```
struct MatchPairsBody : ParallelLoopBody {
        MatchPairsBody(FeaturesMatcher &_matcher, const std::vector<ImageFeatures> &_features,
                       std::vector<MatchesInfo> &_pairwise_matches, std::vector<std::pair<int, int> > &_near_pairs)
            : matcher(_matcher), features(_features),
              pairwise_matches(_pairwise_matches), near_pairs(_near_pairs) {}

        void operator()(const Range &r) const {
            cv::RNG rng = cv::theRNG(); // save entry rng state
            const int num_images = static_cast<int>(features.size());
            for (int i = r.start; i < r.end; ++i) {
                cv::theRNG() = cv::RNG(rng.state + i); // force "stable" RNG seed for each processed pair

                int from = near_pairs[i].first;
                int to = near_pairs[i].second;
                int pair_idx = from * num_images + to;

                matcher(features[from], features[to], pairwise_matches[pair_idx]);
                pairwise_matches[pair_idx].src_img_idx = from;
                pairwise_matches[pair_idx].dst_img_idx = to;

                size_t dual_pair_idx = to * num_images + from;

                pairwise_matches[dual_pair_idx] = pairwise_matches[pair_idx];
                pairwise_matches[dual_pair_idx].src_img_idx = to;
                pairwise_matches[dual_pair_idx].dst_img_idx = from;

                if (!pairwise_matches[pair_idx].H.empty())
                    pairwise_matches[dual_pair_idx].H = pairwise_matches[pair_idx].H.inv();

                for (size_t j = 0; j < pairwise_matches[dual_pair_idx].matches.size(); ++j)
                    std::swap(pairwise_matches[dual_pair_idx].matches[j].queryIdx,
                              pairwise_matches[dual_pair_idx].matches[j].trainIdx);
                LOG(".");
            }
        }

        FeaturesMatcher &matcher;
```

```
        const std::vector<ImageFeatures> &features;
        std::vector<MatchesInfo> &pairwise_matches;
        std::vector<std::pair<int, int> > &near_pairs;

    private:
        void operator=(const MatchPairsBody &);
    };
void FeaturesMatcher::operator()(const std::vector<ImageFeatures> &features,
                                 std::vector<MatchesInfo> &pairwise_matches,
                                 const UMat &mask) {
    const int num_images = static_cast<int>(features.size());

    CV_Assert(mask.empty() || (mask.type() == CV_8U && mask.cols == num_images &&
mask.rows));
    Mat_<uchar> mask_(mask.getMat(ACCESS_READ));
    if (mask_.empty())
        mask_ = Mat::ones(num_images, num_images, CV_8U);

    std::vector<std::pair<int, int> > near_pairs;
    for (int i = 0; i < num_images - 1; ++i)
        for (int j = i + 1; j < num_images; ++j)
            if (features[i].keypoints.size() > 0 && features[j].keypoints.size() > 0 &&
mask_(i, j))
                near_pairs.push_back(std::make_pair(i, j));

    pairwise_matches.clear(); // clear history values
    pairwise_matches.resize(num_images * num_images);
    MatchPairsBody body(*this, features, pairwise_matches, near_pairs);

    if (is_thread_safe_)
        parallel_for_(Range(0, static_cast<int>(near_pairs.size())), body);
    else
        body(Range(0, static_cast<int>(near_pairs.size())));
    LOGLN_CHAT("");
}
```