution.

Standard backtracking is more efficient than the generate-and-test method, but it is still not good enough for many practical applications. Several efforts have been made to improve the performance of the backtracking approach. Gaschnig \cite{Gaschnig79} developed two new backtrack-type algorithms which performed better than the standard backtracking. In the {\it backmark} algorithm all redundant tests for checking the consistency between pairs of units are eliminated. The {\it Backjump} algorithm is like the standard backtracking except it is possible to jump back more than one level when a failure occurs. Yet another approach called {\it forward checking} is proposed in \cite{HaralickElliot79}. It performs better than the two previous algorithms in some cases, but the results are known to be valid only for the problems where each unit is constrained by all other units. Haralick {\it et al.}, \cite{HaralickEtal78} have also described two {\it look-ahead} operators, Φ and Ψ, to reduce the computation during the backtracking process. Haralick and Shapiro generalized these operators to incorporate arbitrary lookahead.

Another approach was taken by Waltz \cite{Waltz75}. The idea is to remove labels that are not consistent with any other label of other units. The removal of a label from one unit may in turn force other labels to be deleted. The changes are propagated and the process terminates when no

further labels can be deleted, or an unit loses all its labels. In the latter case there is no solution to the problem. In the former case, when the process converges to a consistent set of labels, it is still necessary to go through a search procedure to get an unambiguous labeling. The rate of convergence depends on the nature of the constraints and the labels. A modified version of this method is described in \cite{RosenfeldEtal76}. It is a parallel iterative procedure that allows probabilities to be associated with the labels.

Yet another approach to reduce the computation during the backtracking process was taken by Mackworth \cite{Mackworth77}. For binary constraints, the problem can be formulated using graphs, where nodes represent the units and the constraints are denoted by the arcs between the nodes. Each node of the graph has an associated label set. Such a network is called a {network of constraints}. Fundamental properties of such networks are explored in \cite{Montanari74}. Mackworth observed the thrashing behavior of the backtracking algorithms and developed three consistency tests: {\it node}, {\it arc}, and {\it path} consistency tests, which when enforced first may reduce the computation time of the backtrack programs drastically. Several algorithms to enforce this are also given in \cite{Mackworth77}. Mohr and Henderson \cite{MohrHenderson86} developed an optimal algorithm for arc consistency and an improved algorithm for path consistency.

\subsection{Applications of CLP}

Many problems can be formulated as CLP. Gaschnig \cite{Gaschnig74,Gaschnig79} used this approach to solve the n-queens problem, the Soma Cube, Instant Insanity, and the cryptarithmatic problem. Henderson and Davis \cite{DavisHenderson81,HendersonDavis81} used it for syntactic shape analysis. As shown in the next section, the scene analysis problem can also be formulated as a CLP.

\section{Scene Analysis as a CLP}

The goal of scene analysis is to locate and identify all the objects in a given image. There are several ways one might attempt to solve the problem. One approach is to first locate the features (e.g., holes, corners, etc.) in the image. Then use these features and the constraints between them to identify the objects. Thus, the features constitute the set of units, {\bf U}, the possible objects in the scene are the labels, D, and the relations between them define the constraint relation, {\bf R}.

\subsection{Models for the objects}
A model for an object is defined using the location and the orientation of the features, with respect to a fixed coordinate frame. These features define the label space or the domain of the units in the CLP. There are also constraints between these features, which in general may be n-ary. In practice however, it is often sufficient to consider only the binary constraints. Thus each model is completely defined by the features and the constraints, i.e, $M_i \Leftrightarrow (F^m_i, C^m_i)$, where F^m_i is the set of features and C^m_i is the set of constraints for the i^th model, respectively.

It should be noted that the domain of a feature found in the image can contain a feature in any of the possible models. If the scene can have k possible objects, then the model set {\bf M} = \{ $M_1, M_2, \ldots, M_k$ \}. Another important observation is that even though all the constraints are binary, they are not always the same. For example, consider two holes H_1 and H_2 in an image, such that H_1 is both adjacent to and bigger than H_2. The two constraints can be expressed as binary predicates {\it adjacent\/}(H_1,H_2), and {\it bigger\/}(H_1,H_2). Here even though both are binary constraints, they are of different {\it type}. The {\it adjacent} constraint is symmetric while the {\it bigger} is not. The semantics of the two constraints are also very different. Hence, we group the constraints according to their types.

