Algorytmy zarządzania wątkami dążą do tego aby jakaś liczba wątków musi być działająca jednocześnie aby procesory były zajęte, jednak jednocześnie liczba tych wątków nie może być zbyt duża, żeby nie zajmować zbyt dużej ilości pamięci. Należy również wątki, które łączy jakaś relacja trzymać na jednym procesorze ponieważ można zminimalizować komunikację między nimi co zwiększy wydajność algorytmu. Są dwa sposoby próbujące pogodzić te wymagania kradzież pracy oraz podział pracy.

Podział pracy – za każdym razem gdy zostanie utworzony nowy wątek algorytm próbuje przenieść kilka wątków na inny procesor w celu rozłożenia pracy na niewykorzystywane wątki. Jednak przeniesienie wątku, który jest powiązany z innymi na inny procesor powoduje, że ten wątek będzie musiał na bieżąco informować inne wątki o postępach swojej pracy.

Kradzież pracy – w tym przypadku niewykorzystywane procesory przejmują inicjatywę i jakby wykradają wątki zajętym procesorom. W tym algorytmie migracja wątków ma miejsce rzadziej niż w podziale pracy, jeśli wszystkie procesy są zajęte w kradzieży pracy migracja nie występuje, natomiast w podziale pracy występuje zawsze. Początki idei kradzieży pracy miały miejsce gdy Burton i Sleep poszukiwali rozwiązania równoległego wykonywania programów funkcjonalnych. Ci programiści wskazali korzyści jakie daje kradzież pracy w odniesieniu do komunikacji i przestrzeni. Następnie Rudolph, Slivkin-Allalouf analizowali losową kradzież pracy dla niezależnego, równoważenia obciążenia pracy na komputerach równoległych. Następnie Karp i Zang analizowali losową kradzież pracy dla równoległego wstecznego wyszukiwania. Niedawno Zhang i Ortynski uzyskali akceptowalny wynik dla wymagań komunikacyjnych tego algorytmu.

Obliczenia równoległe mogą być postrzeganie jako instrukcje połączone ze sobą przez (czyli jedna instrukcja wywołuje inną itp.). Instrukcje są połączone na zasadzie kontynuacji czyli po wykonaniu jednej instrukcji wykonuje się następna wszystko w obrębie jednego wątku ( widok grafu są to strzałki wewnątrz wątku), może się zdarzyć, że jakaś instrukcja w wątku wywoła instrukcję w innym wątku, wówczas powstaje wątek potomny.

Continue edges – oznaczają instrukcje, które wykonywane są sekwencyjnie wewnątrz jednego wątku.

Spawn edges – instrukcja ta tworzy wątek dziecko, wskazuje pierwszą instrukcję wątku dziacka, jednak pierwsza instrukcja w wątku dziecku nie może się wykonać do czasu aż nie zostanie wykonana instrukcja tworząca wątek dziecko w wątku rodzicu.

Activation frame – podczas tworzenia wątku dziecka alokowana jest pamięć, w której instrukcje wątku będą przetrzymywać potrzebne do obliczeń dane, mówi się że wątek żyje, natomiast gdy ostatnia instrukcja wątku się wykona następuje zwolnienie pamięci i wątek umiera.

Join edges – instrukcje te mają na celu dostarczenie innym instrukcjom danych potrzebnych do obliczeń, producent czyli instrukcja, która produkują dane musi wykonać się wcześniej niż konsument czyli instrukcja, która odbiera te dane. Gdy instrukcja konsument jest wykonywana przed instrukcją producent to konsument zostaje zatrzymany i oczekuje na wykonanie producenta, gdy producent się wykona konsument zostaje wznowiony wznawianie wielu konsumentów jest problemem implementacji i może zostać rozwiązany na różne sposoby.

Program niedeterministyczny – jeżeli na wejściu generuje więcej niż jedno obliczenie wielowątkowe.