Since a constraint can be represented as a relation, the constraint set of a model can be expressed as a set of relations. Thus,

\[C^m_i = \{ R^m_{i,1}, R^m_{i,2}, \ldots, R^m_{i,r} \},\]

where

$R^m_{i,j} \subseteq F^m_i \times F^m_i, 1 \leq j \leq m$ and r is the number of constraint types. The constraint set for the set of models is given by $C^m = \cup_{i=1}^{m} C^m_i$.

\subsection{Information from the image}
\label{subdivide-section}

Low level image processing operations are done in order to get all the features of the image. Now these features are used to do the recognition process, i.e, to associate these feature instances with a feature in any one of the models. The features obtained from the images are also constrained similar to the features in the models, e.g., two line segments may be parallel, one may be longer than the other, etc. Here, also, one may find several kinds of constraints between the features, some of them may even be different from the constraints in the models. However, we choose only those kinds of constraints which are used to define the constraints for the models.

Hence, the information {\bf S}, from the scene can be represented as:

\[{\bf S} = (F^s,C^s),\]

where

F^s is the set of features found in the image and $C^s = \{ C^s_1, C^s_2, \ldots, C^s_r \}$, $C^s_i \subseteq F^s \times F^s$.

There is a direct correspondence between the two sets of constraints C^s and C^m. For each relation in C^s_i in C^s there is an equivalent relation C^m_j in C^m which has the same physical interpretation, although the domains of the two relations are different.

\subsection{Scene analysis as CLP}

In the previous sections we defined the information that can be obtained from the image and the models. Now using this we formulate the scene analysis problem as a CLP. The set of units that need to be labeled is the set of feature instances found in the image. The label set or the domain for each of the features may consist of any feature in the any model with the same type. For example, if we find a hole in the image, then it could be any hole in any of the possible models. It is possible to narrow it down further by using their dimensions as well. See \cite{HendersonSamal86} for several ways to make it more efficient. For simplicity, however, we assume that the domain consists of the union of all the features of all the models. Also, we assume that no two features in the models have the same name. This can be easily avoided by using a naming scheme where the name of the model is also a part of the name of a feature. The constraints in this case are the constraints that are between the features in the image as well as the constraints between the features in the models.

So, the scene analysis (SA) problem can now be formulated as:

\vspace*{0.2in}
\centerline{ SA $\Longleftrightarrow$ ({\bf U, D, R}),}

```
where

$
{\bf U} = F_s,
D = {\cup}^{n}_{i=1} F^s_i, \ and\  {\bf R} = ({\cup}^{k}_{i=1}C^m_i) \cup C^s
$


\subsection{Example}
\label{example-section}
We now present an example, to explain the above concepts.  Figure
\ref{example-figure}  shows two simple industrial parts similar to ones in
\cite{Nitzan82}.
We will refer to them as $Part_1$ and $Part_2$.  They are not as complicated as
many other parts, but they are only used for illustration.  We
use only the boundary edges as the features.  Other useful features can be
holes, corners, etc.

\begin{figure}
\vspace{2.5 in}
\caption{Two industrial parts used in the example}
\label{example-figure}
\end{figure}

\begin{tabbing}
\hspace*{1.0in}\= \kill
\>{\bf M} = \{ $Part_1, Part_2$ \}\\
\>$Part_1 = (F^m_1,C^m_1)$ and $Part_2 = (F^m_2,C^m_2)$\\
\>\\
\>$F^m_1 = \{ Y_1Y_2, Y_2Y_3, X_4Y_3, \ldots, Y_1Y_6, Y_1Y_2 \}$\\
\>$C^m_1 = \{parallel, perpendicular, longer, equal-length \}$\\
\>\\
\>$F^m_2 = \{ Z_1Z_2, Z_2Z_3, Z_3Z_4, Z_4Z_1 \}$\\
\>$C^m_1 = \{parallel, perpendicular, longer, equal-length \}$\\
\end{tabbing}

We have four relations to express the constraints in the model for
$Part_1$. These four relations are given in Table \ref{constraints-table}.
These
relations are not complete, but give an idea what the relations look like.
$Part_2$ has similar relations to express its constraints.

\begin{table}
\begin{center}
\begin{tabular}{||c|c|c|c||}   \hline
{\it Parallel}   &{\it Perpendicular}   &{\it Longer}   &{\it Equal-Length}\\\hli
ne
$(X_3Y_4,\ X_1Y_8)$ & $(Y_1Y_2,\ Y_2Y_3)$ & $(Y_1Y_6,\ Y_1Y_2)$ & $(Y_1Y_2,\ Y_5Y
_6)$\\ \hline
$(X_1Y_8,\ X_2Y_7)$ & $(Y_2Y_3,\ X_4Y_3)$ & $(Y_1Y_6,\ X_3X_4)$ & $(X_3X_4,\ X_1X
_2)$\\ \hline
$(X_2Y_7,\ Y_5Y_6)$ & $(X_4Y_3,\ X_3X_4)$ & $(Y_1Y_2,\ Y_2Y_3)$ & $(X_4Y_3,\ X_3Y
_4)$\\ \hline
$(Y_1Y_6,\ X_3X_4)$ & $(X_3X_4,\ X_3Y_4)$ & $(X_3X_4,\ Y_2Y_3)$ & $(X_1Y_8,\ X_2Y
_7)$\\ \hline
\end{tabular}
\end{center}
\caption{Constraint relations in $Part_1$}
\label{constraints-table}
\end{table}

\section{Split-Level Relaxation}
One of our main motivations for modifying the structure of the standard
```