**Teoria zachłannego planowania** – T(X)<= T1/P+Tniesk. Gdzie:

X - plan wykonywany na procesorze P

T1 – praca obliczeń dla procesora jest to suma wszystkich instrukcji wykonywanych przez wątki.

Tniesk. – jest to długość krytycznej ścieżki czyli długość najdłuższej bezpośredniej ścieżki w dagu.

Głębokość stosu – jest to suma wszystkich zaalokowanych pamięci przez przodków danego wątku wraz z jego pamiecią.

W sekcji 3 pokaże się, że **algorytm zajętych liści** dla P procesorów, dla dowolnych dokładnych obliczeń wielowątkowych o pracy T1, krytycznej długości ścieżki Tniesk oraz głębokości stosu S1 plan wykonywania X realizuje czas T(X)<=T1/Tniesk + P oraz S(X)<=S1P jednocześnie. Algorytm zakłada globalną listę wszystkich żywych wątków, które są dostępne dla procesorów, kiedy tworzony jest wątek zostaje dodany do tej globalnej listy. Następnie gdy procesor potrzebuje pracy usuwa wątek z listy. Algorytm pozwala procesorom dodawać wątki do listy i je z nich usuwać, ignorowany jest efekt rywalizacji procesorów o dostęp do listy. Algorytm zaczyna z głównym wątkiem oraz ze wszystkimi procesorami w stanie bezczynności. Na początku każdego kroku procesor jest bezczynny albo ma jeden wątek pracujący. Wątki, które są bezczynne na początku każdego kroku próbują usunąć gotowy wątek z listy, jeżeli na liście liczba wątków gotowych jest większa niż równa liczbie bezczynnych procesorów to każdy procesor pobierze gotowy wątek i „zaciągnie go do roboty”. Procesor zawierający wątek wykonuje instrukcje wątku według następujących reguł:

1. Jeśli wątek Ta stworzył potomka wątek Tb wtedy wątek Ta kończy wykonywanie instrukcji w tym kroku i zwraca wątek Ta to globalnej listy. Procesor zaczyna następny krok pracując na wątku dziecku Tb.
2. Jeśli wątek Ta zostanie zablokowany, wtedy procesor kończy bieżący krok poprzez zwrócenie wątku Ta do listy globalnej. Procesor następny krok zaczyna w stanie bezczynności.
3. Jeśli wątek Ta ginie wtedy procesor kończy bieżący krok sprawdzając czy Tb czyli rodzic Ta nie ma potomka. Jeśli Tb nie ma potomka i żaden inny procesor nie pracuje na wątku Tb to procesor pobiera Tb z listy i w następnym kroku będzie pracował na wątku Tb, w przeciwnym razie procesor zaczyna kolejny krok w stanie bezczynności.

W ten sposób każdy wątek liść pracuje i wykonuje instrukcje na procesorze.

**Algorytm losowej kradzieży pracy** – algorytm ten zakłada, że każdy procesor ma swoją kolejkę z dwoma końcami: górny i dolny koniec. Procesor pobiera wątki z dolnego końca kolejki i pracuje na nich natomiast gdy mamy do czynienia z migracją wątku do innego procesora wtedy wątek zostaje usunięty z górnego końca kolejki, czyli wątek na samej górze może wyemigrować do innego procesora. Algorytm zakłada następujące reguły pracy procesorów:

1. Gdy wątek Ta tworzy dziecko Tb wtedy wątek Ta zostaje umieszczony na dolnym końcu kolejki i procesor rozpoczyna pracę na wątku Tb.
2. Jeśli wątek Ta zostanie zablokowany wówczas procesor sprawdza swoją kolejkę, jeśli kolejka nie jest pusta wtedy procesor usuwa z dolnego końca wątek i zaczyna na nim pracować, jeśli kolejka jest pusta wtedy procesor wybiera dowolny inny procesor i z górnego końca jego kolejki „kradnie”(usuwa) wątek i zaczyna na nim pracować.
3. Jeżeli wątek Ta ginie procesor postępuje jak w przypadku gdy wątek się zablokuje.
4. Gdy wątek Ta odblokuje zablokowany wątek Tb, nie gotowy wątek Tb zostaje umieszczony na dolnym końcu kolejki procesora.

Prześledzimy na podstawie gry w piłki i kosze opóźnienie jakie powstaje na skutek losowego i asynchronicznego dostępu do danych przez procesory.

Atomowy model dostępu – jest to model maszyny, który zostanie użyty do analizy algorytmu kradzieży pracy. Zakładając, że maszyną jest równoległy komputer, który posiada P procesorów a jego pamięć może być dzielona lub rozkładana. Analiza zakłada, że równoległy dostęp do tej samej danej jest kolejkowany przez przeciwników (konkurentów). W tym modelu podczas jednego kroku jedno żądanie jest obsługiwane a reszta jest kolejkowana przez konkurentów. Dla kolekcji oczekujących żądań przeciwnik wybiera, które żądanie ma być obsłużone a które kontynuują czekanie. Przeciwnik nie może nie wybrać żadnego żądanie znajdującego się w kolejce, musi wybrać jeżeli w kolejce znajduje się jakieś żądanie. Jeśli żądania są obsługiwane losowo przez liczbę P procesorów, oraz żądanie są zakolejkowane w P kolejkach, dla procesora dopuszcza się najwyżej jedno żądanie w danym kroku. Dlatego całkowity czas, w którym procesor czeka na żądanie jest proporcjonalny do liczby M żądań nie ma znaczenia który procesor obsługuje żądanie i nie ma znaczenia jak żądanie są rozłożone w czasie. Widać to w algorytmie piłek i koszy.

Algorytm jest grą pomiędzy przeciwnikami, w której piłki są wrzucane losowo do koszy. Liczba P jest to liczba piłek, liczba piłek jest równa liczbie koszy. Liczba M jest to całkowita liczba piłek wrzuconych do kosza przez przeciwników. Na początku piłki znajdują się w zbiorniku odseparowanym od koszy. W każdym kroku przeciwnicy wykonują 2 kroki:

1. Przeciwnik wybiera losową piłkę ze zbiornika ( usuwa piłkę ze zbiornika), wybiera losowo kosz i wrzuca ją do niego.
2. Przeciwnik sprawdza każdy kosz, dla każdego kosza, który zawiera przynajmniej jedną piłkę usuwa z kosza dowolną jedną piłkę i zwraca do zbiornika.

Gra się kończy gdy liczba M piłek zostanie wrzucona do kosza i zwrócona z powrotem do zbiornika. Ten algorytm modeluje obsługiwanie skradzionych żądań przez algorytm kradzieży pracy. Jeśli piłka znajduje się w zbiorniku to wtedy oznacza, że właściciel piłki nie utworzył jeszcze żądania kradzieży. Jeżeli piłka znajduje się w koszu to oznacza, że właściciel piłki utworzył żądanie kradzieży, to żądanie zostało zakolejkowane w kolejce właściciela kosza ale jeszcze nie zostało obsłużone. Jeśli piłka została usunięta z kosza i zwrócona do zbiornika to oznacza, że żądanie zostało obsłużone.

Zakłada się, że całkowite opóźnienie w powyższym algorytmie spowodowane wykonywaniem liczby M żądań przez procesory wynosi: O(M + PlgP + P(lg1/eps) przy prawdopodobieństwie co najmniej 1-eps. Spodziewane opóźnienie może wynosić co najwyżej M. Na początku można zaobserwować, że nie ma znaczenia który przeciwnik wyjmie piłkę z wszystkich koszów. Można założyć, że piłki w koszu są kolejkowane za pomocą kolejki FIFO. Czyli przeciwnik usuwa piłkę z początku kolejki a dodaje na koniec kolejki. Jeżeli w tym samym czasie do tego samego kosza została dodana większa niż 1 liczba piłek wtedy są one dodawane do kolejki w dowolnym porządku. Porządek FIFO nie wpływa na całkowite opóźnienie, także na opóźnienie nie wpływa która piłka jest usuwana z kosza i do którego kosza piłka jest wrzucana

D = sum (r=1 to p) deltar.