discrete relaxation procedure is that there is no way to use positive information. In relaxation all the labels of a node are treated uniformly. For example, suppose that there is evidence that some labels of a unit are more likely that the others. The relaxation process works just the same as the case where there is no such evidence. One could go to a stochastic model, but it unduly complicates the process. We will take another approach without assigning a probability value to each each label of each unit. First we describe the network model used for the relaxation procedure.

\subsection{Network model for relaxation}
Here the underlying network/graph model for split-level relaxation process is explained. As mentioned before, the units are represented by the nodes of the graph and the arcs denote the constraints. In that respect, our model is similar to that explained in \cite{Mackworth77}. However, there are several major differences. What we have here is conceptually closer to a set of graphs rather than a single graph.

Each of the graphs models a single relation, which represents only one kind of constraint. So, if there are r types of constraints between the features in the
image, then there are r corresponding graphs. There are also r graphs for each of the models that is considered. We call these graphs image and model graphs, respectively.

\subsubsection{Model graphs}

Each model graph G^m_i is a composite graph consisting of a set of graphs each corresponding to a single constraint. Thus,

\[G^m_i = \{ G^m_{i,1}, G^m_{i,2}, \ldots, G^m_{i,r} \},\]

where

r is the number of constraint types in the graph. The nodes of the graph $G^m_{i,j}$ are the set of features in the model, and the arcs are represented by the constraints. So,

\[G^m_{i,j} = <F^m_i, E^m_{i,j}>, \]

where
$(x,y) \in E^m_{i,j}\ \ iff\ \ (x,y) \in C^m_{i,j}$

Recall that $C^m_{i,j}$ is the relation that represent the j^th constraint in the i^th model.

We also denote G^m to denote the composite graph consisting of all the model graphs in the graph. So,

\[G^m = \{ G^m_1, G^m_2, \ldots, G^m_r \}\]

\subsubsection{Image graphs}
\label{image-graph-section}

In contrast to the model graphs, there is only one image graph, although it is also possible to have multiple image graphs if we are looking at multiple images at the same time, or labeling different parts of one image at the simultaneously, in a disjoint fashion (see \cite{HendersonSamal86}). The composite image graph, G^sconsists of a set of graphs corresponding to the constraint types. Thus,

\[G^s = \{ G^s_1, G^s_2, \ldots, G^s_k \},\]

where

k is the number of constraints in the graph. Similarly,

\[G^s_i = <F^s_i, E^s_i>,\]

where

$(x,y) \in E^s_i \ \ iff\ \ (x,y) \in C^s_i$.