**Algorytm Kradzieży pracy –** wzięty będzie pod uwagę koszt komunikacji wykonywania w pełni wielowątkowych obliczeniach przy wykorzystaniu algorytmu kradzieży pracy. Mamy wielowątkowe obliczenia z pracą T1 i krytyczną ścieżką Tniesk. Zakłada się że spodziewany czas działania na P procesorach włączając planowanie wynosi T1/P+O(Tniesk.). Co więcej dla każdego eps>0 czas wykonywania na P procesorach wynosi: T1/P + O(Tniesk. + lgP + lg(1/eps)) z prawdopodobieństwem 1 – eps. Pokaże się, że całkowity koszt komunikacji podczas wykonywania algorytmu wynosi: O(PTniesk.(1+nd)Smax), gdzie nd jest to maksymalna liczba „join egdes” wychodzących z wątku do jego rodziców a Smax jest to największy rozmiar ramki aktywacji. Na każdym kroku algorytmu będzie zbierana liczba P dolarów po jednym z każdego procesora. W każdym kroku każdy procesor umieszcza swój dolar w jednym z trzech wiader, stosownie do ich akcji w danym kroku. Jeśli procesor wykonywał jakąś instrukcję w danym kroku, wtedy wstawia dolara do koszyka „praca”. Jeśli procesor w danym kroku podejmuje próbę kradzieży wtedy wrzuca dolara do kosza „kradzież” a jeśli procesor czeka na swoją kolej w kolejce wtedy wrzuca dolara do kosza „czekaj”. Należy określić granicę wykonywania poprzez ograniczenie liczby dolarów w każdym koszu na koniec wykonywania, należy zsumować dolary i podzielić tą liczbę przez ilość procesorów P. Najpierw należy ograniczyć liczbę dolarów w koszu „praca”.

Twierdzenie: wykonywanie w pełni wielowątkowych obliczeń z pracą T1 przez algorytm kradzieży pracy na komputerze z P procesorami zakończy się gdy w koszu „praca” będzie dokładnie T1 dolarów.

Dowód: procesor umieszcza dolar w koszu „praca” tylko wtedy gdy wykonuje on jakąś instrukcję. Dlatego jest T1 instrukcji w obliczeniach. Wykonywanie kończy się gdy jest dokładnie T1 dolarów w koszu. Należy wprowadzić pojęcie „round”. Jeśli podczas wykonywania algorytmu wystąpi duża liczba prób kradzieży pracy wtedy możemy zidentyfikować instrukcje, które to powodują tzw. Instrukcje opóźniające, każda z tych prób kradzieży jest inicjalizowana przez jakieś instrukcje, które znajdują się w tzw. Sekwencji krytycznej. Z tego wynika, że jest mało prawdopodobne aby instrukcje pozostawały krytyczne podczas skromnej liczby prób kradzieży. Również można założyć, że jest mało prawdopodobne aby wystąpiły sekwencje opóźniające. Cykl próby kradzieży jest ustawiony dla przynajmniej 3P prób kradzieży ale mniej niż 4P kolejnych prób kradzieży. Jeżeli próba kradzieży jest inicjowana w czasie kroku t w danej serii, to każda próba kradzieży inicjowana w danym kroku jest inicjowana w tej samej serii. Można podzielić wszystkie próby kradzieży, które pojawiły się podczas wykonywania wewnątrz serii w następujący sposób:

Pierwsza seria zawiera wszystkie próby kradzieży zainicjonowanych w krokach 1,2,3… t1 gdzie t1 jest to najwcześniejszy czas gdzie co najmniej 3P prób kradzieży była zainicjalizowana w czasie t1 lub przed czasem t1. Mówi się, że pierwsza seria startuje w czasie kroku 1 gdzie (i+1) jest to seria, która zaczyna się w czasie (ti+1) i kończy się w najwcześniejszym czasie kroku ti+1 > ti + 1 gdzie co najmniej 3P prób kradzieży było inicjowanych w czasie pomiędzy ti + 1 a ti+1 włącznie. Te próby kradzieży pozostają w serii i+1. W myśl definicji, że każda seria zawiera co najmniej 3P kolejne próby kradzieży, ponieważ w większości P-1 prób kradzieży może być inicjalizowana w pojedynczym kroku czasowym, każda seria zawiera mniej niż 4P-1 próby kradzieży i każda seria wykonuje co najmniej 4 kroki. Sekwencje instrukcji, które tworzą sekwencje opóźniające są definiowane przestrzegając zwiększony dag uzyskany przez nieznaczną modyfikację oryginalnego daga. Oznaczmy oryginalny dag literą G, jest to dag składający się z instrukcji obliczeniowych jako wierzchołków, kontynuują one obliczenia i join edges to edges. Zwiększony dag G” jest sto dag G wraz z nowymi edges. Dla każdego zbioru instrukcji u, v i w (u,v) są to spawn edges a (u,w) są to continue edges, deque edges są umieszczone w G”. Wcześniej założono, że instrukcja w nie ma żadnych przychodzących join edges i G” jest dagiem. Jeżeli Tniesk. Jest to najdłuższa ścieżka w G to najdłuższa ścieżka w G” ma długość co najwyżej 2Tniesk. Warto zauważyć że G” jest tylko analitycznym narzędziem. Deque edges nie dają żadnego efektu w planowaniu i wykonaniu obliczeń przez algorytm kradzieży pracy. Deque edges są kluczem do definiowania instrukcji krytycznych. W dowolnym kroku czasowym podczas wykonywania mówi się że niewykonane instrukcje v są krytyczne, jeżeli każda instrukcja, która poprzedza v w G” została wykonana, to znaczy, jeżeli dla każdej instrukcji w znajduje się bezpośrednia ścieżka z w do v w G”, instrukcja w została wykonana. Krytyczna instrukcja musi być gotowa ponieważ G” zawiera każde edge z G, ale gotowe instrukcje mogą być krytyczne ale nie muszą.

Delay sequence jest 3-krotna (U,R,II) zgodną z poniższymi warunkami:

* U = (u1, u2, … , uL) jest to maksymalna bezpośrednia ścieżka w G”. odpowiednio dla i=1,2,…,L-1. Edge(ui, ui+1) należy do G”, instrukcja ui nie posiada edges poprowadzonych do niej, (instrukcja u1 musi być pierwszą instrukcją głównego wątku). Natomiast instrukcja uL nie posiada żadnych wychodzących edges w G” (instrukcja uL musi być ostatnią instrukcją głównego wątku).
* R jest to całkowita liczba cyklów prób kradzieży.
* II = (pi1, pi2, pi3, … , piL, pi’L) jest częścią liczby R (R = suma(i=1 do L gdzie(pii + pi’i)) przy czym pi’i należy do [0,1] dla każdego i=1,2,3, …, L.