\subsection{Split-level relaxation}

Since our network model is different from the standard network models, the relaxation process is also very different from others. The first step of course is to build all the graphs: model and image graphs. Also, the possible labels for the units in the image graphs (see \ref{image-graph-section}) are also associated with the corresponding units. It is important to note that although the arcs of the image graphs are different, their nodes are the same. Hence they can be shared in an actual implementation. The topology of all the graphs remains the same during the relaxation process. Only the label sets of the units in the image graphs may change. The models graphs remain totally unchanged; they are essentially used to check for the validity of constraints between the features in the models.

After the graphs are built, the next step is to enforce the node, arc, and path consistency checks in the image graphs. Here is where the system lends itself to parallel execution. It will be explored further in section \ref{parallelism-section}. Here only the structure of the split-level relaxation algorithm is described. We divide the nodes of the graph into two categories: {\it strong} and {\it weak} nodes. In contrast, standard relaxation treats all the nodes equally. The strong nodes signify positive information, and always remain {\it strong} during the whole process. The {\it weak} nodes however, may be elevated to the {\it strong} status. During the relaxation process the strong nodes can affect the label sets of the other nodes, both strong and weak. The {\it weak} nodes however, can affect only the label sets of weak nodes. This causes the bad information, which are in the form of weak nodes, to have much less effect than the strong nodes, which signify positive information. After some initial iterations, the weak nodes that survive are elevated to strong nodes. The ones that don't survive the initial iterations are not considered further and are treated as features that are not part of any model. The enforcement of consistency is done until there is no change in the label set of the nodes.

Other ways to solve the problem of ``weak nodes'' deleting the labels of the ``strong'' nodes are explained in \cite{HendersonSamal86}. Several issues considering the efficiency of the algorithms are also discussed there.

\section{The BBN Butterfly Parallel Processor}

Since the results in this paper are obtained on the BBN Butterfly parallel processor, a brief discussion of the main features of the machine is given in this section. The Butterfly is a shared-memory {\it MIMD} machine capable of having upto 256 processors. All the processors in the machine are identical. Each processor may run its own program independently of the other processors. The memory is shared between all the processors and may be used to communicate between the processors. The memory is physically partitioned among the processors, but each processor can access the memory of every other processor. Thus even though it is tightly coupled, it is not a true shared-memory multiprocessor. In contrast, the Cosmic Cube \cite{Seitz85} is loosely coupled, where a processor can access only the memory that is local to it. An example of truly shared memory machine is the Sequent \cite{Sequent85}.

\subsection{Hardware}

The Butterfly hardware consists of two main subsystems, the processor nodes and the butterfly switch. The processor nodes are responsible for the computing job of the machine, while the switch forms the communication system of the machine, and is responsible for the communication between the processor nodes.

The Butterfly parallel processor can have up to 256 processors nodes. Each processor node consists of an MC68020 microprocessor, a {\it processor node controller} (PNC), and up to 4 megabytes of memory. The machine used to obtain the results in this paper has 18 processors, with all but two nodes with 1MB of memory. Each processor node has a M68881 floating point co-processor along with the MC68020 microprocessor.

The PNC occupies a central role in the machine architecture. All the accesses to the memory including the local memory accesses are directed to the PNC. It uses the switch for the remote memory references. In addition it performs several other operations, e.g., atomic operations, I/O interfaces, etc.

All the processor nodes are connected through a switching network called the Butterfly switch. It is a self-routing, packet switching, non-blocking Ω network. One important characteristic of the Butterfly switch is that the number of switching elements grows as $N \log N$, where N is the number of processing elements. The bandwidth of the network grows linearly with the number of processors. The maximum data transfer rate between two processors nodes is 32Mbit/second per channel. The ratio of time taken for a remote memory access to a local memory access is roughly 5 to 1 (see \cite{Tinker87} for some tests and precise numbers).

\subsection{Software}

The application programs run on the Butterfly under the Chrysalis Operating System. Currently C and a few other languages are supported. All the low level Chrysalis subroutines can be called from the application program. These include primitives for process creation, synchronization, creation of memory objects, creation and maintainance of queues, etc.

An application library called the Uniform System is also provided. It supports a methodology for programming, where all the processors share a common address space. It is written on top of the Chrysalis system. The application is structured such that all the available processors are used. While it is convenient to use for many applications, it is not very flexible. The programs written for obtaining the results in this paper are written using the low level Chrysalis functions directly. The features that have direct bearing on our application are discussed in section \ref{implementation-section}.

The memory management system of the Butterfly has important consequences in the design of algorithms. The first Butterfly was designed using the MC68000, which supports only 24 bit addresses. Now even though MC68020 which support 32 bit addresses, processors are used, only 24 bit addresses can be used because of other hardware restrictions. So, the problem here is to map the 24 bit virtual address to a 32 bit physical address. The details of this scheme
are discussed elsewhere
\cite{Butterfly-overview,RAMFile-manual,Chrysalis-manual}.

\section{Parallelism in Split-Level Relaxation}
\label{parallelism-section}

In this section we explain how the split-level relaxation process works in a

parallel processing framework. It is shown that it is really advantageous to run this on a parallel computer since there is a tremendous amount of parallelism inherent in the algorithm. There are actually three levels of concurrency that may be exploited during this process.

At the topmost level, it is possible to look for objects in different parts of the same image. It can be done in an efficient manner as explained in \cite{HendersonSamal86}. We have not tried to exploit this level of parallelism. Split-level relaxation itself lends itself very well to the parallel execution at the top level, since each of the graphs can be worked on independently. Finally, there is lot of parallelism in the node, arc, and path consistency algorithms that can also be exploited effectively.

\subsection{Parallelism in the relaxation process}

We essentially have a set of graphs which are independent, since they represent different types of constraints, although they share a set of common nodes. So, the consistency (node, arc, and path) tests can be performed in each of the graphs simultaneously. In the example in \ref{example-section} we have four types of constraints, and hence we can have processes working concurrently trying to enforce consistency in the four graphs: parallel, perpendicular, longer, and equal-length. Figure \ref{split-algorithm} gives a top level description of this process.

```
\begin{figure}
\begin{tabbing}
{\bf main}\=()\\
\{\\
\> buil\=d-graphs();\hspace*{1.75in}\= /* Build model and image graphs */ \\
\> state = INIT; \>\> /* Initialize state */\\
\> status = CHANGE;\> \> /* Set the status value */ \\
\> for (i=1; i$<$n; i$++$)\\
\>\> create-process(child,i); \> /* Create one process per graph */ \\
\> whil\=e (status != NOCHANGE) \{ \> /* Repeat until no change */ \\
\>\> counter = 0; \> /* Initialize counter to zero */\\
\>\> status = NOCHANGE; \> /* Reset the status */ \\
\>\> state = WORK; \> /* let the children work */ \\
\>\> while (counter $< n-1$) ; \> /* Wait for all children to start */\\
\>\> state = SYNC; \> /* this is for synchronization */\\
\>\> counter = 0; \> /* reinitialize counter */\\
\>\> consistency(0); \> /* work on own graph */ \\
\>\> while (counter $<$ n-1) ; \> /* Wait for children to sync */ \\
\>\}\\
\>state = QUIT; \>\> /* Kill child processes */\\
\}
\end{tabbing}
\begin{tabbing}
{\bf child}\=(index)\\
\{\\
\> whil\=e (state == INIT) ; \hspace*{1.0in}\= /* Wait for signal */\\
\> while (state != QUIT) \{ \>\> /* Work until parent signals QUIT */\\
\>\> while (state != WORK); \> /* Wait for WORK signal */\\
\>\> consistency(index); \> /* consistency checks */ \\
\>\> AtomicAdd(counter,1); \> /* Update counter */ \\
\>\> while (state == WORK) ; \> /* Wait for others to finish */\\
\>\> AtomicAdd(counter,1); \> /* used for synchronization */\\
\>\}\\
\}
\end{tabbing}
\caption{Structure of parallel split-level relaxation}
\label{split-algorithm}
\end{figure}
```

The {\bf main} process creates $r-1$ processes, where r is the number of constraint types. It synchronizes the children, and also does work on one graph. The procedure {\it consistency(i)} does the node, arc, and path consistency in the i^th graph. Three shared variables are used for synchronization of the processes: {\it counter}, {\it state}, and {\it status}. If any label is deleted while doing the consistency checks, {\it consistency} procedure sets the {\it status} flag to CHANGE. {\it AtomicAdd} does the addition atomically. This is a synchronous algorithm
and they have to be synchronized after each iteration. It can be proved that there is no deadlock in this scheme, although there are some spin waits.