Część II zawiera część sekwencji cyklów R. Pierwsza część partycji odpowiada pierwszym pi1 cyklom. Druga część odpowiada następnemu pi’1 kolejnym cyklom po pierwszym pi1 cyklach. Trzecia część odpowiada następnym pi2 kolejnym cyklom po (pi1 + pi’1) cyklom itd. Nas głównie będzie interesowało w tych częściach odpowiedni pii a nie pi’i, i w związku z tym zdefiniujemy i-tą grupę cykli, które są pii kolejnymi cyklami startującymi po r i-tym cyklu, gdzie ri = suma(od j=1 do i-1 gdzie(pij + pi’j)). Ponieważ II jest częścią R i pi’i należy do [0,1], dla każdego i=1,2,3,4, …, L mamy:

Suma(od i=1 do L pii ) >= R-L

Powiemy, że dany cykl próby kradzieży, który ma miejsce podczas instrukcji v jest krytyczny wtedy gdy wszystkie próby kradzieży, które zawiera cykl są inicjowane w kroku czasowym, w którym instrukcja v jest krytyczna. Inaczej v musi być krytyczna przez cały cykl. Delay sequence (U,R,II) mówi się, że występuje podczas wykonywania jeśli dla każdego i = 1,2, …, L wszystkie pii cyklr w i-tej grupie mające miejsce podczas instrukcji ui są krytyczne. Inaczej ui musi być krytyczny przez wszystkie cykle pii. Poniższe twierdzenie określa, że jeżeli co najmniej R cykli ma miejsce podczas wykonywania, to wtedy delay sequence (U,R, II) musi wystąpić. W szczególności jeśli spojrzymy na dowolny wykonywany program, w którym występuje co najmniej R cykli, to możemy zidentyfikować ścieżkę U = (u1, u2, … , uL) w dagu G” i część II = (pi1, pi2, …, piL, pi’L) dla pierwszych R cykli dla każdego i = 1,2,3, …, L wszystkie pii cykle w i-tej grupie występujące podczas instrukcji ui są krytyczne. Każdy pi’i wskazuje czy ui jest krytyczna na początku cyklu ale zostanie wykonany przed końcem cyklu. Jako, że cykl nie może być częścią żadnej grupy, ponieważ żadna instrukcja nie jest krytyczna.

Rozważmy wykonanie w pełni wielowątkowych obliczeń z długością ścieżki krytycznej Tnieks. przez algorytm kradzieży pracy na komputerze z ilością P procesorów. Jeśli co najmniej $PR prób kradzieży ma miejsce podczas wykonywania algorytmu, wtedy delay sequence(U,R,II) musi wystąpić.

Dowód: Dla danego wykonania algorytm, w którym ma miejsce co najmniej 4PR prób kradzieży, stworzymy dealy sequenct(U,R,II) i pokażemy, że ona się pojawi. Gdy mamy co najmniej 4PR próby kradzieży musi być co najmniej R cykli. Skonstruujemy delay sequence poprzez identyfikację zbioru instrukcji na bezpośredniej ścieżce w G”, z tego wynika że dla każdego kroku czasowego podczas wykonywania algorytmu jedna z tych instrukcji jest krytyczna. Dlatego podzielimy pierwsze R cykli stosownie to tego kiedy dany cykl występuje względnie kiedy każda instrukcja w ścieżce jest krytyczna.

Aby skonstruować ścieżkę U pracujemy w tył od ostatniej instrukcji do głównego wątku, który oznaczymy przezu1 Oznaczmy ul1, że jest to instrukcja poprzednika u1 w G’ z ostatnim czasem wykonania. Niech (ul1……u2, u1) oznaczają bezpośrednią ścieżkę z ul1 do u1 w G’. Zawężamy tą ścieżkę do pierwszej instrukcji wątku głównego poprzez iterację po tej konstrukcji: W i-tej iteracji mamy instrukcję uli w G’ z ostatnim czasem wykonywania i niech (uli+1………..ulli+1, uli) oznacza bezpośrednią ścieżkę z uli+1 do uli w G’. Kończymy iterację po konstrukcji kiedy dostaniemy iterację k w której ulk jest pierwszą instrukcją w głównym wątku. Nasza sekwencja jest następująca U = (u1, u2, …, uL), gdzie L=lk i ui = uL-i+1 dla i = 1,2, …, L. Można zweryfikować, że w każdym kroku czasowym wykonywania algorytmu jeden z uli jest krytyczny.