It is easy to show that this scheme is indeed correct, i.e, it always produces the correct result when it terminates.

\newtheorem{theorem}{Theorem}
\begin{theorem}
: If a label {\em l} is removed from the label set of any node {\em u} during the parallel split-level relaxation process, then the node {\em u} cannot have the label {\em l} in any complete consistent labeling.
\end{theorem}
{\bf Proof} : If the label l is removed from by some process i during the process, then l is not a consistent label in the i^th graph. Since, it is necessary for the label to be consistent in all the graphs (i.e., to obey all the constraints of all the types), it can't be a part of complete consistent labeling. It doesn't make a difference when a label gets deleted
during the relaxation nor by which process.

\begin{theorem}
: If a label {\em l} is {\em not}\/ removed from the label set of any node {\em u}\/ during the parallel split-level relaxation process, then it doesn't violate any consistency tests.
\end{theorem}
{\bf Proof} : It is obvious from the structure of the relaxation process.

The two theorems prove that the parallel split-level relaxation
process will produce the exact same result as the sequential one. However, it will arrive at the result much faster.

\subsection{Parallelism in the consistency algorithms}

In this section it is shown that there is a lot of parallelism in the consistency algorithms. Several ways to take advantage of this are also described.

It is trivial to see that the node consistency can be done in \bigo{1} time, if mn processors are available. This can be done by doing the consistency checks for each label in each node in parallel. There is also parallelism in the path consistency algorithms, but we have not tried to exploit it, since in general it not clear if the cost of using it is worth the effort. Most of our
effort has been devoted to finding and exploiting the parallelism in the arc consistency algorithms.

Parallelism in arc consistency algorithms has been explored in detail in \cite{Samal87-book} and
\cite{Samal-ijpp}. Here we present only the relevant results and a sample parallel version of one of the algorithms.

It has been proved in \cite{Samal87-book} that any parallel algorithm doing arc consistency will at least take \bigo{mn} time, where n and m are the number of units and the number of labels per unit, respectively. It is also shown that the parallel versions of AC1, AC3, and AC4 (called PAC1, PAC3, and PAC4,

respectively) are all \bigo{mn} algorithms which means that they are all optimal. However, their performance in the average case may be very different. Their computational requirements are also very different. The PAC1 (parallel algorithm for AC1) always needs \bigo{em^2} (e is the number of arcs in the graph) processors to attain maximum speedup, but the PAC3, and PAC3 algorithms need fewer processors on the average. In the worst case, however, they all need the same number of processors.

Next we briefly sketch the structure of a parallel algorithm doing AC3. It is shown in Figure \ref{pac3-algorithm}. The
difference between AC1 and AC3 is that only the nodes whose label sets are changed contribute to the arcs to be examined next. So, the number of arcs, which could be viewed as a task, varies from iteration to iteration. Thus a queue which maintains the list of current tasks is very suitable. However, the {\it enqueue} and {\it dequeue} operations should be done atomically to maintain the correct semantics.

```
\begin{figure}
\begin{tabbing}
{\bf main}\=()\\
\{\\
\> buil\=d-graphs();\hspace*{1.75in}\= \kill
\> state = WORK; \>\> /* Initialize state */\\
\> TaskQ = Make-Queue();\\
\> initialize-queue(); \>\> /* add the initial task descriptions */\\
\> for (i=1; i$<$n; i++)\\
\>\> create-process(child,i,TaskQ);\> /* Create one process per processor */ \\
\> while (state != QUIT \{ \>\> /* Work until Queue is empty */\\
\>\> task = Wait-Queue(TaskQ); \> /* Wait for a task description */\\
\>\> change = work(task); \> /* work on the task */\\
\>\> if (change) \\
\>\> \ \ \ \  add-tasks-to-queue(task,TaskQ);\> /* Add new tasks to queue */\\
\>\> if empty(TaskQ) state = QUIT; \> /* No more tasks; so quit */\\
\>\}\\
\}
\end{tabbing}
\begin{tabbing}
{\bf child}\=(index)\\
\{\\
\> buil\=d-graphs();\hspace*{1.75in}\= \kill
\>\> while (state != QUIT) \{ \> /* Work until parent signals QUIT */\\
\>\>task = Wait-Queue(TaskQ);\> /* Wait for a task description */\\
\>\> change = work(task); \> /* work on the task */\\
\>\> if (change) \\
\>\> \ \ \ \  add-tasks-to-queue(task,TaskQ);\> /* Add new tasks to queue */\\
\>\}\\
\}
\end{tabbing}
\caption{Structure of a parallel AC3 algorithm}
\label{pac3-algorithm}
\end{figure}
```

It should be noted that the termination criterion is slightly complicated. The processes should stop iff the queue is empty and no process is working (or will add to the queue). The {\it work} procedure essentially does the consistency checks as per the specification of the task. If there are any changes then {\it add-tasks-to-queue} adds new tasks to the queue. The parent after creating the child processes also waits for the tasks and works on them as well. Several details which are necessary for the algorithm to work correctly on a multiprocessor are omitted, but this should give an overall structure of the algorithm.

```
\section{Implementation details}
```

\label{implementation-section}

In this section we describe the details of the implementation of the split-level relaxation on the BBN Butterfly. Our machine has 18 nodes of which two nodes are used by the system and hence are unavailable. All the code was written using the low level Chrysalis primitive functions. We chose to do that in order to have more control over the process creation, task partitioning, memory management, etc.

There are several ways to divide the tasks in order to exploit the parallelism in the system, since it is so rich in parallelism. However, since we have only 16 processors available, we take a multi-sequential programming approach. The basic idea here is that we create exactly one process on each processor which runs until the whole program is terminated. Since process creation is expensive on the Butterfly, it is a good design decision. Each process structure looks like the {\it child} process in Figure \ref{split-algorithm}. It goes through several stages: work, synchronization, etc. One could design an asynchronous version of this algorithm, but our implementation is a synchronous one.

We use a {\it queue} (DualQ, in Butterfly terminology) to keep all the tasks that need to be executed. Each process waits for a task in the queue. Once it finds a task, it dequeues it, and does the work specified by the task. The granularity of the task
should be carefully chosen. If the granularity is too large then the time taken to synchronize at the end of each iteration may be large. If on the other hand it is too fine, the overhead costs in waiting for the queue etc. may be overwhelming. There are several ways to divide the tasks in our algorithms. We could make each task do the consistency checks on one whole graph. This becomes too large grained. On the other hand if we go down to the level of doing consistency checks for each label on each node it becomes too fine grained. We take an intermediate approach. The duty of each task in our system is to enforce consistency along one arc in a particlar type of graph. So, if there are r constraint-types and e arcs, there are re tasks in the system. Also, we chose to use the parallel version of AC1 as the underlying consistency algorithm. We didn't attempt do to path consistency.

Several low level Chrysalis function calls are very useful. Among them are Atomic_Add to do the atomic operations, Make_DualQ, Enq_DualQ, and Wait_DualQ to create and manipulate queues, etc. The memory is partitioned over all the processors to reduce memory contention. We use only one centralized queue whose address is informed to all the processes. This will be a bottleneck for a large multiprocessor.

\subsection{Test Problems}
The algorithms were tested on three types of graphs: cyclic graphs, complete graphs, and random graphs.

The cyclic graph problem is described in \cite{Samal87-book}. The nodes of the graph are connected to form a cycle, see Figure \ref{cycle-figure}. The
label set of node i is given by \{ x : jn + i, 1 $\leq$ j $\leq$ m \}, where there are m labels and n nodes. If the constraint used is the {\it greater-than} relation, only
one label is removed from the network in one iteration and hence it takes mn iterations to converge. There is no solution to this problem and hence, the label sets all go to nil. In addition to the {\it greater than} relation we
use two more constraint types. The result is still the same, but the convergence to the solution is faster.

```
\begin{figure}
\vspace{3in}
\caption{A network of constraints}
\label{cycle-figure}
\end{figure}

The complete graph is used to model the n-queens problem, using the row
placement strategy.  The possible labels for each queen are 1, 2, $\ldots$,
$N$.
The constraint is that no two queens should attack each other.  This can be
broken up into two constraints as follows. If two queens in $i^th$ and $j^th$
row are in columns $k$ and $l$, respectively, the two constraints are: $k \neq
l$ and $|i-j| \neq |k-l|$.  We use these two constraints for this problem.
If we don't
constrain the search space further, nothing will happen since, for every
position of queen {\it i} there is at least one position for queen {\it j}
which is compatible.  We chose to fix the position of two queens ($n^th$ and
$n-1^st$) such that they conflict and hence there is no solution.  This forces
the label sets of all units to go to nil.

The underlying graph in the n-queens problem is a complete graph; hence the
number of arcs is \bigo{n^2}.  In the case of the cyclic graph the number of
arcs is \bigo{n}.  So, we decided to test some graphs whose number of
arcs is between the two.  We generate a random graph with number of arcs
= {\it O}($n^{1.5}$).  They are connected depending on the results of a
random event, in this case on the value of a uniform random number generated.
The number of labels is the same as the number of units.

All these problems were tried for several problem sizes (number of units).
The results are shown for $n$ =  20, 30, 40, and 50.  The number of
labels in n-queens is automatically fixed by {\bf n}.  In the case of the
other two problems we chose it to be the same as $n$.

\subsection{Results}

The major goal was to see how much benefit is obtained by running the algorithm
on the Butterfly.  So the major yardstick we use here is the {\it speedup}
figure.  It is defined as follows:

\[
S_k(A) = \frac{Time\ taken\ by\ an\ algorithm\ A\ on\ 1\ processor}
{Time\ taken\ by\ the\ same\ algorithm\ A\ on\ {\em k}\ processors}
\]

In the ideal case one would like to get a linear speedup with a slope of one.
It is also very hard to obtain in practice.  The speedup as a function of the
number of processors for the three problems is given in Figures
\ref{Speedup-cyclic}, \ref{Speedup-complete}, \ref{Speedup-random}.
In the random graph problem, the maximum speedup obtained was about 13, using
all
16 nodes.  In the other two problems we achieve what is called {\it
super-linear} speedup, i.e., the speedup is greater than the number of
processors.  We achieve this in the cyclic graph problem and the complete graph
problem.  It happens in this case because of combinatorial implosion.  Since
the processes are working concurrently, a change effected by any one of them
is immediately seen by the others.  This may save some work for the parallel
algorithm which the sequential algorithm has to do.  In the case of complete
graph, we achieve super-linear speedup only for $n=50$ using 15 or 16
processors.

For smaller problem sizes, the speedup numbers are not very good since there
are fewer tasks and the time spent in synchronizing is high, particularly as
the number of processors goes up.  That explains the drastic drop in
```

```
performance of the algorithm for $n=20$.  As the problem size increases
however, the performance improves considerably.



\begin{figure}
\vspace{2.5 in}
%\special{psfile=cyclic.ps hoffset = 10}
\caption{Speedup figures for the cyclic graph}
\label{Speedup-cyclic}
\end{figure}


\begin{figure}
\vspace{2.5 in}
%\special{psfile=complete.ps hoffset = 10}
\caption{Speedup figures for the complete graph}
\label{Speedup-complete}
\end{figure}


\begin{figure}
\vspace{2.5 in}
%\special{psfile=random.ps hoffset = 10}
\caption{Speedup figures for the random graph}
\label{Speedup-random}
\end{figure}

\section{Conclusion and Future Research}

In this paper we analyzed the split-level relaxation technique
in a parallel processing framework.  It was shown that there is much
parallelism inherent in the algorithm that can be exploited.  The algorithm
was implemented on an actual multiprocessor and the results confirm this.

Although the results look good, the implementation can be made more efficient.
There are several ways to improve upon it.  Using multiple queues instead of
one centralized queue will reduce the memory contention, particularly in large
multiprocessors.  Also, switching to an asynchronous model should provide an
improvement in the performance.  Another direction for future research is to
explore parallelism in the path consistency algorithms.

\vfill\eject
\bibliography{/u/samal/bib/general,/u/samal/bib/parallel,/u/samal/bib/gp}
\vfill\eject
\pagenumbering{roman}
\setcounter{page}{1}
\tableofcontents
\end{document}
```